Aduct

Release 1.1.0

Jul 09, 2020

Contents:

CHAPTER 1

Introduction

Aduct is a toolkit to design graphical applications that can be dynamically changed with a little work as possible. It is designed by inheriting objects provided by Gtk and thus by following principles of Aduct with Gtk, one can make powerful applications that are easy for a developer to develop, third-party person to improve and end user to use.

1.1 The Need

Let us assume you make an amazing application. As its core developer, you design the interface in such a way that it looks awesome to you. In other words, it has an interface that reflects your ideas on how a graphical user interface should look. Apart from the code (which nobody cares when your application is closed-source), the *where and how* widgets are arranged in the user interface is also important (which everybody cares except you).

After designing the application, naturally you get more comfortable with the interface, so it doesn't look strange to you in any way. But this isn't the case with users. There are also certain challenges that you often need to overcome after publishing or improving the application :

- **You need to add a feature** New features become a nightmare when it involves a lot of rewrites. The problem with adding the new feature is the doubt if it will be accepted by users or whether they will work properly with other parts of application.
- **Allowing third-party plugins** The developer of the plugin may not have the same thinking you do. Sometimes by mistake, they completely spoil the working of your application. Also a plugin is not always cooperative with other plugins. It may inhibit the working of other plugins.
- **Impressing the users** Unfortunately there is also a good chance that your users may be totally against the interface of your application. For them, you need to have an interface that can be changed at their will. Some applications may even have a different layout of interface for each file. So you need to make an application whose interface can literally be saved and ported.

How can you tackle these hurdles? Let us try to describe the *sudden solutions* that comes to your mind.

For the first issue, the complexity depends on your code and interface. Depending on that, adding a new feature could be simple as adding a few lines of code or a complete rewrite of your entire application. But what if you want a solution that doesn't depend on the complexity of your application?

The second issue can be avoided by dividing your application interface into loosely coupled parts. Then you allow the plugins to affect only certain parts of the application. Mostly it is some panel situated at one edge of the application, where the plugin gets its place. In other words, you place a restriction on the scope of a plugin, so that it doesn't interfere with other blocks of application. If a plugin wants a central representation in your application, how can you achieve that? Can you design an approach that gives your plugin all the essential freedom?

The issue with users is the most important one to take care. To make an interface that can be easily changed by users is the most irritating. Sorting out the requirements, checking whether a feature *will be useful or not* is also difficult to resolve. For you, placing the toolbox at right seems plausible, but there could be a creepy user who wants it at bottom. Possibilities exist. Saving the interface is another headache. *Which widget was modified ?*, *which should be saved ?*, *how to save ?*, all questions makes you *think*.

There are more such issues which doesn't go easy with a developer.

1.2 A Solution

Aduct can be used to kick-out the above mentioned issues. Some developers may be think that such issues can be avoided by writing a few more lines of code. And yes, Aduct is such a package made with a few lines of code. It is very small, but when you follow its approach, it saves your essential time and resources, which can be instead used in improving other concepts of your application.

Aduct's design principle is very simple. You make widgets that are independent of each other. Make a basic layout of the interface, then sit back and relax, because Aduct now takes care of other requirements. We believe in *"take care of small things and big things will automatically be taken care"*. It should not be hidden that at first you may have trouble writing independent widgets, but once you are able to make one, then you hardly need to focus in its working with interface.

1.3 To The Point

Aduct is inspired by Blender's interface. Blender has such an awesome one where it is possible for a user to customize it in any way they want. Aduct, which tries to mimic the behavior, is written in Python using Gtk. We will cover the working and making of interfaces with Aduct later. For now, what you need to understand is that in Aduct, we have two things that work together. A provider that can produce widgets and a view, that can hold and display them. Views come with various tweaks, which just need to be *enabled*. When the requirements of both are satisfied, you get a good interface that can suit all the use cases. The description so far may seem so absurd and you may not have even able to get a single point. It's okay, continue reading, you will *understand*.

CHAPTER 2

Installation

Simply put, to use Aduct, you need to install it. The following guide should be able help you in getting Aduct. For advanced users and more configuration, you are always welcome to grab the source code of Aduct and do literally whatever you want.

2.1 Requirements

Aduct is written in Python and requires Python. Apart from that you need Gtk, nothing more. So we directly follow the guidelines required to use Gtk.

- **Python 3.5+** If not installed, get the latest version of Python.
- **Gtk 3.0+** The latest version of Gtk is recommended. The instructions to install it are highlighted in the official website.
- **PyGObject** PyGObject provides Python bindings to GI modules (that is Gtk and its friends). If this is your first try with Gtk, you may have to install it. Directly install it using PIP.

2.2 Getting Aduct

After the requirements are satisfied, installing Aduct won't be a trouble. The best way is to install it using PIP.

```
$ pip install Aduct
```

As already stated, for those not comfortable with PIP, you can always get the source code and do necessary things to get Aduct on your *PYTHONPATH*.

2.3 New to Gtk ?

It might be case, that you don't even know what is Gtk and perhaps be wondering how to get started. The following is a quotation from Wikipedia's page for Gtk.

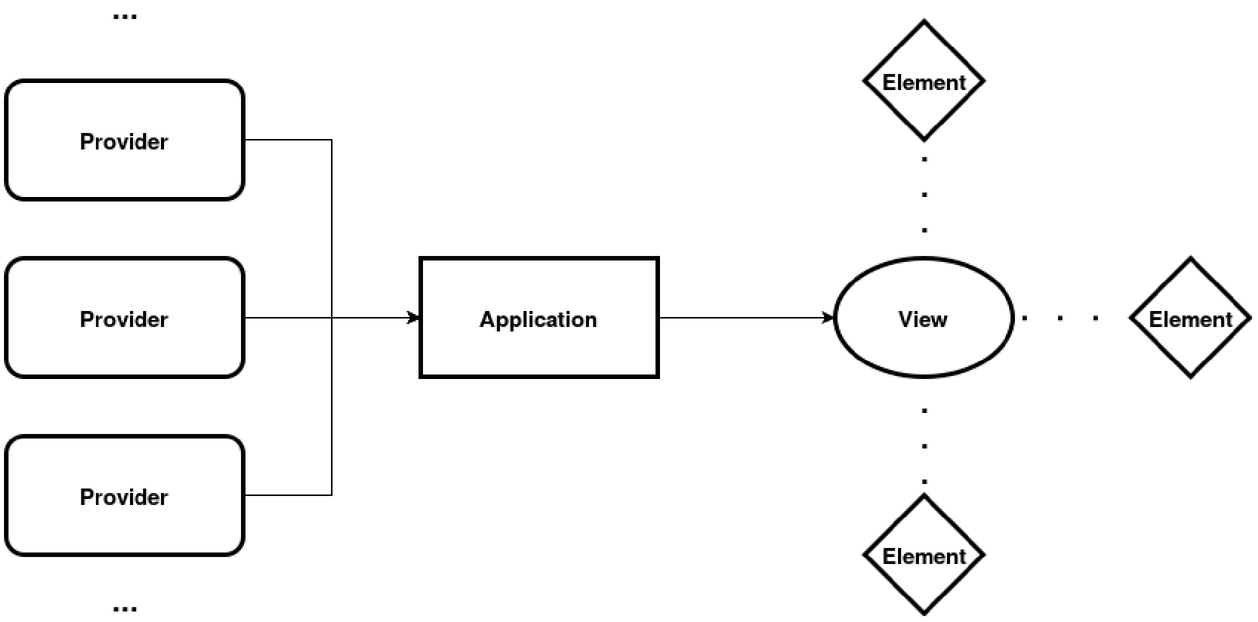
> GTK (formerly GTK+ GIMP Toolkit) is a free and open-source cross-platform widget toolkit for creating graphical user interfaces (GUIs). It is licensed under the terms of the GNU Lesser General Public License, allowing both free and proprietary software to use it. Along with Qt, it is one of the most popular toolkits for the Wayland and X11 windowing systems. (Source : Wikipedia)

Learning the basics of Gtk will be easy with Python. To get familiar with developing Gtk applications, please have a look at the official docs.

CHAPTER 3

Overview

Let us now learn how to design a flexible interface using Aduct. It isn't any difficult, so let's get started. An interface designed with Aduct is backed up by three things; providers, views and elements. The following sections describe them in an elaborate manner.

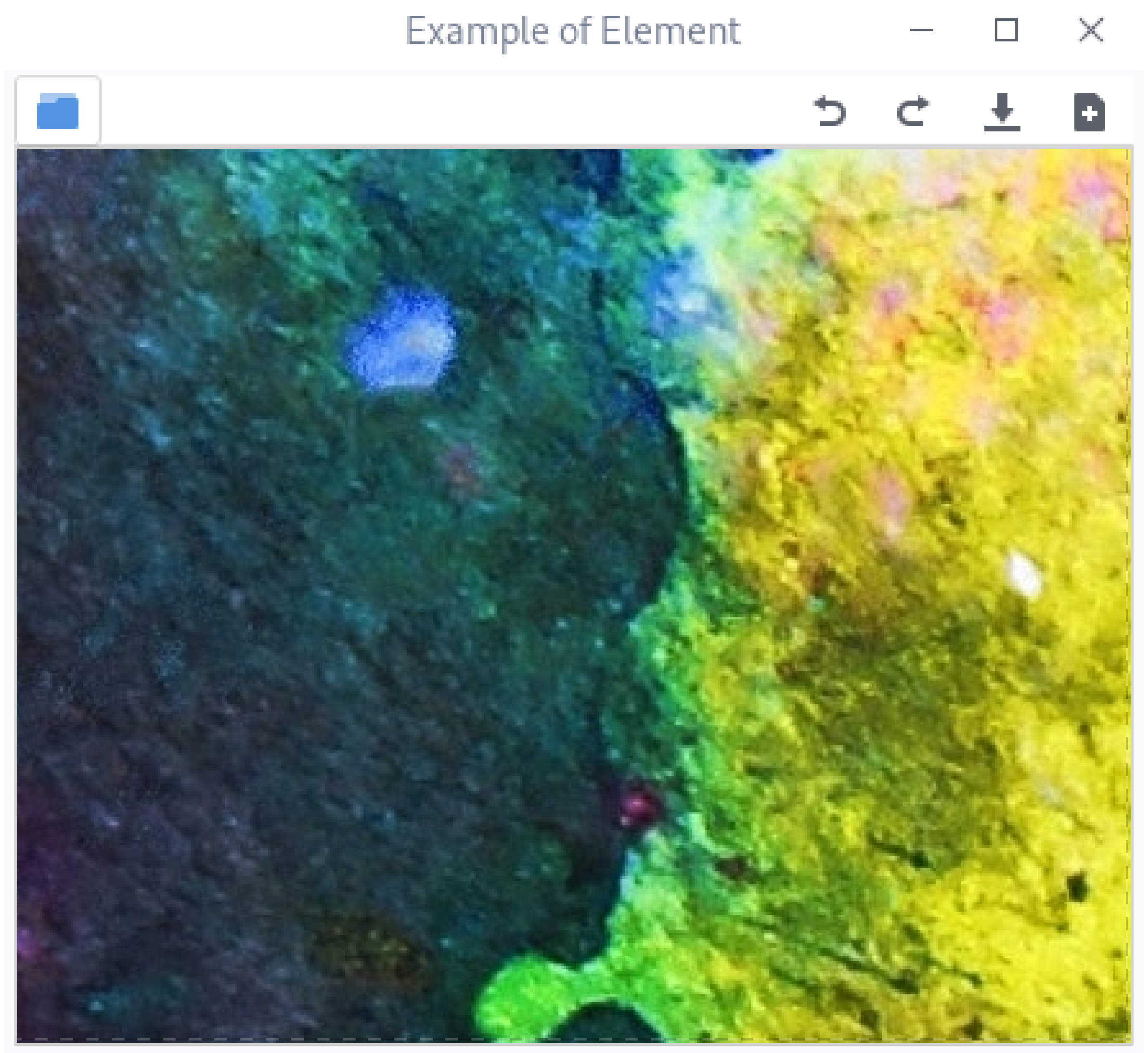


3.1 Aduct.Provider

Provider is an object that can produce widgets. They can be taken as a factory that can assemble and give you widgets when required. The provider owns every information about the widget it produces, so it can easily produce duplicate widgets and also control them as required. Apart from assembling widgets, they can also dissemble a widget when required. A dissembling process ideally extracts information from a widget and then destroys it.

Widgets can however be stored (in memory), if making a widget is so tedious compared to storing them. So, you can also reuse a widget and make a new one only when you run *out of stock*. The process of dissembling a widget is called *clearing*. A widget produced by a provider is known as child. In fact, the word *widget* is rarely used and instead the term *child* is used.

3.2 Aduct.Element



Element is an container that can hold the child produced by a provider. They are the front-end in an application, so any sort of interaction happens via elements. An element has an action button. It is used to alter the child in the element, like changing, removing, clearing a child. Action buttons are recommended, but can be avoided if you have any other approach. On the side of an action button, you can also add widgets like quick tool buttons. Ideally you can attach only one widget, so if you want to add multiple widgets, put them inside a container like grid or box. Elements can not be directly attached to an interface, they need a container called views.

3.3 Aduct.View

View is a container of widgets. Aduct comes with three basic views, that are enough for most of the use cases. New views can also be made easily if they don't satisfy your need. The three views are :

3.3.1 Aduct.Bin

A bin can contain only one child. The child can either be another view or element.

3.3.2 Aduct.Paned

A paned is a container that can hold two children, either in vertical or horizontal direction. Similar to a bin, paned can hold either a view or an element. Paned contains a movable handle between its children, which can change the *space* allocated to each child.

3.3.3 Aduct.Notebook

A notebook can contain an arbitrary number of children, but they all should be elements. In a notebook, only one child is visible at a time and the visible child can be changed using the tabs located on an edge. Notebook also has action buttons, attached at either side of tabs or at one side. They are optional, so can be avoided if not needed.

3.4 Framing an Application

Now we describe how to create a convenient interface using Aduct.

- Make all the basic things required to make the application, like collecting plugins, user data.
- From your plugins and own collection, make a list of providers which can produce child widgets.
- Design an interface that you and a lot of users find convenient. The interface designed should be using views and elements, with children from providers.
- Then connect the elements, views with *tweak* functions. Tweak functions are those that can modify properties of views and elements.
- Provide a way for the users to save and load interfaces using Aduct's built-in functions.

The above points do not explain how they are done practically, so let's get our hands wet in a short tutorial in next chapter.

CHAPTER 4

Making an Application

No more theory, let us now get into the business of making applications. In this tutorial we will make a very basic application that helps you in understanding the logics. We have divided the tutorial into simpler parts so it is easy to follow.

Since the application should be easy, we will handle only a very few widgets of `Gtk`. The details of the widgets that are used in our providers are given below :

- `Gtk.Entry`

  An entry widget takes single line input from user. It can also be used to display text that can only be copied but not modified. When the text in an entry is changed, it emits *changed* signal. To prevent editing the text of an entry, we set its *editable* property to *False*.

- `Gtk.FileChooserButton`

  To open, save, select files or folders we need a file chooser widget. After user selects a file in the file chooser dialog, a *file-set* signal is emitted.

- `Gtk.Grid`

  A grid allows packing widgets into a single widget in a tabular format. To attach a widget to grid, we use the gird's `Gtk.Grid.attach()` method.

- `Gtk.Label`

  Text can be displayed using a label widget. In our demo application, we are using only basic features of a label, that is *just display text*.

- `Gtk.ModelButton`

  Menu is often a linear list of textual buttons. To break that rule and include other widgets in menu, we use model buttons.

- `Gtk.Popover`

  Popovers are used to display something for a short period of time. They are usually used for drop-down menus.

- `Gtk.ScrolledWindow`

  If a widget is too large to be accommodated in given space, we use a scrolled window. This shows only the portion of the widget that could be displayed and rest can be scrolled.

- `Gtk.TextBuffer`

Text buffer is used to store the text for a text view widget. A single text buffer can be shared across multiple text views.

- `Gtk.TextView`

  To enable multi-line text compatibility a text view widget is used. Text view is a front-end and the text in it is controlled using a text buffer. As in the case of label, we are not going to use the full power of a text view.

- `Gtk.ToggleButton`

  A toggle button is like a switch, it can have two states *active* and *inactive*. A *toggled* signal is emitted if the state of the button is changed.

- `Gtk.Window`

  A window is the top-level widget that represents your application. We quit Gtk's main loop when the main window is destroyed.

So let's get started.

4.1 Making Providers

Providers are the core part of our application. Because of the design philosophy of Aduct, every part of an application behaves like a plugin. This makes an application modular in every way. To keep things simple, we make only three providers.

- *Provider A*

  It gives a text entry and a toggle button. The toggle button allows editing the entry.

- *Provider B*

  It gives a text view and a file chooser button. The file chooser button opens the file for displaying.

- *Provider C*

  It gives a label with text *Hello World*.

Note: If the code to make the first two providers are difficult, then copy the code for Provider C and change the text for label, but the results will vary.

To make providers, we inherit *`Aduct.Provider`*. Then we add the required methods (please refer *`Aduct.Provider`* for more details on required methods).

In Aduct, you will often see variables named `child_dict`. A `child_dict` is of the following format.

```
child_dict = {
    "child": Gtk.Widget,
    "child_name": str,
    "icon": Gtk.Image,
    "header_child": Gtk.Widget or None,
    "provider": Aduct.Provider
}
```

The source code for our providers are as follows. If you are lazy to copy-paste, `download` it.

```
import gi
gi.require_version("Gtk", "3.0")
from gi.repository import Gtk, GObject

import Aduct
```

(continues on next page)

(continued from previous page)

```python
class Provider_A(Aduct.Provider):

    name = GObject.Property(type=str, default="Provider A", flags=GObject.ParamFlags.
↪READABLE)

    def __init__(self):

        Aduct.Provider.__init__(self)
        self.text = ""
        self.toggles = []
        self.entries = []
        self.editable = False

    def change_text(self, entry):

        self.text = entry.get_text()
        for entry in self.entries:
            entry.set_text(self.text)

    def clear_child(self, child_props):

        self.entries.remove(child_props["child"])
        self.toggles.remove(child_props["header_child"])
        del child_props

    def get_a_child(self, child_name):

        entry = Gtk.Entry(margin=5, text=self.text, editable=self.editable)
        entry.connect("changed", self.change_text)

        icon = Gtk.Image.new_from_icon_name("terminal", 2)
        # Choose whatever icon you want

        switch = Gtk.ToggleButton(
            label="Allow Edit", hexpand=True, halign=2, active=self.editable
        )
        switch.connect("toggled", self.toggle_editable)

        self.entries.append(entry)
        self.toggles.append(switch)

        child_props = {
            "child_name": "Entry",
            "child": entry,
            "icon": icon,
            "header_child": switch,
        }
        return child_props

    def get_child_props(self, child_name, child, header_child):

        props = {"child_name": child_name, "text": self.text, "editable": self.
↪editable}
        return props
```

(continues on next page)

(continued from previous page)

```
    def get_child_from_props(self, props):

        self.editable = props["editable"]
        self.text = props["text"]

        for toggle in self.toggles:
            toggle.set_active(self.editable)
        for entry in self.entries:
            entry.set_editable(self.editable)
            entry.set_text(self.text)

        return self.get_a_child(props["child_name"])

    def toggle_editable(self, toggle):

        self.editable = toggle.get_active()
        for toggle in self.toggles:
            toggle.set_active(self.editable)
        for entry in self.entries:
            entry.set_editable(self.editable)


class Provider_B(Aduct.Provider):

    name = GObject.Property(type=str, default="Provider B", flags=GObject.ParamFlags.
↪READABLE)

    def __init__(self):

        Aduct.Provider.__init__(self)
        self.file_choosers = []
        self.buffer = Gtk.TextBuffer()
        self.path = None

    def clear_child(self, child_props):

        self.file_choosers.remove(child_props["header_child"])
        del child_props

    def change_text_at_buffer(self, fp_but):

        path = fp_but.get_filename()
        self.path = path
        fp = open(path)
        text = fp.read()
        self.buffer.set_text(text)
        fp.close()

        for fp_chooser in self.file_choosers:
            fp_chooser.set_filename(self.path)

    def get_a_child(self, child_name):

        textview = Gtk.TextView(margin=5, buffer=self.buffer)
        scrolled = Gtk.ScrolledWindow(expand=True)
        scrolled.add(textview)
```

(continues on next page)

(continued from previous page)

```
        icon = Gtk.Image.new_from_icon_name("folder", 2)

        fp_but = Gtk.FileChooserButton(title="Choose file", hexpand=True, halign=2)

        if self.path:
            fp_but.set_filename(self.path)

        fp_but.connect("file-set", self.change_text_at_buffer)
        self.file_choosers.append(fp_but)

        child_props = {
            "child_name": "TextView",
            "child": scrolled,
            "icon": icon,
            "header_child": fp_but,
        }
        return child_props

    def get_child_props(self, child_name, child, header_child):

        props = {"child_name": child_name, "path": self.path}
        return props

    def get_child_from_props(self, props):

        self.path = props["path"]
        if self.path:
            fp = open(self.path)
            text = fp.read()
            self.buffer.set_text(text)
            fp.close()

            for fp_chooser in self.file_choosers:
                fp_chooser.set_filename(self.path)

        return self.get_a_child(props["child_name"])


class Provider_C(Aduct.Provider):

    name = GObject.Property(type=str, default="Provider C", flags=GObject.ParamFlags.
↪READABLE)

    def __init__(self):

        Aduct.Provider.__init__(self)

    def clear_child(self, child_props):

        del child_props

    def get_a_child(self, child_name):

        label = Gtk.Label(margin=5, label="Hello world")

        icon = Gtk.Image.new_from_icon_name("glade", 2)
        child_props = {
```

(continues on next page)

(continued from previous page)

```
            "child_name": "Label",
            "child": label,
            "icon": icon,
            "header_child": None,
        }
        return child_props

    def get_child_props(self, child_name, child, header_child):

        props = {"child_name": child_name}
        return props

    def get_child_from_props(self, props):

        return self.get_a_child(props["child_name"])


A = Provider_A()
B = Provider_B()
C = Provider_C()
```

We prefer keeping the above code in a separate file (could be named *providers.py*), because in practical situations (while making real applications) it is better to isolate providers from the core application, this makes it easy to maintain. The reason we imported Gtk from Aduct is not so crucial. It was done to reduce typing, also it makes sure that we are using the same version of Gtk that Aduct is using.

4.2 Designing the Application

Now we have made providers, our next step is to frame the application. Open a new file (could be named *app.py*). To allow widgets in the application, we should put a way for users to view available widgets and select the required. This is done using action button of element and notebook. The convention is when users left-click an action button, it should show widgets (from providers) and when they right-click, it should show options to modify the interface.

There are two suitable ways to show the users the widgets to select from. It could be a popup window. But pop-ups are considered distracting. The second option is drop-down menu (also known as popover menu). Popovers are better as they cover only a small area and are not as annoying as popup windows. We populate the menu with model buttons.

Before adding providers, we should also spend time in a kind of functions known as *creator functions*. As Aduct is an interface to dynamically modify an interface, you need to be able to dynamically make Aduct widgets. So we make small functions that, when called give the required widget. An advantage of such functions is that they can be used to add custom changes to widgets like changing border spacing, connecting signals and automate other repeating tasks. The first few lines of *app.py* is given below. (The complete file is also available for `download`.)

```
import Aduct
from Aduct import Gtk

from providers import A, B, C  # providers from provider.py

last_widget = None  # The last widget (element/notebook) where popover was shown.


def new_element():
    element = Aduct.Element(margin=5)
    element.connect("action-clicked", show_popover_element)
```

(continues on next page)

(continued from previous page)

```
    # show_popover_element is a function to show the popover for an element.
    return element


def new_bin():
    bin_ = Aduct.Bin()
    return bin_


def new_paned(orientation=0):
    paned = Aduct.Paned(orientation=orientation)
    return paned


def new_notebook():
    notebook = Aduct.Notebook()
    button = Gtk.Button()
    icon = Gtk.Image.new_from_icon_name("list-add", 2)
    button.add(icon)
    notebook.set_action_button(button, 1)
    notebook.connect("action-clicked", show_popover_notebook)
    # show_popover_notebook is a function like show_popover_element.
    return notebook
```

The idea of the above code is simple. When action button of an element or notebook is clicked, it emits a signal and popover is shown in return. The popover contains model buttons for various purposes, when they are clicked, they need to know *for which* element or notebook they were clicked. To tackle this, when an action button is clicked, we correspondingly set the value of `last_widget` to that widget. With that, let's append the next lines of code.

```
def show_popover_element(ele, but, event):

    global last_widget
    last_widget = ele

    if event == 1:  # 1 -> left-click of mouse
        prov_popover.set_relative_to(but)
        prov_popover.popup()

    elif event == 3:  # 3 -> right-click of mouse
        for modbs in tweaks.values():
            for modb in modbs:
                modb.set_sensitive(True)
        tweak_popover.set_relative_to(but)
        tweak_popover.popup()


def show_popover_notebook(nb, but, event):

    global last_widget
    last_widget = nb

    if event == 1:
        prov_popover.set_relative_to(but)
        prov_popover.popup()

    elif event == 3:
```

(continues on next page)

(continued from previous page)

```
        for modb in tweaks["Element"]:
            modb.set_sensitive(False)
        for modb in tweaks["Notebook"]:
            modb.set_sensitive(False)
        tweak_popover.set_relative_to(but)
        tweak_popover.popup()
```

Both the above functions are same but the difference between them is that first one is for an element and second is for a notebook. `prov_popover` is for displaying widgets from providers and `tweak_popover` is for showing options to modify the interface. As per the convention mentioned earlier, we show `prov_popover` when users left-click and `tweak_popover` when users right-click an action button. We will cover later why we are changing sensitivities of model buttons.

Now let us make some more functions that can modify the interface.

```
def remove_element(wid):
    global last_widget
    Aduct.remove_element(last_widget, last_widget.get_parent())


def add_to_paned(wid, position):
    global last_widget
    element = new_element()
    paned = new_paned()
    if position == 0:
        paned.set_orientation(0)
        Aduct.add_to_paned(last_widget, element, paned, 1)
    elif position == 1:
        paned.set_orientation(0)
        Aduct.add_to_paned(last_widget, element, paned, 2)
    elif position == 2:
        paned.set_orientation(1)
        Aduct.add_to_paned(last_widget, element, paned, 1)
    elif position == 3:
        paned.set_orientation(1)
        Aduct.add_to_paned(last_widget, element, paned, 2)


def add_to_notebook(wid, position):
    global last_widget
    notebook = new_notebook()
    notebook.set_tab_pos(position)
    Aduct.add_to_notebook(last_widget, notebook)
```

Please read *Functions* to know the details of the functions used from Aduct. Next we add more functions for changing child at an element, saving and loading interfaces.

```
def change_child_at_element(wid, prov, child_name):
    global last_widget
    if last_widget.get_type() == "element":
        Aduct.change_child_at_element(last_widget, prov, child_name)
    elif last_widget.get_type() == "notebook":
        element = new_element()
        Aduct.change_child_at_element(element, prov, child_name)
        Aduct.add_to_notebook(element, last_widget)
        element.show_all()
```

(continues on next page)

(continued from previous page)

```python
def save_interface(wid):
    from json import dump

    with open("aduct.ui", "w") as fp:
        ui_dict = Aduct.get_interface(top_level)
        dump(ui_dict, fp, indent=2)


def load_interface(wid):
    from json import load

    with open("aduct.ui") as fp:
        ui_dict = load(fp)
        creator_maps = {
            "type": {
                "element": (new_element, (), {}),
                "bin": (new_bin, (), {}),
                "notebook": (new_notebook, (), {}),
                "paned": (new_paned, (), {}),
            }
        }
        init_maps = {
            "provider": {"Provider A": A, "Provider B": B, "Provider C": C, None:␣
↪None}
        }
        Aduct.set_interface(ui_dict, top_level, creator_maps, init_maps)
```

The first function does some straight-forward tasks. It changes a child at element when called from element. In case it is called from a notebook, we make a new element, get a child from provider and add it to the element. Then we append the element to the notebook.

The second function gets the interface; it is a dictionary with strings, numbers and None. So it can be dumped using *json* in human-readable format. We are using a file named *aduct.ui* for saving and loading interfaces. `top_level` (declared later) is the view or element from which the interface should be fetched. It is usually the root widget.

The third function surely deserves a mention. After completing this tutorial, you can run the script (*app.py*), try playing with the interface. Then save the interface. After that open the file named *aduct.ui*, you will see a *JSON-styled* data with keys like *type*, *provider* etc. Now when you ask Aduct to create the interface from the same file, it replaces all the required values with objects (or widgets here). The convention is that key *type* states the type of Aduct widget and *provider* states the name of provider.

The dictionaries whose name ends with *maps*, does the job of replacing strings or numbers with an object. They are nested-dictionaries of depth two. It is like *what key to replace? If found replace the value of that key with the value from maps.* For example, from `init_maps` we have the key *provider.* So first the `set_interface` function will look for any key named *provider* in `ui_dict`. If found it will look at its value, say it is *Provider A,* now it will go back to `init_maps` and look for the value of key *Provider A* in the dictionary which is the value of key named *provider*. From the above it is provider `A`, then the function replaces the value *Provider A* in `ui_dict` with the actual object; provider `A`. So you can consider it as a mapping of strings to objects.

The purpose of `creator_maps` and `init_maps` are pretty same. Their difference lies in values they are replacing. `init_maps` maps to object already created, that is initialized objects, like providers, plugins, file objects. It is to replace object that are already available and should not created again. On the other hand, `creator_maps`, dynamically creates objects as needed. It is for the purpose where each object has to be unique or can be created multiple times for multiple usage. The values of `creator_maps` are of this order `(function, args, kwargs)`. While replacing strings with objects, the object is created by calling the function like this : `object = function(*args,` `**kwargs)`.

Pretty simple as that, however if you are confused, remember them as mappings. That's it.

The third function needs a root widget. It will remove whatever child it holds and replaces it with the children from the *JSON* file. However it returns the old child for recovering data, something not so necessary in our application.

The finishing parts of our application is just connecting everything, creating a new window and adding a top level view.

```
provs = [
    (
        A,
        Gtk.ModelButton(text="Entry"),
        "Entry",
    ),  # Making a model-button for each provider.
    (B, Gtk.ModelButton(text="TextView"), "TextView"),
    (C, Gtk.ModelButton(text="Label"), "Label"),
]

prov_grid = Gtk.Grid()  # A grid to store them

for y, (prov, modb, child_name) in enumerate(provs):
    prov_grid.attach(modb, 0, y, 1, 1)
    modb.connect("clicked", change_child_at_element, prov, child_name)

prov_popover = Gtk.PopoverMenu()
prov_popover.add(prov_grid)
prov_grid.show_all()

# Pretty same as providers, but for tweak functions.
tweaks = {
    "Element": (Gtk.ModelButton(text="Remove"),),
    "Notebook": (
        Gtk.ModelButton(text="Add to top notebook"),
        Gtk.ModelButton(text="Add to side notebook"),
    ),
    "Paned": (
        Gtk.ModelButton(text="Split left"),
        Gtk.ModelButton(text="Split right"),
        Gtk.ModelButton(text="Split up"),
        Gtk.ModelButton(text="Split down"),
    ),
    "Interface": (
        Gtk.ModelButton(text="Load interface"),
        Gtk.ModelButton(text="Save interface"),
    ),
}

tweak_grid = Gtk.Grid()

for x, title in enumerate(tweaks):
    label = Gtk.Label(label=title)
    tweak_grid.attach(label, x, 0, 1, 1)
    modbs = tweaks[title]
    for y, modb in enumerate(modbs):
        tweak_grid.attach(modb, x, y + 1, 1, 1)

tweak_popover = Gtk.PopoverMenu()
```

(continues on next page)

(continued from previous page)

```
tweak_popover.add(tweak_grid)
tweak_grid.show_all()


def connect_tweaks():
    # Connecting model-buttons to required functions.
    elem_modb = tweaks["Element"][0]
    elem_modb.connect("clicked", remove_element)

    top_nb_modb = tweaks["Notebook"][0]
    top_nb_modb.connect("clicked", add_to_notebook, 2)
    side_nb_modb = tweaks["Notebook"][1]
    side_nb_modb.connect("clicked", add_to_notebook, 0)

    l_paned_modb = tweaks["Paned"][0]
    r_paned_modb = tweaks["Paned"][1]
    u_paned_modb = tweaks["Paned"][2]
    d_paned_modb = tweaks["Paned"][3]

    # 0, 1, 2, 3 are integer values of Gtk.PositionType.
    l_paned_modb.connect("clicked", add_to_paned, 0)
    r_paned_modb.connect("clicked", add_to_paned, 1)
    u_paned_modb.connect("clicked", add_to_paned, 2)
    d_paned_modb.connect("clicked", add_to_paned, 3)

    load_modb = tweaks["Interface"][0]
    save_modb = tweaks["Interface"][1]
    load_modb.connect("clicked", load_interface)
    save_modb.connect("clicked", save_interface)


connect_tweaks()

top_level = new_bin()
element = new_element()
top_level.add_child(element)  # Making a single element and adding it.

win = Gtk.Window(default_height=500, default_width=750)
win.add(top_level)
win.connect("destroy", Gtk.main_quit)
win.show_all()
Gtk.main()
```

Phew… we completed making the application! You might not have understood some parts, but still, run the application (run *app.py*) and see how it looks. Well just an empty screen with an empty button, right? Click on the action button of element and add new child to the element. Next try right clicking the action button to split it or add it to a notebook. Have fun removing the views and adding new one. When you are comfortable with the interface, save the interface and close the application. Now open it again and load the interface. You should see the interface you saved.

Let us discuss something we promised earlier. Run the application and add an element to notebook. Now right-click the element's action button and click on *Add to side notebook*, you should see an error in your terminal or console that a Aduct notebook can only have a Aduct element as child. It is not a bug, it is a feature! Aduct notebook can only attach a named child and the only named child in Aduct is element. So you will get error when you try to add a child of irrelevant type to a notebook. This is the reason we changed the sensitivities of a few menu items. Because we don't want to allow users to do something not permitted. We could have made separate popovers for notebook and element, but to avoid repeating codes with small difference, we omitted it. You might wonder why we didn't change sensitivities of menu items of popovers shown for elements inside a notebook, the answer is we want you to try!

Note a few more things which might look absurd. When you remove an element from a paned, it collapses to a bin. When you remove an element from notebook with three or more pages, nothing goes wrong. But when you remove an element from a notebook with only two pages, the notebook drops to a bin. In case of a bin, when you try to remove its child element, instead of removing, it only clears the element.

This also has some reasons behind it. A paned is meant to hold two child, so when you remove its one child, the purpose of a paned is destroyed, so it becomes a bin. Similarly, a notebook is meant to hold a number of elements and show only one of them at a time. A notebook with only page is against its purpose, so it becomes a bin. For bin, the same logic is applied. A Aduct bin is, by convention, used as a top-level for holding other view. When you remove the element from it, the complete interface link is broken and you get a blank space, where no kind of interaction is possible. To avoid this, bin always clears the element instead of removing it.

However, if these behaviors are not acceptable to you, you are always free to create your own functions and use them.

While using the widgets like entry, text view in our application, you might have noticed that an entry is just a copy of another entry, with same text and mode, synchronized between them. This is to symbolize that widgets with same name are basically the same. But this is not enforced. It depends on the provider, it may produce a new widget or just a copy.

So here we reach the end of tutorial. There are some lines, paragraphs or entire section that doesn't even make any sense to you. Feel free to discuss it with other developers to get help. Also if you think, the same matter could be presented in a better manner, you are always welcome to suggest your edits.

At last, we would like to say, designing an application is like painting. Everyone has a brush and seven basic colors. It depends on the painter how great he/she is going to make his/her art look. He/she may have a different ideology and style, its unique and can't be duplicated.

Same in case of an application, Aduct is like brush and paint, it lies in your method, how well you are going to utilize it. Sometimes it could come out worse, where you should surely retry. Sometimes it could come great, where you should share the method with others (including us!). Also beauty lies in the eyes of the viewer, not in the painting… cheers!

CHAPTER 5

API Reference

5.1 Classes

5.1.1 Element

Element represents an individual block. It can hold a widget and handle operations with it. While setting and removing, a `child_dict` named dictionary is taken and returned. The format of `child_dict` is as follows

```
child_dict = {
    "child": Gtk.Widget,
    "child_name": str,
    "icon": Gtk.Image,
    "header_child": Gtk.Widget or None,
    "provider": Aduct.Provider
}
```

Here only `header_child` key is optional.

class `Aduct.Element.`**`Element`**(*child_dict=None*, *use_action_button=True*, *pack_type=0*, ***kwargs*)

Makes an element based on given properties. Its CSS name is *aduct-element*.

Parameters

- **`child_dict`** (`dict`) – A dictionary object containing properties of child. Given a valid dictionary, the child is added to `self` while initializing. It is `None`, when not given.
- **`use_action_button`** (`bool`) – States whether to use an action button. Default is `True`.
- **`pack_type`** (`Gtk.PackType`) – Specfies the position of action button. It can be an integer of value either 0 or 1, which represents `Gtk.PackType.START` or `Gtk.PackType.END` respectively. The default value is `Gtk.PackType.START`.
- **`**kwargs`** – The values to be passed to `Gtk.Grid`, from which *`Element`* is derived.

`action_button`

Action button that is used to handle interactions with user. Its default name is *aduct-element-action_button*.

Type `Gtk.Button`

`child_name`
The name of child held by `self`.

Type `str`

`pack_type`
The position of action button in `self`.

Type `Gtk.PackType`

`provider`
The provider that produced the child of `self`.

Type *`Provider`*

`Signals`

action-clicked Emitted with an integer when action button of `self` is clicked. The integer is 1, 2, 3 for LMB, MMB, RMB respectively.

child-added Emitted when a child is added to `self`.

child-cleared Emitted when the child of `self` is cleared.

child-removed Emitted when the child of `self` is removed.

Note: Incase having `use_action_button` as `False`, an action button is still created, but is not attached to the `self`.

`clear_child``()`
Clears the child at `self`. After clearing, `child-cleared` signal is emitted.

`disable_action_button``()`
Removes the action button of `self`. Nothing is done if it is already disabled.

`enable_action_button``()`
Adds the action button of `self`. Nothing is done if it is already enabled.

`get_child``()`
Gets the child held by `self`.

Returns The child of `self` or `None` if `self` has no child.

Return type `Gtk.Widget` or `None`

`get_child_name``()`
Gets the name of child held by `self`.

Returns The child name of `self` or `None` if `self` has no child.

Return type `str` or `None`

`get_header_child``()`
Gets the child packed at the header of `self`.

Returns The header child of `self` or `None` if `self` has no header child.

Return type `Gtk.Widget` or `None`

`get_icon``()`
Gets the icon representing child held by `self`.

Returns The icon of *`action_button`* or `None` if `self` has no icon for child.

Return type `Gtk.Image` or `None`

get_props()
Gets the properties of child held by `self`.

Returns The dictionary that can be later used to build the same interface.

Return type `dict`

get_provider()
Gets the provider for child held by `self`.

Returns The provider of `self` or `None` if `self` has no child.

Return type *`Provider`* or `None`

get_type()
Gets the type of *self*.

Returns The name of provider.

Return type `str`

remove_child()
Removes the child held by `self`.

By removing a child, all its associated properties like icon, header child are also removed. A `child-removed` signal is emitted by `self` after removal.

Raises `ValueError` – Raised when `self` has no child.

Returns A dictionary with child properties.

Return type `dict`

set_child(*child_dict*)
Sets the child in `self` from given properties.

If `self` already has a child, then its cleared before adding this new child. A `child-added` signal is emitted after addition.

Parameters **child_dict** (`dict`) – A valid dictionary with properties of child.

set_child_name(*child_name*)
Sets the name of child held by `self`.

Parameters **child_name** (`str`) – The new name of child.

set_from_props(*props*)
Sets the interface of `self` from given properties.

If `self` already has a child, then its cleared before adding this new child.

Parameters **props** (`dict`) – The dictionary from which properties are set.

set_header_child(*header_child*)
Sets the header child of `self`.

Parameters **header_child** (`Gtk.Widget`) – The new header child of `self`.

set_icon(*icon*)
Sets the icon of child held by `self`.

Parameters **icon** (`Gtk.Image`) – The new icon of child.

set_provider(*provider*)
Sets the provider of child held by `self`.

Parameters **provider** (*Provider*) – The new provider of child.

type
Used by autodoc_mock_imports.

5.1.2 Views

Bin

Bin is a view that can hold only one child. The child can be either an *View* or *Element*.

class Aduct.Views.Bin.**Bin**(***kwargs*)
Makes a bin based on given properties. Its CSS name is *aduct-bin*.

Parameters ****kwargs** – The keyword arguments to be passed to `Gtk.Bin` from which *Bin* is made.

add_child(*child*)
Adds the child `self`.

Parameters **child** (*View* or *Element*) – The child to be added to `self`.

Raises `ValueError` – Raised when `self` already has a child.

get_props()
Gets the interface properties.

Returns A dictionary with interface properties.

Return type `dict`

remove_child(*child*)
Removes the given child.

Parameters **child** (*View* or *Element*) – The child which has to be removed from `self`.

Raises `ValueError` – Raised when `child` is not present in `self`.

replace_child(*old_child*, *new_child*)
Replaces the existing child with a new child.

Parameters

- **old_child** (*View* or *Element*) – The child present in `self` which has to be replaced.
- **new_child** (*View* or *Element*) – The child that will replace the given `old_child` of `self`.

set_from_props(*props*)
Sets the interface from given properties.

Parameters **props** (`dict`) – The dictionary containig properties of interface.

type
Used by autodoc_mock_imports.

Notebook

Notebook is a view that can hold only children of type *Element*. With this restriction, there is no limitation in number of children it can hold.

class Aduct.Views.Notebook.**Notebook**(***kwargs*)

Makes a notebook based on given properties. Its CSS name is *aduct-notebook*.

Parameters

- ****kwargs** – The values to be passed to Gtk.Notebook from which *Notebook* is made.
- **Signals** –

action-clicked Emitted with an integer when action button of `self` is clicked. The integer is 1, 2, 3 for LMB, MMB, RMB respectively.

add_child(*child*, *position=-1*)

Adds the child `self`.

Parameters **child** (*Element*) – The child to be added to `self`.

Raises TypeError – Raised when `child` is not a *Element*.

change_child_label(*child*)

Changes the tab label for existing child.

The text for new label is taken as the name of child of `child` (`Aduct.Element.child_name`). It is set to *No child* when *child* has no name for its child (`Aduct.Element.child_name is None`).

Parameters **child** (*Element*) – The child whose tab label has to be changed.

get_action_button(*pack_type*)

Gets the action button at given position.

Parameters **pack_type** (Gtk.PackType) – The position from which action button has to retrieved.

Returns The action button of `self` or None if `self` has no action button.

Return type Gtk.Button or None

get_number_of_action_buttons()

Gets the number of action buttons present.

Returns The number of action buttons.

Return type int

get_props()

Gets the interface properties.

Returns A dictionary with interface properties.

Return type dict

get_tab(*child*)

Gets a tab label for child.

The text for new label is taken as the name of child of `child` (`Aduct.Element.child_name`). It is set to *No child* when *child* has no name for its child (`Aduct.Element.child_name is None`). Based on position of tabs in `self`, the orientation of text in the label also varies.

Parameters **child** (*Element*) – The child which requires a tab label.

Returns Label with text determined from `child`.

Return type Gtk.Label

remove_child(*child*)

Removes the given child from `self`.

Parameters **`child`** (*`Element`*) – The child which has to be removed from `self`.

Raises `ValueError` – Raised when `child` is not present in `self`.

`replace_child`(*old_child*, *new_child*)
Replaces the existing child with a new child.

Parameters

- **`old_child`** (*`Element`*) – The child present in `self` which has to be replaced.
- **`new_child`** (*`Element`*) – The child that will replace the given existing child of `self`.

Raises `TypeError` – Raised when `new_child` is not a *`Element`*.

`set_action_button`(*action_button*, *pack_type*)
Sets the action button to notebook.

Parameters

- **`action_button`** (`Gtk.Button`) – The button to be added to notebook. It need not be a `Gtk.Button` actually, it could be any widget that can handle *button-press-event*.
- **`pack_type`** (`Gtk.PackType`) – The position of the action button.

`set_from_props`(*props*)
Sets the interface from given properties.

Parameters **`props`** (`dict`) – The dictionary containig properties of interface.

Raises `ValueError` – Raised when there is a mismatch of number of action buttons in properties and `self`.

`type`
Used by autodoc_mock_imports.

Paned

Paned is a view that can hold two children. The two children can either be *`View`* or *`Element`*.

`class` `Aduct.Views.Paned.`**`Paned`**(***kwargs*)
Makes a paned based on given properties. Its default name is *aduct-paned*

Parameters **`**kwargs`** – The keyword arguments to be passed to `Gtk.Paned` from which *`Paned`* is made.

`add_child`(*child*, *position=0*)
Adds the child to paned.

When `position` is 1 or 2, `child` is added at panel 1 or 2 of `self` respectively. When it is 0, `child` is added to the first available panel. When it is neither of specified values, nothing is done.

Parameters

- **`child`** (*`View`* or *`Element`*) – The child to be added to `self`.
- **`position`** (`int`) – The panel at which the child has to be added. `position` can be 0 or 1 or 2, with default value being 0.

Raises `ValueError` – Raised when `self` already has a child.

`get_props`()
Gets the interface properties.

Returns A dictionary with interface properties.

Return type `dict`

`remove_child`(*child*)

Removes the given child from `self`.

Parameters **`child`** (*`View`* or *`Element`*) – The child which has to be removed from `self`.

Raises `ValueError` – Raised when `child` is not present in `self`.

`replace_child`(*old_child*, *new_child*)

Replaces the existing child with a new child.

Parameters

- **`old_child`** (*`View`* or *`Element`*) – The child present in `self` which has to be replaced.
- **`new_child`** (*`View`* or *`Element`*) – The child that will replace the given existing child of `self`.

Raises `ValueError` – Raised when `old_child` is not in `self`.

`set_from_props`(*props*)

Sets the interface from given properties.

Parameters **`props`** (`dict`) – The dictionary containig properties of interface.

`type`

Used by autodoc_mock_imports.

View

View can hold children of type *`Element`*. In some case, there is a restriction on number of children it can hold.

`class` `Aduct.Views.View.`**`View`**(***kwargs*)

This an abstract class, that gives an idea of methods a *`View`* must have. Unless otherwise stated, all the description of methods are generalised expected behavior of *`View`*. Depending upon the nature of view, the type of child it can hold also varies.

`add_child`(*child*)

Adds the child `self`.

Parameters **`child`** (*`View`* or *`Element`*) – The child to be added to `self`.

Raises

- `ValueError` – Raised when there is insufficient information to add `child` to `self`.
- `TypeError` – Raised when `child` is of invalid type.

Note: When there is a lack of information to add `child`, `self` may try its best to add `child` in suitable position.

`get_props`()

Gets the interface properties.

Returns A dictionary with interface properties.

Return type `dict`

`get_type`()

Gets the interface properties.

Returns A dictionary with interface properties.

Return type `dict`

`remove_child`(*child*)
Removes the given child.

Parameters **`child`** (`View` or `Element`) – The child which has to be removed from `self`.

Raises `ValueError` – Raises when `child` is not present in `self`.

`replace_child`(*old_child*, *new_child*)
Replaces the existing child with new child.

Parameters

- **`old_child`** (`View` or `Element`) – The child present in `self` which has to be replaced.
- **`new_child`** (`View` or `Element`) – The child that will replace *old_child* of `self`.

`set_from_props`(*props*)
Sets the interface from given properties.

Parameters **`props`** (`dict`) – The dictionary containig properties of interface.

5.1.3 Provider

Provider acts as a producer of widgets that are placed as child in `Element`.

class `Aduct.Provider.`**`Provider`**(**args*, ***kwargs*)
This a template that gives an idea of methods a `Provider` must have. Unless otherwise stated, all the description of methods are generalised expected behavior of a `Provider`.

`clear_child`(*child_dict*)
Clears the given child.

Parameters **`child_dict`** (`dict`) – A dictionary with properties of the child.

`get_a_child`(*child_name*)
Gets a child with given name.

Parameters **`child_name`** (`str`) – The name of child to be retrieved.

Returns A dictionary with properties of child.

Return type `dict`

`get_child_from_props`(*props*)
Gets a child based on given interface properties.

Parameters **`props`** (`dict`) – The interface properties for child.

Returns A dictionary with properties of child.

Return type `dict`

`get_child_props`(*child_name*, *child*, *header_child*)
Gets the interface properties from given values.

Parameters

- **`child_name`** (`str`) – The name of child.
- **`child`** (`Gtk.Widget`) – The child produced by `self`.
- **`header_child`** (`Gtk.Widget`) – The header child produced by `self`.

Returns A dictionary with interface properties of child.

Return type `dict`

`get_name()`

Gets the name of the provider.

Returns The name of provider.

Return type `str`

5.2 Functions

`Aduct.add_to_notebook`(*element*, *notebook*, *position=-1*)

Adds an element to the notebook at given position.

When the given `element` is already a child of some container, the function removes it from the parent and adds the `notebook` to parent. Then the orphan `element` is added to `notebook` at `position`.

Parameters

- **element** (*Element*) – The element to be added to *notebook*.
- **notebook** (*Notebook*) – The notebook to which `element` has to be added
- **position** (`int`) – The position at which `element` has to be inserted. When not provided, it takes up value of -1, which inserts the element as last page.

Raises `TypeError` – When given `element` is not a *Element*.

`Aduct.add_to_paned`(*child1*, *child2*, *paned*, *position*)

Adds the children to given paned determined by position.

The main child `child1` is added at first panel if `position` is 1 or second panel if `position` is 2. With respect to position of `child1`, `child2` is added at the complement panel. When `position` is neither 1 nor 2, nothing is done. If `child1` is already a child of parent, it is removed from the parent and `paned` is added back in its position. Then, the orphans `child1` and `child2` are added at requred places.

Parameters

- **child1** (`Gtk.Widget`) – The main child which has to be added at given `position`.
- **child2** (`Gtk.Widget`) – The other child which has to be added at the complement of given `position`. It has to be an orphan.
- **paned** (*Paned*) – The paned to which the children has to be added.
- **position** (`int`) – An integer value that is either 1 or 2. The complement of 1 is 2 and vice-versa.

`Aduct.add_to_view`(*child*, *view*)

Adds a child to the view.

If `child` is not an orphan, it is removed from its parent and `view` is added back in its place. Then `child` is added to `view`.

Parameters

- **child** (`Gtk.Widget` or Aduct.Element.Element) – The child that has to be added to `view`. It has to be an orphan.
- **view** (*View*) – The view to which `child` has to be added.

> **Warning:** This function is given as a fall-back case and should never be used blindly without knowing the properties of `child` and `view`. When the `view` already has a child or requires more information about adding it, exceptions are raised. Still, the `view` may try its best to add the `child` at the possible place, only when it can.
>
> Incase of `view` being a *`Notebook`*, it appends the `child` to the last position. But it requires `child` to be a *`Element`*. So at either case, you still have limitations that may end up in a weird result. For the same reasons, `child` has to be an orphan.

`Aduct.`**`change_child_at_element`**(*element*, *provider*, *child_name*)

Changes the child at given element with a child of given name provided by provider.

The previous child at `element` is cleared, after which the new child is added.

Parameters

- **`element`** (*`Element`*) – The element whose child has to be changed.
- **`provider`** (*`Provider`*) – The provider that acts as source of child.
- **`child_name`** (`str`) – The name of child to be added to `element`.

`Aduct.`**`get_interface`**(*top_level*)

Gets the interface starting from the given top level.

Parameters **`top_level`** (*`View`*) – A view which acts as the root widget.

Returns A dictionary with properties to build interface.

Return type `dict`

`Aduct.`**`remove_element`**(*element*, *view*)

Removes the given element from view.

When `view` is a *`Bin`*, it clears `element`. For other types of views, it removes `element`. Then, if the number of children in `view` is one, it replaces `view` with the other child of `view`.

Parameters

- **`element`** (*`Element`*) – The element to be removed.
- **`view`** (*`View`*) – The view from which `element` has to be removed.

`Aduct.`**`replace_child`**(*view*, *child1*, *child2*)

Replaces the given child of a view with another child.

Parameters

- **`view`** (*`View`*) – The view whose child has to be replaced.
- **`child1`** (`Gtk.Widget`) – The child of given view which has to be replaced.
- **`child2`** (`Gtk.Widget`) – The child which replaces *child1* in `view`.

Raises `TypeError` – Raised when the given `view` is not a *`View`*

`Aduct.`**`set_interface`**(*interface_dict*, *top_level*, *creator_maps*, *init_maps*)

Sets the interface starting from given the top level.

Parameters

- **`interface_dict`** (`dict`) – A dictionary that can be used to set interface.
- **`top_level`** (*`View`*) – The root widget from which the interface has to be set.

- **creator_maps** (dict) – A dictionary of format {key: (func, args, kwargs)}, that is used to create the required object. The object is then created using func(*args, **kwargs) and is substitued as value in interface_dict which has key key.
- **init_maps** (dict) – A dictionary of format {key: object} already initialized objects. The occurences of key in inerface_dict is then replaced with object.

Returns The widget that was previous child of top_level, None if top_level has no child.

Return type Gtk.Widget

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index