# URCap Gripper Driver

Universal Robots A/S

Version 1.13.0

**Abstract**

As of URCap API version 1.8, a driver framework is introduced for version 3.11.0/5.5.0 of PolyScope. This allows for adding functionality to PolyScope to support certain devices and having program nodes with UI as well as other contributions for free. This document will describe how to create a gripper driver with minimal Java code.

# Contents

**1 Introduction** **3**

**2 General Information about URCap Development** **3**

**3 Building a Gripper Driver** **3**
3.1 Mandatory Functionality . . . 3
3.2 Optional Functionality . . . 4
3.3 Capabilities (Optional) . . . 4
3.3.1 Parameter-based Capabilities . . . 4
3.3.2 Multi-gripper Capability . . . 5
3.4 Feedback Capabilities (Optional) . . . 7
3.5 Custom User Configuration (Optional) . . . 8
3.5.1 Checkbox . . . 9
3.5.2 Combo box . . . 9
3.6 Other Configuration (Optional) . . . 10
3.6.1 TCP Contribution . . . 11
3.6.2 Tool I/O Interface . . . 12

**4 Workflow** **12**
4.1 PolyScope Integration . . . 12
4.1.1 Startup . . . 12
4.1.2 Creating an Installation . . . 13
4.1.3 Loading an Installation . . . 13
4.2 Integrated Payload Support . . . 13
4.3 Script Code Generation . . . 13
4.3.1 Preamble . . . 14
4.3.2 Grip Action . . . 14
4.3.3 Release Action . . . 14
4.3.4 Call Scenarios . . . 15

**5 Tips** **15**

**6 Advanced** **15**
6.1 Backwards Compatibility . . . 16
6.2 Supporting Translations . . . 16

**7 Samples** **17**

**Appendices** **17**

**A Activator** **17**

**B Basic Sample** **17**

**C Advanced Sample** **19**

**D Dual Zone Sample** **21**

**E Dynamic Multi-Gripper Sample** **25**

## 1 Introduction

A driver in the context of PolyScope can be thought of as a way of having a specific device behaving in a uniform way when used in PolyScope.

In the case of a gripper driver, this means that PolyScope will provide a program node, a toolbar and a configurable installation node for programming, operating and setting up the gripper, respectively. Except for the installation node, everything behaves similarly to the end user regardless of the specific gripper being used.

Note the toolbar is not available on CB3 robots.

A gripper is defined as a device which can, as a minimum, grip and release. How it performs these actions is defined using script code. A gripper can optionally register the additional gripper capabilities it might support. Doing this opens up the program node for further configuration by the end user. The resulting user-defined parameters are accessible when generating script code for the given action (i.e. grip or release).

In the following sections, starting from section 3. Building a Gripper Driver, a basic gripper driver will be constructed and then expanded to support various physical capabilities.

## 2 General Information about URCap Development

The URCap SDK includes the separate *URCap Tutorial Swing* document.

Here you can find general information about development of URCaps, such as how to the specify your URCap's compatibility with the different robot series (section 12.1. Robot Series Compatibility) which is mandatory, and how to deploy and install a URCap using Maven (section 6. Deployment with Maven).

The sections 7. Contribution of an Installation Node, 8. Contribution of a Program Node and 9. Contribution of a Daemon are not relevant, if your URCap is a "pure" gripper driver URCap that does not include any additional installation node contribution(s), program node contribution(s) or daemon contribution(s).

## 3 Building a Gripper Driver

PolyScope uses the OSGi framework to run. This means that to register a gripper driver, it is necessary to construct a bundle with an `Activator`. In the `Activator`, an implemenetation of the `GripperContribution` interface must be registered within the OSGi framework.

When PolyScope starts, it will pick up the gripper driver and construct a program node, toolbar and an installation node. See appendix A. Activator for an example of this.

In the following subsections the implementation of the `GripperContribution`-interface will be covered.

### 3.1 Mandatory Functionality

All methods in the `GripperContribution` interface are mandatory to implement, but the implementation for some of them can be left blank.

The `getTitle(Locale)` method must return the title of the gripper. It is optional to use the `Locale` parameter depending on whether or not translations are supported. The title is shown in the program node UI and structure section of the Program tab, the installation node UI and the left-hand side navigation of the Installation tab as well as in the toolbar.

The method `generateGripActionScript(ScriptWriter, GripActionParameters)` must generate the necessary script code for gripping using the `ScriptWriter` parameter. If any capabilities are registered, the configuration of these will be accessible through the `GripActionParameters` parameter.

The `generateReleaseActionScript(ScriptWriter, ReleaseActionParameters)` method must generate the necessary script code for releasing using the `ScriptWriter` parameter. If any capabilities are registered, the configuration of these will be accessible through the `ReleaseActionParameters` parameter.

## 3.2 Optional Functionality

The `generatePreambleScript(ScriptWriter)` method can be used to generate any script code necessary for initializing the gripper. The script code will be added to the preamble section of the final script code generated for a robot program. It will also be used in conjunction with the script code generated for grip or release actions when operating the gripper using the toolbar buttons (as well as the Test button on the program node). If no script code is necessary, just leave the implementation of this method blank.

The `configureContribution(ContributionConfiguration)` method can be used to configure the properties of the contribution itself using the `ContributionConfiguration` parameter. In the current version of the API, only a custom logo for gripper can be registered. This is shown in the UI of the program and installation node as well as on the toolbar (e-Series only). The logo will automatically be scaled to fit in the mentioned places.

## 3.3 Capabilities (Optional)

A capability in the context of a gripper can be thought of as a physical property the gripper supports which can be configured by the user.

The `configureGripper(GripperConfiguration, GripperAPIProvider)` method can be used to register any capabilities the gripper may support using the `GripperConfiguration` interface. For access to PolyScope or robot domain specific information, use the `GripperAPIProvider` parameter.

The available capabilities in the API are described in the following subsections.

### 3.3.1 Parameter-based Capabilities

Capabilities which allow the end user to adjust different control parameters for a grip or release action are described below:

**Width** This capability can be used if the gripper supports moving to a fixed position. The width can be referring to the distance between the "fingers" of a gripper, but could also be a radius if the gripper is using a "three-finger" setup. The exact meaning of the capability is for the gripper manufacturer to decide and document. This capability applies to both grip and release actions.

**Force** This capability expresses that a gripper can operate with a certain gripping force. This capability only applies to a grip action.

**Speed** This capability can be used if the gripper supports moving its "fingers" with a certain speed (with respect to the gripper, not the movement in the environment). This capability applies to both grip and release actions.

**Vacuum** This capability can be used if the gripper operates using vacuum. A positive pressure range should be specified if the gripper operates using absolute pressure. A negative pressure range should be specified if the gripper uses a relative pressure. The user-specified value is then to be considered as the pressure below room pressure. This capability only applies to a grip action.

Each capability is registered using a range of operation as well as default values for grip and/or release actions. All numbers are interpreted in the specified unit. This is not necessarily the unit shown in PolyScope which is dependent on user preference. Any conversion between the specified unit and the unit displayed in the UI is handled automatically by PolyScope.

The capability instance (e.g. the `WidthCapability` interface) returned when registering a capability can be used to adjust the ranges dynamically if needed. This could be necessary, if the gripper has the option of swapping out gripping "fingers" for an extended range. This should be controlled by a checkbox or alternatively a combo box in the installation node, where the user can configure the exact setup of the gripper. The capability can then be adjusted accordingly depending on the settings.

Registering a capability adds a slider and input field to the program node UI, where the user can adjust the parameter for the gripper action.

Some capabilities are only for grip or release actions and these will only have one default parameter and only appear in the UI of the program node when the corresponding action has been chosen. Any combination of capabilities is allowed, although not likely to occur.

If the gripper, for instance, supports force gripping, the following code could be added:

```
gripperConfiguration.getGripperCapabilities().registerGrippingForceCapability
    (0, 10, 10, Force.Unit.N);
```

This will display the Force slider and input field on the Gripper program node screen. The slider's minimum and maximum values are 0 and 10 Newton, respectively. The default gripping force is 10 Newton. This value is used when showing the UI initially as well as the force parameter value for grip actions executed using the toolbar.

### 3.3.2 Multi-gripper Capability

A gripper can physically be constructed in a way where it is logically partitioned in multiple zones, or a number of separate, identical grippers can be mounted on the robot at the same time. In both cases, it can be thought of as a multi-gripper. If a gripper manufacturer chooses to support this, the gripper driver can register this as a capability. This will enable the end user to select which individual gripper or zone to operate. In the following when mentioning grippers, it can refer to both a set of separate, individual identical grippers or zones interchangeably.

Registering the multi-gripper capability adds a combo box to the toolbar and program node UI (when two or more gripper are enabled), where the user can select the gripper to use for a gripper action.

The multi-gripper capability is more advanced than the parameter-based capabilities described in previous section (3.3.1. Parameter-based Capabilities), but it is still registered using the same interface. However, due to its more advanced setup it is configured using a callback, more specifically an implementation of the `GripperListProvider` interface. This interface has a single method, `getGripperList(GripperListBuilder, Locale)`, where the full list of supported grippers must be supplied.

An example of the multi-gripper capability registration can be seen in listing 1 where two individual grippers are added.

Listing 1: Registering the multi-gripper capability

```
...
@Override
public void configureGripper(GripperConfiguration gripperConfiguration,
    GripperAPIProvider gripperAPIProvider) {
  gripperConfiguration.getGripperCapabilities().registerMultiGripperCapability
      (new GripperListProvider() {
    @Override
    public GripperList getGripperList(GripperListBuilder gripperListBuilder,
        Locale locale) {
      // Add two grippers with names "Gripper 1" and "Gripper 2"
      SelectableGripper gripper1 = gripperListBuilder.createGripper("
          gripper1_id", "Gripper␣1", true);
      SelectableGripper gripper2 = gripperListBuilder.createGripper("
          gripper1_id", "Gripper␣2", false);
      return gripperListBuilder.buildList();
    }
  });
}
...
```

The `getGripperList(...)` method is called once, so there is no option to add more grippers later on, however they can be created in an initial disabled state. A disabled gripper will not be available (nor visible) in PolyScope. This can be helpful if the gripper driver is meant to support both a use case with only a single gripper as well as supporting the option of mounting several separate, identical grippers on the robot at the same time (e.g. using on a special mounting bracket). In this case, the second gripper should still be added, but can initially be disabled, and then enabled when the end user is using a multi-gripper setup. Control of the enablement of the added grippers can be handled with a custom user configuration (see section 3.5. Custom User Configuration (Optional) for more details).

The individual grippers can be enabled and disabled dynamically using the method `setEnabled(SelectableGripper, boolean)` in the `MultiGripperCapability` instance returned when registering the multi-gripper capability.

When creating a gripper using the provided `GripperListBuilder` parameter, it is important to use a permanent identifier (id) for each gripper. The id is used for persistence, so changing it between versions of the gripper driver URCap will cause well-defined robot programs to become undefined thereby inconveniencing the end user.

A name must also be provided for each gripper which will be displayed in PolyScope (e.g. in the Gripper toolbar and program node). This can be localized if the gripper driver chooses to support translations (see section 6.2. Supporting Translations).

When the end user selects a gripper to use for a Gripper program node, the program tree title of the node will reflect that by having the selected gripper's name as prefix. The only exception is if the multi-gripper only has a single enabled gripper, in which case the prefix of the program node will be the title of the gripper driver itself.

A multi-gripper can support all other capabilities previously described in section 3.3.1. Parameter-based Capabilities. Each of the individual grippers will support any other (parameter-based) capability that has been registered, i.e. all the grippers will have the exact same set of capabilities.

The capabilities can be dynamically adjusted for all grippers or independently for each individual gripper. In a setup with two identical, separate grippers mounted simultaneously on the robot, it is possible to support only one of them having "fingers" with wide reach attached. This can be achieved by increasing the supported range of the registered width capability for that specific gripper only. Capability adjustments which apply to all the grippers are performed using the capability instance (e.g. the `WidthCapability` interface) returned when the capability was registered as described in the previous section 3.3.1. Parameter-based Capabilities.

Individual adjustments happen through the `MultiGripperCapability` returned when registering the multi-gripper capability. Calling the `getRegisteredCapabilities(SelectableGripper)` method, while specifying the gripper for which to adjust a capability, will return a `RegisteredCapabilities` instance. Here each registered capability can be accessed and adjusted for that particular gripper. Be careful to only access previously registered capabilities, as otherwise an exception will be thrown. Adjusting a capability for an individual gripper and afterwards making adjustments for all grippers will overwrite the individual settings.

Adding an independent, preconfigured TCP for each individual gripper is described in section 3.6.1. TCP Contribution. This section also describes what to be aware of regarding an existing single gripper TCP when adding multi-gripper support to an existing gripper driver URCap (how to migrate).

## 3.4 Feedback Capabilities (Optional)

A feedback capability in the context of a gripper is when the gripper is capable of feeding back information or status to PolyScope. In some cases, this opens up for additional opportunities for the end user to configure his program.

Feedback capabilities must be configured in the same method as regular capabilities, namely the `configureGripper(GripperConfiguration, GripperAPIProvider)` method.

Initially only the basic grip and/or release detected feedback capabilities can be supported by the gripper. Registering a feedback capability involves implementing a `ScriptCodeGenerator` that will supply the script code to be executed repeatedly by the controller. The supplied script code will be embedded in a script function so it must return a boolean (True if either grip or release was detected, False if either grip or release was not (yet) detected).

When registering the grip detected feedback capability this will add a toggle switch on the Gripper program node in PolyScope (when configured for grip). Toggling this will add two program nodes under the Gripper program node. The first is the 'Grip detected' program node, and the second is a 'Grip Timeout' node. Both nodes have the option to add nodes below it by toggling the 'Add action' toggle switch.

After the gripper action script code has been executed, a loop will repeatedly check for a grip detected until the configured timeout (in the 'Grip Timeout' program node) has been reached or a grip was in fact detected.

If a grip was detected, its children (if any) will now be executed, otherwise a timeout must have been reached, in which case the children (if any) of the 'Grip Timeout' program node will be executed.

The described behaviour also applies if the gripper supports and registers the release detected feedback capability, and is configured accordingly.

Given that the supplied script code runs often it must execute quickly. It must also be idempotent meaning that it cannot have any side effects when called multiple times. There are no guarantees from PolyScope's side as to how many or few times the script code is called.

The ScriptCodeGenerator to implement takes a different parameter type depending on the type of feedback capability being registered. The parameter type may contain relevant information for the script code generation, such as the selected gripper if the multi-gripper capability (see section 3.3.2. Multi-gripper Capability) has been registered. If this is the case, then the feedback script generator will be called once per gripper. Care must be taken to report back the correct feedback for the gripper specified in the parameter set.

An example of the Grip detected feedback capability registration can be seen in listing 2.

Listing 2: Registering grip detected feedback

```
...
@Override
public void configureGripper(GripperConfiguration gripperConfiguration,
    GripperAPIProvider gripperAPIProvider) {
  gripperConfiguration.getGripperFeedbackCapabilities().
      registerGripDetectedCapability(new ScriptCodeGenerator<
      GripDetectedParameters>() {
    @Override
    public void generateScript(ScriptWriter scriptWriter,
        GripDetectedParameters parameters) {
      scriptWriter.appendLine("return␣get_standard_digital_in(0)");
    }
  });
}
...
```

## 3.5 Custom User Configuration (Optional)

The configureInstallation(CustomUserInputConfiguration, ...) method can be used to register any custom user-defined input required for setting up the gripper. This could for instance be an IP-address, the gripper mounting or similar.

A number of widget types are supported, among other checkboxes, integer inputs and combo boxes.

All user inputs require an identifier (id) and a label. The id is the key used for storing the value in the underlying data model and the label is displayed next to the widget in the UI. It is important to never change the id of a user input, since this will break the persistence.

Non-user input widgets can also be added. These can be text labels (with an optional icon) and filler elements for controlling/grouping the layout of the UI.

The UserInput instance (or a sub type thereof) returned when a user input is registered should be stored locally in a class member field. The instance can be used when generating script code (either for the preamble section or a gripper action) using the getValue() method which will return the value selected by the user.

It is possible to listen for changes to the user input in case this should trigger an action. This is done by calling the `setValueChangedListener(ValueChangedListener<T>)` method supplying a value change listener. The change listener will be called when the user changes the user input as well as when a new installation is loaded or created.

It is also possible to attach an error validator for some of the user inputs (generally the ones where the user enters the input) if need be.

### 3.5.1 Checkbox

To register a user input for a checkbox, use the following code:

```
BooleanUserInput checkbox = customConfiguration.registerBooleanInput("Id", "
    Label", false);
```

The third parameter (with the value of `false`) is a default value. This value is only used if the user has not yet changed the value of the user input. In case the user changed the value, the checkbox is initialized using the persisted value.

### 3.5.2 Combo box

Registering a combo box user input is a little different. There are two ways of registering a combo box input depending on whether the initial selection is valid or not (the invalid selection is a string value guiding the user to make a selection).

A combo box has an initial selection (either valid or invalid), a list of elements to show in the drop-down list, and finally an `ElementResolver` instance to correctly identify and display the elements in the UI. On top of this, it has the required id and label described previously.

The role of the element resolver is to help persisting the selection in the data model. Since the data model does not support persisting the entire selected object (which can be any type), the element resolver will return a `String` identifier (id) for the selected element. This id is persisted and when an installation is loaded, it is used to select the correct element.

Each element in the list must have a unique id, which must be constant over time and versions of the URCap, otherwise loading an installation may result in the combo box being unable to correctly initialize with the persisted selection. This could also happen if the contents of the list changed since the selection was made and persisted. In any case, the combo box's selection will be invalid, and a program using this gripper will be undefined and unable to run. The user must select a valid element in the combo box to fix the issue.

The implementation of the element resolver can optionally override the `getDisplayName(T)` method. This gives the gripper the option to localize the UI or simply display a more meaningful name than the default implementation, which calls the `toString()` method on each element. The display name for the selected element is also stored in the data model, so it can be displayed in case the element is no longer in the list.

The code snippet in Listing 3 shows an example of the registration of a combo box with a custom element resolver.

Listing 3: Registering a combo box

```
...
  // Content displayed in the combo box
  private enum Mounting {
    TOOL_FLANGE("ToolFlange", "Normal"),
    ANGLE45("Angle45", "45␣degrees"),
    ANGLE_NEG_45("Angle-45", "-45␣degrees");

    private String id;
    private String name;

    Mounting(String id, String name) {
      this.id = id;
      this.name = name;
    }

    public String getId() {
      return id;
    }

    public String getDisplayName() {
      return name;
    }
  }
...

  //-----------------

    // Registering the combo box user input
    SelectableUserInput<Mounting> comboBoxInput = customConfiguration.
        registerComboBoxInput("Id", "Mounting", "Select...", Arrays.asList(
        Mounting.values()), new ElementResolver<Mounting>() {
      @Override
      public String getId(Mounting element) {
        return element.getId();
      }

      @Override
      public String getDisplayName(Mounting element) {
        return element.getDisplayName();
      }
    });
...
```

The resulting combo box displayed in PolyScope can be seen in figure 1.

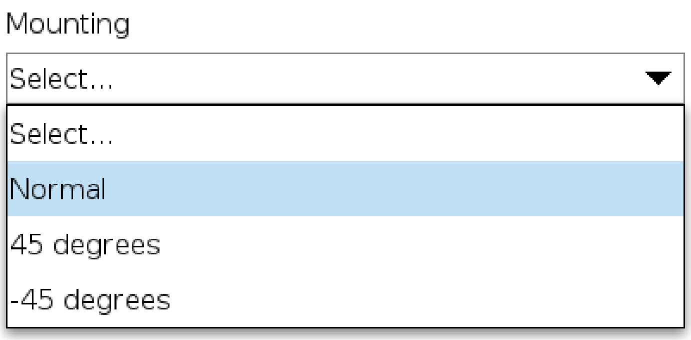

Figure 1: Combo box in PolyScope

## 3.6 Other Configuration (Optional)

This section describes configuration of the installation not directly related to the properties of the gripper itself. The configuration is performed in the `configureInstallation(...)` method using the different parameters as described in the following sections.

### 3.6.1 TCP Contribution

It is possible to help the user by preconfiguring the TCP of the gripper using the `TCPConfiguration` parameter of the `configureInstallation(...)` method. The TCP is added by calling the method `setTCP(String, Pose)` with a suggested name for the TCP and the pose for the offset of the TCP as input parameters. Use the `GripperAPIProvider` parameter to gain access to the pose factory capable of creating the needed pose for the TCP.

Any attempts to overwrite (calling the `TCPConfiguration` interface's `setTCP(String, Pose)` method when a TCP has already been added), remove or update an existing TCP are ignored when an existing installation is loaded. This is to ensure that the end user's previous settings are retained and not unintentionally modified.

Note that the class of a gripper driver contribution (implementing the `GripperContribution` interface) must *not* be renamed, if it contributes TCPs. Changing the class name between versions of the gripper driver URCap will make the TCPs inaccessible (not possible to remove and update) to the URCap as well as the user.

#### Multi-gripper TCPs

When the multi-gripper capability (see section 3.3.2. Multi-gripper Capability) has been registered, it is possible to assign each individual gripper its own TCP. This is done using the `TCPConfiguration` instance returned by the `getTCPConfiguration(SelectableGripper)` available in the `SystemConfiguration` parameter of the `configureInstallation(...)` method.

The TCPs associated to the multi-gripper will respect the enablement state of each gripper. This means that a disabled gripper will not have its TCP available in PolyScope. When the gripper is re-enabled, the TCP will become accessible again. The TCP for a single gripper (not registering the multi-gripper capability) does not have this functionality and will always be available. It is technically still possible to also add the single gripper TCP, however it is discouraged for this reason. Migrating a single gripper TCP to a set of multi-gripper TCPs will be described in the following section.

#### Migrating to a Multi-gripper

A special situation can arise when transitioning an existing gripper driver from a single gripper to a set of multiple grippers. If the single gripper driver previously contributed a TCP, that TCP could be used in various programs.

Programs referencing a TCP do so by name. This means that removing the single gripper TCP and replacing it with an identically named TCP assigned to one of the individual grippers will keep the end-user's programs working.

However, it is not possible to remove or update the contributed TCP after a new gripper driver with multi-gripper capabilities has been installed, since altering an installation is not allowed during load (as mentioned previously in 3.6.1. TCP Contribution). Instead, the migration of the single gripper TCP could occur as the result of a user action in a custom user configuration (see 3.5. Custom User Configuration (Optional)) after the installation has been loaded.

It is also a valid solution to *not* remain backwards compatible regarding the TCP and inform the user that upgrading to the new multi-gripper driver will have that consequence. Another option is to let the old single gripper TCP remain in the installation.

### 3.6.2 Tool I/O Interface

If the gripper is controlled and/or powered through the Tool I/O interface, a request to control the configuration of this interface must be made. This is done using the `SystemConfiguration` parameter.

Since some of the functionality is not available on CB3 robots, care must be taken to ensure the availability of the functionality before attempting to use it. To do this use the `CapabilityManager` interface to query the supported capabilities (note that these capabilities are referring to the robot and are not capabilities of the gripper).

After ensuring the necessary capabilities are available, a request to control the Tool I/O interface can be made. This happens through the `ControllableResourceModel` interface using the `requestControl(ToolIOInterfaceController)` method. The argument passed to the method will receive a callback if the end user assigns control to the gripper. When this happens the desired configuration of the Tool I/O interface can be applied.

Regardless of whether or not the user has granted the gripper the control of the Tool I/O interface, it is not permitted to control the settings directly via script code. The proper procedure is to have the user assign control, manipulate the settings in PolyScope in the callback described previously and let PolyScope generate the script code for applying the settings for the Tool I/O interface.

If read-only access to the Tool I/O interface is sufficient, simply use the `ResourceModel` interface provided by the aforementioned `SystemConfiguration` interface.

For more details on this, please see the separate *Resource Control* document.

# 4 Workflow

Different parts of the implementation of the `GripperContribution` interface correspond to different phases in PolyScope. In the following section these parts and their phases will be described in detail.

## 4.1 PolyScope Integration

The phases PolyScope go through that will influence the gripper driver are:

- Startup
- Creation of an installation
- Loading an installation

### 4.1.1 Startup

When PolyScope starts, it will look for any registered contributions as previously described. This happens only on startup, so only the instance registered in the OSGi framework will be used. This one and only instance is used for the rest of the lifecycle of PolyScope. This means that no state other than the user input configuration should be maintained. The implementation should therefore rely on PolyScope keeping its state in the underlying data model.

During the startup phase, all methods in the `GripperContribution` interface will be called, except for the methods responsible for generating script code.

### 4.1.2 Creating an Installation

When creating an installation, the methods `configureGripper(...)` and `configureInstallation(...)` are called again.

All user inputs will have the specified default values. Any value change listeners attached to the user inputs are not called (since technically there has been no change in value). This means that any state change that would happen when the default value is set, should also be the starting state of the gripper (e.g. if a checkbox determines whether a TCP should be offset with a camera ring, for instance, then the TCP pose offset should reflect the default value of the checkbox).

### 4.1.3 Loading an Installation

When loading an installation, the same methods are called as when creating a new installation. However, the default value is not being used, if the user has previously modified the value. In this case, the user input will have the persisted value restored and a call to the attached value change listener will be made, so any dependent state can reflect the value (e.g. same scenario as when creating an installation, only here the call will be made and the TCP pose offset can be adjusted according to the actual value of the checkbox).

## 4.2 Integrated Payload Support

As of PolyScope version 3.13/5.8, the UI of a Gripper program node will contain an option to set the total payload mass after the execution of a gripper action. With the release of PolyScope version 5.11, this payload setting is extended with the option of either specifying a payload mass or selecting a payload from the installation. Selecting a payload from the installation will in addition to applying the payload's mass also apply the corresponding center of gravity (CoG) and inertia matrix (defined for the payload in the installation). The gripper node's payload option will encourage the end user to operate the robot with a correct payload setting when gripping or releasing an object.

Note that applying the new payload is **not** the responsibility of the gripper driver, since this is automatically handled by PolyScope.

To get the timing right around when to apply the new payload, care must be taken when generating script code for gripper actions. This will be described in the following section.

## 4.3 Script Code Generation

The main part of a gripper driver is its script code generation. This determines how the gripper is initialized and operated. The script code generation consists of three separate parts:

- Preamble
- Grip action
- Release action

The gripper preamble is included in the normal preamble of a robot program and the configured gripper action is included in the appropriate place in the robot program.

When the gripper action has finished executing, PolyScope will set the new payload immediately after, if a payload has been specified by the end user. The script code for the gripper action must not finish earlier than when it is appropriate to apply the new payload (i.e. when the object has been gripped or released). This typically means that the script code should wait a time period corresponding to opening/closing the gripper's "fingers" fully (if the gripper is vacuum operated then long enough to achieve some level of vacuum).

If the gripper supports the grip and/or release detected feedback capability and the end user has enabled this functionality (in the Gripper program node), the payload will only be set if the grip/release detection was successful. In this case, the payload will be applied immediately after a grip or release is detected. When grip/release detected feedback is supported, it is not necessary to wait since the built-in detection timeout functionality will handle the waiting part. However, if the end user chooses *not* to enable grip/release detection, the gripper should behave as described in the first part (previous paragraph).

The enablement state of grip/release detection is available when generating script code for grip and release actions. See sections 4.3.2. Grip Action and 4.3.3. Release Action below for more details.

### 4.3.1 Preamble

The implementation of the `generatePreambleScript(ScriptWriter)` method should use the `ScriptWriter` parameter to initialize the gripper in such a way that each consecutive operation of the gripper (i.e. grip action or release action) can be executed without further initialization between each operation. The implementation of this method can reference any custom user input registered if necessary, but there are no capability parameters as these are tied to the actual gripper action execution.

### 4.3.2 Grip Action

When a grip action is to be executed, the `generateGripActionScript(..., GripActionParameters)` method is called. The implementation of this method can also reference any registered custom user input if necessary.

It will also have access to the relevant parameters for any grip capability registered through the provided `GripActionParameters` parameter. This includes the selected gripper if the multi-gripper capability (see section 3.3.2. Multi-gripper Capability) has been registered. It is not legal to retrieve a value for a capability not registered. Doing so will throw an exception.

If the grip detected feedback capability has been registered, the `isGripDetectionEnabled()` method in the `GripActionParameters` parameter can be used to determine if the end user has enabled or disabled this functionality.

### 4.3.3 Release Action

The `generateReleaseActionScript(..., ReleaseActionParameters)` method is called when a release action is to be executed. The implementation of this method can also reference any custom user input registered if necessary.

It will also have access to the relevant parameters for any release capability registered through the provided `ReleaseActionParameters` parameter. This includes the selected gripper if the multi-gripper capability (see section 3.3.2. Multi-gripper Capability) has been registered. It is not legal to retrieve a value for a capability not registered. Doing so will throw an exception.

If the release detected feedback capability has been registered, the `isReleaseDetectionEnabled()` method in the `ReleaseActionParameters` parameter can be used to determine if the end user has enabled or disabled this functionality.

Note that some capabilities are only available for grip actions.

### 4.3.4 Call Scenarios

The methods for generating script code described in previous sections are called in different situations and using different parameters. When mentioning default grip or release parameters, this is referring to the default values specified when registering a capability.

All the scenarios can be seen in table 1.

| Case | Script method | Parameters | Notes |
|---|---|---|---|
| Program run | preamble | | Normal program script code generation |
| Program run | grip/release | user configuration | Normal program script code generation |
| Test (on program node) | preamble + grip/release | user configuration[2] | Standalone program[1] |
| Toolbar grip | preamble + grip | default for grip[2] | Standalone program[1] |
| Toolbar release | preamble + release | default for release[2] | Standalone program[1] |

Table 1: Call Scenarios

# 5 Tips

In general, all method implementations should execute fast to provide a smooth PolyScope user experience. This is especially true for custom error validators attached to user inputs, since these get called each time the input changes (e.g. when the user enters something, the error validator is called for each keystroke).

Take an IP address input as an example. The user input itself validates by default that what the user inputs is in fact a valid IP address, but it could seem natural to ensure that the entered IP address is the address used by the device by pinging a server. Since this is time consuming, it might delay the user feedback in PolyScope. Consider solving it by adding a status label informing the user of whether the server on the entered IP address is responding. This way, a change listener could run a separate thread that checks whether the server is accessible on the IP address, and once this check is over update the status label.

Script code should also run fast and be responsive, so all the standalone programs (separate from the user's robot program) in table 1 execute fast. This means that the script code for the gripper should be the bare minimum needed. This will speed up both the generation itself, but also the compilation performed by the controller. It will also improve the user experience, if

[1]Separate from the user's robot program
[2]Grip or release detection will be off

the gripper actually reacts when a given button (generating a gripper action) is pressed and not several seconds later.

# 6 Advanced

In this section, a few advanced topics will be covered.

## 6.1 Backwards Compatibility

If the gripper driver evolves over time and any required custom user input changes, backwards compatibility must be considered. As a running example a checkbox determining whether or not a camera ring is mounted will be used.

In the first version of the gripper driver this is, as mentioned, simply a checkbox. In the next version of the gripper driver the manufacturer now supports a force/torque ring and the camera ring. The checkbox has to be deprecated, since the natural custom user input would now be a combo box. However, it is possible to help the user by preselecting the proper replacement in the combo box. The way to handle this is shown in the code snippet in listing 4.

If the checkbox was checked, the proper replacement is the 'Camera Ring' element in the combo box. On the other hand, if the checkbox was unchecked, the proper replacement is the 'No offset' element in the combo box. This is shown in the helper method `convertValue(Boolean)`.

When deprecating a user input, it will no longer be shown in the UI and its persisted value will be removed once the user saves the installation (but the value replacing it will now be stored). Note that the id of the user input being deprecated must not be reused for the new user input.

Listing 4: Deprecating a user input

```
...
@Override
public void configureInstallation(CustomUserInputConfiguration
    customConfiguration, SystemConfiguration systemConfiguration,
    TCPConfiguration tcpConfiguration, GripperAPIProvider apiProvider) {
  BooleanUserInput camera = customConfiguration.registerBooleanInput("Camera",
       "Camera ring mounted", false);
  Boolean cameraValue = customConfiguration.deprecateUserInput(camera);

  customConfiguration.registerPreselectedComboBoxInput("TCPOffset", "Select 
      additional mounting", convertValue(cameraValue), Arrays.asList(TCPOffset
      .values()), new ElementResolver<TCPOffset>() {
    @Override
    public String getId(TCPOffset element) {
      return element.toString();
    }
  });
}

private TCPOffset convertValue(Boolean cameraValue) {
  return cameraValue ? TCPOffset.CAMERA_RING : TCPOffset.NO_OFFSET;
}
...

//-------------------

public enum TCPOffset {
  NO_OFFSET,
  CAMERA_RING,
  FT_SENSOR
}
```

```
...
```

## 6.2 Supporting Translations

Supporting translations in a gripper driver only applies to custom user inputs, the title of the gripper and the name of each of the individual grippers if the multi-gripper capability (see section 3.3.2. Multi-gripper Capability) is supported. The rest is handled by PolyScope.

For the title, the `Locale` is provided as parameter to the `getTitle(Locale)` method in the interface `GripperContribution`.

For the custom user inputs, the `Locale` can be found in the `SystemSettings` interface which is accessed through the `SystemAPI` interface provided by the `GripperAPIProvider` interface. The only thing that should be translated are labels and the return value of the `getDisplayName(T)` method of an `ElementResolver` implementation (if using combo boxes).

For the name of each of the individual grippers in a multi-gripper, the `Locale` is provided as parameter to the `getGripperList(GripperListBuilder gripperListBuilder, Locale locale)` method in the `GripperListProvider` interface.

Never translate the identifier (id) of a user input or a gripper, nor change it from version to version of the gripper driver, as this will break the persistence of the settings for the gripper driver.

## 7 Samples

The appendices show listings with the code of an activator, a simple gripper and an advanced gripper and two multi-grippers. For further details of the included gripper driver samples, please see the 10.2. Driver Samples section in the separate *URCap Tutorial Swing* document.

# Appendices

## A Activator

Listing 5: Activator class registering a simple gripper driver

```java
package com.ur.urcap.examples.driver.gripper.simplegripper;

import com.ur.urcap.api.contribution.driver.gripper.GripperContribution;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

  @Override
  public void start(final BundleContext context) {
    context.registerService(GripperContribution.class, new SimpleGripper(),
        null);
  }

  @Override
  public void stop(BundleContext context) {
  }
}
```

## B Basic Sample

Listing 6: SimpleGripper class containing main functionality of the simple gripper driver

```
package com.ur.urcap.examples.driver.gripper.simplegripper;

import com.ur.urcap.api.contribution.driver.general.tcp.TCPConfiguration;
import com.ur.urcap.api.contribution.driver.general.userinput.
    CustomUserInputConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.ContributionConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.GripActionParameters;
import com.ur.urcap.api.contribution.driver.gripper.GripperAPIProvider;
import com.ur.urcap.api.contribution.driver.gripper.GripperConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.GripperContribution;
import com.ur.urcap.api.contribution.driver.gripper.ReleaseActionParameters;
import com.ur.urcap.api.contribution.driver.gripper.SystemConfiguration;
import com.ur.urcap.api.domain.script.ScriptWriter;

import javax.swing.ImageIcon;
import java.util.Locale;


public class SimpleGripper implements GripperContribution {

  private static final String GRIPPER_NAME = "Simple␣Gripper";

  @Override
  public String getTitle(Locale locale) {
    return GRIPPER_NAME;
  }

  @Override
  public void configureContribution(ContributionConfiguration configuration) {
    configuration.setLogo(new ImageIcon(getClass().getResource("/logo/logo.png
        ")));
  }

  @Override
  public void configureGripper(GripperConfiguration gripperConfiguration,
      GripperAPIProvider gripperAPIProvider) {
    // Intentionally left empty
  }

  @Override
  public void configureInstallation(CustomUserInputConfiguration
      configurationUIBuilder, SystemConfiguration systemConfiguration,
                     TCPConfiguration tcpConfiguration, GripperAPIProvider
                         gripperAPIProvider) {
    // Intentionally left empty
  }

  @Override
  public void generatePreambleScript(ScriptWriter scriptWriter) {
    // Intentionally left empty
  }

  @Override
  public void generateGripActionScript(ScriptWriter scriptWriter,
      GripActionParameters gripActionParameters) {
    scriptWriter.appendLine("set_tool_digital_out(0,␣False)");
    scriptWriter.appendLine("set_tool_digital_out(1,␣True)");
  }

  @Override
  public void generateReleaseActionScript(ScriptWriter scriptWriter,
      ReleaseActionParameters releaseActionParameters) {
    scriptWriter.appendLine("set_tool_digital_out(0,␣True)");
    scriptWriter.appendLine("set_tool_digital_out(1,␣False)");
  }
}
```

## C Advanced Sample

Listing 7: AdvancedGripper class containing main functionality of the advanced gripper driver

```
package com.ur.urcap.examples.driver.gripper.advancedgripper;

import com.ur.urcap.api.contribution.driver.general.script.ScriptCodeGenerator
    ;
import com.ur.urcap.api.contribution.driver.general.tcp.TCPConfiguration;
import com.ur.urcap.api.contribution.driver.general.userinput.
    CustomUserInputConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.ContributionConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.GripActionParameters;
import com.ur.urcap.api.contribution.driver.gripper.GripperAPIProvider;
import com.ur.urcap.api.contribution.driver.gripper.GripperConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.GripperContribution;
import com.ur.urcap.api.contribution.driver.gripper.ReleaseActionParameters;
import com.ur.urcap.api.contribution.driver.gripper.SystemConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.capability.
    GripDetectedParameters;
import com.ur.urcap.api.contribution.driver.gripper.capability.
    GripperCapabilities;
import com.ur.urcap.api.contribution.driver.gripper.capability.
    GripperFeedbackCapabilities;
import com.ur.urcap.api.contribution.driver.gripper.capability.
    ReleaseDetectedParameters;
import com.ur.urcap.api.domain.resource.ControllableResourceModel;
import com.ur.urcap.api.domain.script.ScriptWriter;
import com.ur.urcap.api.domain.value.simple.Force;
import com.ur.urcap.api.domain.value.simple.Length;
import com.ur.urcap.api.domain.value.simple.Pressure;
import com.ur.urcap.api.domain.value.simple.Speed;

import javax.swing.ImageIcon;
import java.util.Locale;


public class AdvancedGripper implements GripperContribution {

  private static final String GRIPPER_NAME = "Advanced␣Gripper";

  @Override
  public String getTitle(Locale locale) {
    return GRIPPER_NAME;
  }

  @Override
  public void configureContribution(ContributionConfiguration configuration) {
    configuration.setLogo(new ImageIcon(getClass().getResource("/logo/logo.png
        ")));
  }

  @Override
  public void configureGripper(GripperConfiguration gripperConfiguration,
      GripperAPIProvider gripperAPIProvider) {
```

```java
    GripperCapabilities gripperCapabilities = gripperConfiguration.getGripperCapabilities();

    registerForce(gripperCapabilities);
    registerWidth(gripperCapabilities);
    registerVacuum(gripperCapabilities);
    registerSpeed(gripperCapabilities);

    GripperFeedbackCapabilities fc = gripperConfiguration.getGripperFeedbackCapabilities();

    fc.registerGripDetectedCapability(new ScriptCodeGenerator<GripDetectedParameters>() {
      @Override
      public void generateScript(ScriptWriter scriptWriter, GripDetectedParameters parameters) {
        scriptWriter.appendLine("return get_standard_digital_in(0)");
      }
    });

    fc.registerReleaseDetectedCapability(new ScriptCodeGenerator<ReleaseDetectedParameters>() {
      @Override
      public void generateScript(ScriptWriter scriptWriter, ReleaseDetectedParameters parameters) {
        scriptWriter.appendLine("return get_standard_digital_in(1)");
      }
    });
  }

  @Override
  public void configureInstallation(CustomUserInputConfiguration configurationUIBuilder, SystemConfiguration systemConfiguration,
                    TCPConfiguration tcpConfiguration, GripperAPIProvider gripperAPIProvider) {
    ControllableResourceModel resourceModel = systemConfiguration.getControllableResourceModel();

    resourceModel.requestControl(new ToolIOController());
  }

  @Override
  public void generatePreambleScript(ScriptWriter scriptWriter) {
    // Intentionally left empty
  }

  @Override
  public void generateGripActionScript(ScriptWriter scriptWriter, GripActionParameters gripActionParameters) {
    System.out.println("Grip action :" + printCapabilityParameters(gripActionParameters));
  }

  @Override
  public void generateReleaseActionScript(ScriptWriter scriptWriter, ReleaseActionParameters releaseActionParameters) {
    System.out.println("Release action :" + printCapabilityParameters(releaseActionParameters));
  }

  private void registerWidth(GripperCapabilities capability) {
    capability.registerWidthCapability(40, 100, 50, 60, Length.Unit.MM);
  }

  private void registerForce(GripperCapabilities capability) {
    capability.registerGrippingForceCapability(0, 100, 40, Force.Unit.N);
  }

  private void registerVacuum(GripperCapabilities capability) {
    capability.registerGrippingVacuumCapability(0, 100, 70, Pressure.Unit.KPA)
        ;
  }

  private void registerSpeed(GripperCapabilities capability) {
    capability.registerSpeedCapability(0, 100, 40, 50, Speed.Unit.MM_S);
  }

  private String printCapabilityParameters(GripActionParameters
      gripActionParameters) {
    return "\n" +
        printWidthCapabilityParameter(gripActionParameters.getWidth()) + "\n"
            +
        printSpeedCapabilityParameter(gripActionParameters.getSpeed()) + "\n"
            +
        printForceCapabilityParameter(gripActionParameters.getForce()) + "\n"
            +
        printVacuumCapabilityParameter(gripActionParameters.getVacuum()) + "\n
            ";
  }

  private String printCapabilityParameters(ReleaseActionParameters
      releaseActionParameters) {
    return "\n" +
        printWidthCapabilityParameter(releaseActionParameters.getWidth()) + "\
            n" +
        printSpeedCapabilityParameter(releaseActionParameters.getSpeed()) + "\
            n";
  }

  String printWidthCapabilityParameter(Length width) {
    return "Width:␣" + width.getAs(Length.Unit.MM) + "␣mm";
  }

  String printSpeedCapabilityParameter(Speed speed) {
    return "Speed:␣" + speed.getAs(Speed.Unit.MM_S) + "␣mm/s";
  }

  String printForceCapabilityParameter(Force force) {
    return "Force:␣" + force.getAs(Force.Unit.N) + "␣N";
  }

  String printVacuumCapabilityParameter(Pressure vacuum) {
    return "Vacuum:␣" + vacuum.getAs(Pressure.Unit.KPA) + "␣kPa";
  }
}
```

# D Dual Zone Sample

Listing 8: DualZoneGripper class containing basic functionality of the dual zone gripper driver

```
package com.ur.urcap.examples.driver.gripper.dualzonegripper;

import com.ur.urcap.api.contribution.driver.general.tcp.TCPConfiguration;
import com.ur.urcap.api.contribution.driver.general.userinput.
    CustomUserInputConfiguration;
import com.ur.urcap.api.contribution.driver.general.userinput.
    ValueChangedListener;
import com.ur.urcap.api.contribution.driver.general.userinput.selectableinput.
    BooleanUserInput;

import com.ur.urcap.api.contribution.driver.gripper.ContributionConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.GripActionParameters;
```

```java
import com.ur.urcap.api.contribution.driver.gripper.GripperAPIProvider;
import com.ur.urcap.api.contribution.driver.gripper.GripperConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.GripperContribution;
import com.ur.urcap.api.contribution.driver.gripper.ReleaseActionParameters;
import com.ur.urcap.api.contribution.driver.gripper.SystemConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.capability.
    GripVacuumCapability;
import com.ur.urcap.api.contribution.driver.gripper.capability.
    GripperCapabilities;
import com.ur.urcap.api.contribution.driver.gripper.capability.multigripper.
    GripperList;
import com.ur.urcap.api.contribution.driver.gripper.capability.multigripper.
    GripperListBuilder;
import com.ur.urcap.api.contribution.driver.gripper.capability.multigripper.
    GripperListProvider;

import com.ur.urcap.api.domain.program.nodes.contributable.device.gripper.
    configuration.SelectableGripper;
import com.ur.urcap.api.domain.script.ScriptWriter;
import com.ur.urcap.api.domain.value.PoseFactory;
import com.ur.urcap.api.domain.value.simple.Angle;
import com.ur.urcap.api.domain.value.simple.Length;
import com.ur.urcap.api.domain.value.simple.Pressure;

import javax.swing.ImageIcon;
import java.util.Locale;


public class DualZoneGripper implements GripperContribution {

  private static final String GRIPPER_TITLE = "Dual␣Zone␣Gripper";

  private static final String ZONE_A_NAME = "Zone␣A";
  private static final String ZONE_B_NAME = "Zone␣B";
  private static final String ZONE_AB_NAME = "Zone␣A+B";

  private static final String ZONE_A_TCP_NAME = "Zone_A";
  private static final String ZONE_B_TCP_NAME = "Zone_B";
  private static final String ZONE_AB_TCP_NAME = "Zone_AB";

  // Ids must remain constant over time and versions of the Gripper URCap,
      since they are used for persistence and
  // can be used by other URCaps for configuring Gripper program nodes.
  private static final String ZONE_A_ID = "ZoneA_id";
  private static final String ZONE_B_ID = "ZoneB_id";
  private static final String ZONE_AB_ID = "ZoneAB_id";

  private static final String FRAGILE_HANDLING_LABEL = "Use␣Fragile␣Handling";
  private static final String FRAGILE_HANDLING_ID = "fragile_handling_id";

  private SelectableGripper zoneAGripper;
  private SelectableGripper zoneBGripper;
  private SelectableGripper zoneABGripper;
  private GripVacuumCapability gripVacuumCapability;

  private BooleanUserInput fragileHandlingInput;

  @Override
  public String getTitle(Locale locale) {
    return GRIPPER_TITLE;
  }

  @Override
  public void configureContribution(ContributionConfiguration configuration) {
    configuration.setLogo(new ImageIcon(getClass().getResource("/logo/logo.png
        ")));
  }

  @Override
  public void configureGripper(GripperConfiguration gripperConfiguration, GripperAPIProvider gripperAPIProvider) {
    GripperCapabilities capabilities = gripperConfiguration.getGripperCapabilities();

    capabilities.registerMultiGripperCapability(new GripperListProvider() {
      @Override
      public GripperList getGripperList(GripperListBuilder gripperListBuilder, Locale locale) {
        zoneAGripper = gripperListBuilder.createGripper(ZONE_A_ID, ZONE_A_NAME, true);
        zoneBGripper = gripperListBuilder.createGripper(ZONE_B_ID, ZONE_B_NAME, true);
        zoneABGripper = gripperListBuilder.createGripper(ZONE_AB_ID, ZONE_AB_NAME, true);

        return gripperListBuilder.buildList();
      }
    });

    gripVacuumCapability = capabilities.registerGrippingVacuumCapability(0, 100, 70, Pressure.Unit.KPA);
  }

  @Override
  public void configureInstallation(CustomUserInputConfiguration configurationUIBuilder,
                    SystemConfiguration systemConfiguration,
                    TCPConfiguration tcpConfiguration,
                    GripperAPIProvider gripperAPIProvider) {
    configureGripperTCPs(systemConfiguration, gripperAPIProvider);
    customizeInstallationScreen(configurationUIBuilder);
  }

  private void configureGripperTCPs(SystemConfiguration systemConfiguration, GripperAPIProvider gripperAPIProvider) {
    PoseFactory poseFactory = gripperAPIProvider.getPoseFactory();

    TCPConfiguration zoneATCPConfiguration = systemConfiguration.getTCPConfiguration(zoneAGripper);
    zoneATCPConfiguration.setTCP(ZONE_A_TCP_NAME, poseFactory.createPose(75, 0, 50, 0, 0, 0, Length.Unit.MM, Angle.Unit.DEG));

    TCPConfiguration zoneBTCPConfiguration = systemConfiguration.getTCPConfiguration(zoneBGripper);
    zoneBTCPConfiguration.setTCP(ZONE_B_TCP_NAME, poseFactory.createPose(-75, 0, 50, 0, 0, 0, Length.Unit.MM, Angle.Unit.DEG));

    TCPConfiguration zoneABTCPConfiguration = systemConfiguration.getTCPConfiguration(zoneABGripper);
    zoneABTCPConfiguration.setTCP(ZONE_AB_TCP_NAME, poseFactory.createPose(0, 0, 50, 0, 0, 0, Length.Unit.MM, Angle.Unit.DEG));
  }

  private void customizeInstallationScreen(CustomUserInputConfiguration configurationUIBuilder) {
    fragileHandlingInput = configurationUIBuilder.registerBooleanInput(FRAGILE_HANDLING_ID, FRAGILE_HANDLING_LABEL, false);
    fragileHandlingInput.setValueChangedListener(new ValueChangedListener<Boolean>() {
      @Override
      public void onValueChanged(Boolean useFragileHandling) {
        updateVacuumCapability(useFragileHandling);
      }
    });
  }

  // This method updates the parameters of the registered vacuum capability
      for all individual grippers
  private void updateVacuumCapability(boolean useFragileHandling) {
    if (useFragileHandling) {
      gripVacuumCapability.updateCapability(0, 70, 40, Pressure.Unit.KPA);
    } else {
      gripVacuumCapability.updateCapability(0, 100, 70, Pressure.Unit.KPA);
    }
  }

  @Override
  public void generatePreambleScript(ScriptWriter scriptWriter) {
    // Intentionally left empty
  }

  @Override
  public void generateGripActionScript(ScriptWriter scriptWriter,
      GripActionParameters gripActionParameters) {
    System.out.println("Grip␣Action␣:");

    printFragileHandlingSelection();
    if (fragileHandlingInput.getValue()){
      // Simulate applying fragile handling
      scriptWriter.appendLine("sleep(0.01)");
    }

    SelectableGripper selectedGripper = gripActionParameters.
        getGripperSelection();
    printSelectedGripper(selectedGripper);

    if (zoneAGripper.equals(selectedGripper)) {
      scriptWriter.appendLine("set_tool_digital_out(0,␣True)");
    } else if (zoneBGripper.equals(selectedGripper)) {
      scriptWriter.appendLine("set_tool_digital_out(1,␣True)");
    } else if (zoneABGripper.equals(selectedGripper)) {
      scriptWriter.appendLine("set_tool_digital_out(0,␣True)");
      scriptWriter.appendLine("set_tool_digital_out(1,␣True)");
    }
  }

  @Override
  public void generateReleaseActionScript(ScriptWriter scriptWriter,
      ReleaseActionParameters releaseActionParameters) {
    System.out.println("Release␣Action␣:");

    printFragileHandlingSelection();
    if (fragileHandlingInput.getValue()){
      // Simulate applying fragile handling
      scriptWriter.appendLine("sleep(0.01)");
    }

    SelectableGripper selectedGripper = releaseActionParameters.
        getGripperSelection();
    printSelectedGripper(selectedGripper);

    if (zoneAGripper.equals(selectedGripper)) {
      scriptWriter.appendLine("set_tool_digital_out(0,␣False)");
    } else if (zoneBGripper.equals(selectedGripper)) {
      scriptWriter.appendLine("set_tool_digital_out(1,␣False)");
    } else if (zoneABGripper.equals(selectedGripper)) {
      scriptWriter.appendLine("set_tool_digital_out(0,␣False)");
      scriptWriter.appendLine("set_tool_digital_out(1,␣False)");
    }
  }

```

```
  private void printSelectedGripper(SelectableGripper selectedGripper) {
    System.out.println("Selected␣Gripper:␣" + selectedGripper.getDisplayName()
         + "\n");
  }

  private void printFragileHandlingSelection() {
    System.out.println("Using␣Fragile␣Handling:␣" + fragileHandlingInput.
        getValue());
  }
}
```

## E Dynamic Multi-Gripper Sample

Listing 9: DynamicMultiGripper class containing basic functionality of the dynamic multi-gripper driver

```
package com.ur.urcap.examples.driver.gripper.dynamicmultigripper;

import com.ur.urcap.api.contribution.driver.general.tcp.TCPConfiguration;
import com.ur.urcap.api.contribution.driver.general.userinput.
    CustomUserInputConfiguration;
import com.ur.urcap.api.contribution.driver.general.userinput.
    ValueChangedListener;
import com.ur.urcap.api.contribution.driver.general.userinput.selectableinput.
    ElementResolver;
import com.ur.urcap.api.contribution.driver.general.userinput.selectableinput.
    SelectableUserInput;

import com.ur.urcap.api.contribution.driver.gripper.ContributionConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.GripActionParameters;
import com.ur.urcap.api.contribution.driver.gripper.GripperAPIProvider;
import com.ur.urcap.api.contribution.driver.gripper.GripperConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.GripperContribution;
import com.ur.urcap.api.contribution.driver.gripper.ReleaseActionParameters;
import com.ur.urcap.api.contribution.driver.gripper.SystemConfiguration;
import com.ur.urcap.api.contribution.driver.gripper.capability.
    GripperCapabilities;
import com.ur.urcap.api.contribution.driver.gripper.capability.
    MultiGripperCapability;
import com.ur.urcap.api.contribution.driver.gripper.capability.WidthCapability
    ;
import com.ur.urcap.api.contribution.driver.gripper.capability.multigripper.
    GripperList;
import com.ur.urcap.api.contribution.driver.gripper.capability.multigripper.
    GripperListBuilder;
import com.ur.urcap.api.contribution.driver.gripper.capability.multigripper.
    GripperListProvider;

import com.ur.urcap.api.domain.program.nodes.contributable.device.gripper.
    configuration.SelectableGripper;
import com.ur.urcap.api.domain.script.ScriptWriter;
import com.ur.urcap.api.domain.value.Pose;
import com.ur.urcap.api.domain.value.PoseFactory;
import com.ur.urcap.api.domain.value.simple.Angle;
import com.ur.urcap.api.domain.value.simple.Length;

import javax.swing.ImageIcon;
import java.util.Arrays;
import java.util.List;
import java.util.Locale;


public class DynamicMultiGripper implements GripperContribution {

  private static final String GRIPPER_TITLE = "Dynamic␣Multi-Gripper";
```

```java

	private static final String GRIPPER_1_NAME = "Gripper␣1";
	private static final String GRIPPER_2_NAME = "Gripper␣2";

	private static final String GRIPPER_1_TCP_NAME = "Gripper_1";
	private static final String GRIPPER_2_TCP_NAME = "Gripper_2";
	// Ids must remain constant over time and versions of the Gripper URCap,
	    since they are used for persistence and
	// can be used by other URCaps for configuring Gripper program nodes.
	private static final String GRIPPER_1_ID = "Gripper1_id";
	private static final String GRIPPER_2_ID = "Gripper2_id";

	private static final String MOUNTING_INPUT_ID = "mounting_id";
	private static final String MOUNTING_INPUT_LABEL = "Gripper␣Mounting";
	private static final String GRIPPER_1_FINGER_TYPE_ID = "
	    gripper1_finger_type_id";
	private static final String GRIPPER_2_FINGER_TYPE_ID = "
	    gripper2_finger_type_id";

	private SelectableGripper gripper1;
	private SelectableGripper gripper2;
	private MultiGripperCapability multiGripperCapability;

	private Pose gripper1TCPPose;
	private Pose gripper2TCPPose;
	private Pose singleGripperTCPPose;

	private SelectableUserInput<MountingType> mountingSelector;
	private SelectableUserInput<FingerType> gripper1FingerType;
	private SelectableUserInput<FingerType> gripper2FingerType;

	private enum MountingType {
	  SINGLE("single_id", "Single"),
	  DUAL("dual_id", "Dual");

	  private final String id;
	  private final String displayName;

	  MountingType(String id, String displayName) {
	    this.id = id;
	    this.displayName = displayName;
	  }

	  public String getDisplayName() {
	    return displayName;
	  }

	  public String getId() {
	    return id;
	  }
	}

	private enum FingerType {
	  STANDARD("standard_id", "Standard␣[0␣-␣60mm]", 0, 60, 10, 30),
	  EXTENDED("extended_id", "Extended␣[40␣-␣200mm]", 40, 200, 50, 70);
	  private final String id;
	  private final String displayName;

	  private final double minWidth;
	  private final double maxWidth;
	  private final double defaultGripWidth;
	  private final double defaultReleaseWidth;

	  FingerType(String id, String displayName, double minWidth, double maxWidth
	      , double defaultGripWidth, double defaultReleaseWidth) {
	    this.id = id;
	    this.displayName = displayName;
```

```
      this.minWidth = minWidth;
      this.maxWidth = maxWidth;
      this.defaultGripWidth = defaultGripWidth;
      this.defaultReleaseWidth = defaultReleaseWidth;
    }

    public String getId() {
      return id;
    }

    public double getMinWidth() {
      return minWidth;
    }

    public double getMaxWidth() {
      return maxWidth;
    }

    public double getDefaultGripWidth() {
      return defaultGripWidth;
    }

    public double getDefaultReleaseWidth() {
      return defaultReleaseWidth;
    }

    public String getDisplayName() {
      return displayName;
    }
  }

  @Override
  public String getTitle(Locale locale) {
    return GRIPPER_TITLE;
  }

  @Override
  public void configureContribution(ContributionConfiguration configuration) {
    configuration.setLogo(new ImageIcon(getClass().getResource("/logo/logo.png
        ")));
  }

  @Override
  public void configureGripper(GripperConfiguration gripperConfiguration,
      GripperAPIProvider gripperAPIProvider) {
    GripperCapabilities capabilities = gripperConfiguration.
        getGripperCapabilities();

    capabilities.registerWidthCapability(
        FingerType.STANDARD.getMinWidth(),
        FingerType.STANDARD.getMaxWidth(),
        FingerType.STANDARD.getDefaultGripWidth(),
        FingerType.STANDARD.getDefaultReleaseWidth(),
        Length.Unit.MM);

    multiGripperCapability = capabilities.registerMultiGripperCapability(new
        GripperListProvider() {
      @Override
      public GripperList getGripperList(GripperListBuilder gripperListBuilder,
          Locale locale) {
        gripper1 = gripperListBuilder.createGripper(GRIPPER_1_ID,
            GRIPPER_1_NAME, true);
        gripper2 = gripperListBuilder.createGripper(GRIPPER_2_ID,
            GRIPPER_2_NAME, false);

        return gripperListBuilder.buildList();
      }
    });
  }

  @Override
  public void configureInstallation(CustomUserInputConfiguration
      configurationUIBuilder,
                    final SystemConfiguration systemConfiguration,
                    TCPConfiguration tcpConfiguration,
                    GripperAPIProvider gripperAPIProvider) {
    configureGripperTCPs(systemConfiguration, gripperAPIProvider);
    customizeInstallationScreen(configurationUIBuilder, systemConfiguration);
  }

  private void customizeInstallationScreen(CustomUserInputConfiguration
      configurationUIBuilder,
                                          final SystemConfiguration
                                              systemConfiguration) {
    mountingSelector = configurationUIBuilder.registerPreselectedComboBoxInput
        (
        MOUNTING_INPUT_ID,
        MOUNTING_INPUT_LABEL,
        MountingType.SINGLE,
        Arrays.asList(MountingType.values()), new ElementResolver<MountingType
            >() {
          @Override
          public String getId(MountingType mountingType) {
            return mountingType.getId();
          }

          @Override
          public String getDisplayName(MountingType mountingType) {
            return mountingType.getDisplayName();
          }
        });
    mountingSelector.setValueChangedListener(new ValueChangedListener<
        MountingType>() {
      @Override
      public void onValueChanged(MountingType mountingType) {
        TCPConfiguration gripper1TCP = systemConfiguration.getTCPConfiguration
            (gripper1);

        if (mountingType == MountingType.SINGLE) {
          multiGripperCapability.setEnabled(gripper2, false);
          gripper1TCP.updateTCP(singleGripperTCPPose);
        } else if (mountingType == MountingType.DUAL) {
          gripper1TCP.updateTCP(gripper1TCPPose);
          multiGripperCapability.setEnabled(gripper2, true);
        }
      }
    });

    configurationUIBuilder.addFiller();
    configurationUIBuilder.addFiller();
    configurationUIBuilder.addFiller();

    List<FingerType> fingerTypes = Arrays.asList(FingerType.values());

    gripper1FingerType = configurationUIBuilder.
        registerPreselectedComboBoxInput(
        GRIPPER_1_FINGER_TYPE_ID,
        GRIPPER_1_NAME,
        FingerType.STANDARD,
        fingerTypes,
        new ElementResolver<FingerType>() {
            @Override
            public String getId(FingerType fingerType) {
              return fingerType.getId();
            }

            @Override
            public String getDisplayName(FingerType fingerType) {
              return fingerType.getDisplayName();
            }
          });
    gripper1FingerType.setValueChangedListener(new ValueChangedListener<
        FingerType>() {
      @Override
      public void onValueChanged(FingerType fingerType) {
        updateWidthCapability(gripper1, fingerType);
      }
    });

    gripper2FingerType = configurationUIBuilder.
        registerPreselectedComboBoxInput(
        GRIPPER_2_FINGER_TYPE_ID,
        GRIPPER_2_NAME,
        FingerType.STANDARD,
        fingerTypes,
        new ElementResolver<FingerType>() {
          @Override
          public String getId(FingerType fingerType) {
            return fingerType.getId();
          }

          @Override
          public String getDisplayName(FingerType fingerType) {
            return fingerType.getDisplayName();
          }
        });
    gripper2FingerType.setValueChangedListener(new ValueChangedListener<
        FingerType>() {
      @Override
      public void onValueChanged(FingerType fingerType) {
        updateWidthCapability(gripper2, fingerType);
      }
    });
  }

  // This method updates the parameters of the registered width capability for
       a specific individual gripper
  private void updateWidthCapability(SelectableGripper gripper, FingerType
      fingerType) {
    WidthCapability widthCapability = multiGripperCapability.
        getRegisteredCapabilities(gripper).getWidthCapability();

    widthCapability.updateCapability(
        fingerType.getMinWidth(),
        fingerType.getMaxWidth(),
        fingerType.getDefaultGripWidth(),
        fingerType.getDefaultReleaseWidth(),
        Length.Unit.MM);
  }

  private void configureGripperTCPs(SystemConfiguration systemConfiguration,
      GripperAPIProvider gripperAPIProvider) {
    createMountingTCPPoses(gripperAPIProvider);

    TCPConfiguration gripper1TcpConfiguration = systemConfiguration.
        getTCPConfiguration(gripper1);
    gripper1TcpConfiguration.setTCP(GRIPPER_1_TCP_NAME, singleGripperTCPPose);

    TCPConfiguration gripper2TcpConfiguration = systemConfiguration.
        getTCPConfiguration(gripper2);
    gripper2TcpConfiguration.setTCP(GRIPPER_2_TCP_NAME, gripper2TCPPose);
  }

  private void createMountingTCPPoses(GripperAPIProvider gripperAPIProvider) {
    PoseFactory poseFactory = gripperAPIProvider.getPoseFactory();

    singleGripperTCPPose = poseFactory.createPose(0, 0, 100, 0, 0, 0, Length.
        Unit.MM, Angle.Unit.RAD);
    gripper1TCPPose = poseFactory.createPose(50, 0, 80, 0, 0.61, 0, Length.
        Unit.MM, Angle.Unit.RAD);
    gripper2TCPPose = poseFactory.createPose(-50, 0, 80, 0, -0.61, 0, Length.
        Unit.MM, Angle.Unit.RAD);
  }

  @Override
  public void generatePreambleScript(ScriptWriter scriptWriter) {
    // Intentionally left empty
  }

  @Override
  public void generateGripActionScript(ScriptWriter scriptWriter,
      GripActionParameters gripActionParameters) {
    System.out.println("Grip␣Action␣:");

    // The mounting type could be used during script generation
    printMountingType();

    SelectableGripper selectedGripper = gripActionParameters.
        getGripperSelection();
    printSelectedGripper(selectedGripper);

    if (gripper1.equals(selectedGripper)) {
      // The selected finger type could be used during script generation
      printFingerType(selectedGripper);
      scriptWriter.appendLine("set_tool_digital_out(0,␣True)");
    } else if (gripper2.equals(selectedGripper)) {
      // The selected finger type could be used during script generation
      printFingerType(selectedGripper);
      scriptWriter.appendLine("set_tool_digital_out(1,␣True)");
    }
  }

  @Override
  public void generateReleaseActionScript(ScriptWriter scriptWriter,
      ReleaseActionParameters releaseActionParameters) {
    System.out.println("Release␣Action␣:");

    // The mounting type could be used during script generation
    printMountingType();

    SelectableGripper selectedGripper = releaseActionParameters.
        getGripperSelection();
    printSelectedGripper(selectedGripper);

    if (gripper1.equals(selectedGripper)) {
      // The selected finger type could be used during script generation
      printFingerType(selectedGripper);
      scriptWriter.appendLine("set_tool_digital_out(0,␣False)");
    } else if (gripper2.equals(selectedGripper)) {
      // The selected finger type could be used during script generation
      printFingerType(selectedGripper);
      scriptWriter.appendLine("set_tool_digital_out(1,␣False)");
    }
  }

  private void printSelectedGripper(SelectableGripper selectedGripper) {
    System.out.println("Selected␣Gripper:␣" + selectedGripper.getDisplayName()
        );
```

```
	}
	private void printMountingType() {
	  System.out.println("Mounting␣Type:␣" + mountingSelector.getValue().
	      getDisplayName());
	}

	private void printFingerType(SelectableGripper selectedGripper) {
	  if (gripper1.equals(selectedGripper)) {
	    System.out.println("Finger␣Type:␣" + gripper1FingerType.getValue().
	        getDisplayName() + "\n");
	  } else if (gripper2.equals(selectedGripper)) {
	    System.out.println("Finger␣Type:␣" + gripper2FingerType.getValue().
	        getDisplayName() + "\n");
	  }
	}
}
```