AXUI document Documentation

Release 0.1.5

xcgspring

Sep 30, 2017

Contents

Contents:

CHAPTER 1

AXUI introduce

Page Status Development

Last Reviewed

AXUI philosophy

Due to varies of reasons, writing and maintaining UI automation is easy to turn into a time cost/unpleasant task, and becomes hard to reach the predefined automation goal, and eventually UI automation is give up and leave people a bad impression

Let's have a summary of these reasons:

1. UI is hard for programming, UI is a good/direct interface for people, but not a good interface for programming. On the contrary, CLI is good for programming. controlling UI is basically a inter process communication, typical step is:

   (a) find the target UI with some UI special features

   (b) control process/scripts send a request to UI

   (c) check for response/state

   Potential problems in upper steps:

   - UI features used to find UI is not user friendly, some feature is not visible or have no meaning
   - UI control request is not user friendly, like a mouse/keyboard/touch event
   - UI could have lots of responses under different condition (UI change/no change/long time response/hang/crash), need to handle all of them

2. Tool support for UI automation is not easy to use, especially for PC platform due to there is a lot of UI frameworks on PC mainly problems for varies UI automation tools:

   - Different tool has different flavour of the programming language/style
   - Some tool is not powerful enough, but has no way to extend the function

- There are a lot of tools, need good experience to select a proper tool to use

---

Note: There are a lot of good tools emerging, like selenium for web test automation, appium for smartphone automation

---

3. Testers are lack of programming skills to make test scripts robust and easy to maintain. Testers are responsible for test design and test execution. Thus, testers might not have enough coding skills to use complex libraries. Automation tools should be easy for testers to use, let testers focus their energy to improve test cases

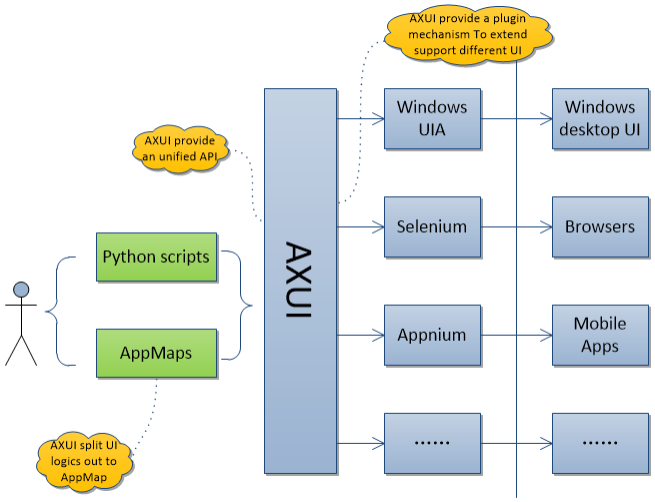
AXUI is a collection of solutions for these problems:

1. AXUI provide a plug-in mechanism for automation guy to extend support for different UI
2. AXUI provide an unified, easy to use python interface for use in test scripts
3. AXUI separate UI logic from test scripts, make test scripts more readable and easier to maintain
4. AXUI provide mechanism to handle auto met UI automation issues, like UI response time

To summarize, AXUI is to minimize the gap between testers and UI automation technologies.

AXUI structure

AXUI basically includes four modules, XML module to parse `app map`, driver module to manage different drivers, interface module to provide testers an easy to use API, core module provides basic functions used by other modules



AXUI technologies details

This section gives a brief introduce about some main features of AXUI

separate UI logic from test script

UI automation technologies often use a formatted string as identifier to find UI element, test scripts could contain a lot of these strings, makes scripts harder to understand and maintain. AXUI separate these strings to a standalone XML file, we usually create a XML file for one app, and thus we normally call this XML file an `app map` An `app map`

is just like a header file containing definitions, so scripts can reuse the definitions in `app map`, no need to string in scripts.

Here is a sample `app map` for windows media player:

```
<AXUI:app_map xmlns:AXUI="AXUI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"␣
↪xsi:schemaLocation="AXUI AXUI_app_map.xsd">

    <AXUI:funcs>
        <AXUI:func name="wmplayer_start" description="">
            <AXUI:step type="CLI" cmd='"C:\Program Files\Windows Media␣
↪Player\wmplayer.exe"'/>
        </AXUI:func>

        <AXUI:func name="wmplayer_stop" description="">
            <AXUI:step type="GUI" cmd='wmplayer_Window.WindowPattern.Close'/>
        </AXUI:func>

        <AXUI:func name="wmplayer_ctrl_o" description="">
            <AXUI:step type="GUI" cmd="wmplayer_Window.keyboard.Input '^o'" />
        </AXUI:func>

        <AXUI:func name="open_dialog_stop" description="">
            <AXUI:step type="GUI" cmd="wmplayer_Window.Open_Dialog.WindowPattern.Close
↪"/>
        </AXUI:func>
    </AXUI:funcs>

    <AXUI:UI_elements>
        <AXUI:Root_element name="desktop"/>

        <AXUI:UI_element name="wmplayer_Window" parent="desktop" identifier="Name=
↪'Windows Media Player' OR Name='Now Playing' AND LocalizedControlType='window'"␣
↪start_func="wmplayer_start" stop_func="wmplayer_stop">
            <AXUI:UI_element name="Open_Dialog" identifier="Name='Open' AND␣
↪LocalizedControlType='dialog'" start_func="wmplayer_ctrl_o" stop_func="open_dialog_
↪stop">
                <AXUI:UI_element name="FileName_ComboBox" identifier="Name='File name:
↪' AND LocalizedControlType='combo box'">
                    <AXUI:UI_element name="FileName_Edit" identifier="Name='File name:
↪' AND LocalizedControlType='edit'"/>
                </AXUI:UI_element>
                <AXUI:UI_element name="Open_Button" identifier="Name='Open' AND␣
↪LocalizedControlType='button' AND Index=2"/>
            </AXUI:UI_element>
        </AXUI:UI_element>
    </AXUI:UI_elements>

</AXUI:app_map>
```

plug-in mechanism to extend support for different UI

AXUI provide a plug-in mechanism to support extend other UI automation technologies to AXUI, See *Extend AXUI*

other supports for UI automation

AXUI provide other functions may used in UI automation,

- timeout mechanism to handle UI response time
- image compare for UI verification
- screenshot for every UI operation
- multiple languages support for internationalization (TODO, which means I will never do it :))

AXUI interface

AXUI restructure the original UI API into two parts, common operations like UI search is taken into AXUI internal, user should use `app map` to search UI element, UI element specified operations like button invoke, set editor value is ported out directly, user can use original API to operate the UI element after UI element is found, so AXUI can make code disciplined while not reducing the power of original API.

AXUI native API:

- for configuration, `AXUI.Config(config_file)`, also see *AXUI configurations*
- for UIElement checking, `AXUI.assertIsValid(element, msg)`, also see valid AXUI element
- for AppMap control, `AXUI.AppMap`, also see *AXUI AppMap*

Driver special API, mostly AXUI keep original API, for detail usage, you could refer original API documents:

- for windows, check windows native UIAutomation Client API
- for selenium, check selenium project
- for appnium, check appium project

a simple example

Let's have a simple demo about how to control wmplayer to playback some media file on one 32bit windows machine:

1. Environment prepare
   - First to use AXUI, you need install python environment, prefer python 2.7 (TODO, support python 3 :))
   - Install AXUI
   - Install `comtypes` as windows driver needs this library
2. Write a appmap, like upper example, also see *AXUI AppMap*
3. Write a config file, usually you can modify the default config file, also see *AXUI configurations*
4. Write a script like below:

```
############################################################
#prepare part
############################################################
import AXUI

config_file = "windows.cfg"             #config file name, abs path needed
app_map = "windows_media_player.xml"    #app map name, just need a file name,␣
↪path not needed
```

```
AXUI.Config(config_file)                #load the config file
appmap = AXUI.AppMap(app_map)           #load the appmap

############################################################
#action part, now you can do the UI actions, here we open a media file
############################################################
media_file = ""                         #your media file need to open

#set the media file path to wmplayer open dialog file name edit
appmap.wmplayer_Window.Open_Dialog.FileName_ComboBox.FileName_Edit.ValuePattern.
↪SetValue(media_file)

#press the open button
appmap.wmplayer_Window.Open_Dialog.Open_Button.InvokePattern.Invoke()
```

5. Run this script as below, and listen the music:

```
python script_name.py
```

CHAPTER 2

AXUI configurations

Page Status Development

Last Reviewed

AXUI configurations overview

AXUI support some custom configurations, to make AXUI suite your test environment. AXUI use a config file to specify these configurations, config format is compatible with RFC 822, so that we can parse it with python built-in ConfigParser module

Below is the default configuration file in `AXUI/global.cfg`:

```
[logging]
#logger name
logger_name = AXUI

#logging level
#valid levels are DEBUG,INFO,WARNING,ERROR,CRITICAL
logging_level_file = DEBUG
logging_level_stream = ERROR

#logging file name
logging_file = AXUI.log

#logging file mode
#"w" for overwrite, will create a new file
#"a" for append, will append the log if there is an existing file
file_logging_mode = a

#logging format
#please check https://docs.python.org/2/library/logging.html#logrecord-attributes
#for more available formats
formatter = %(message)s
```

```
#if enable colorful logging, "True" or "False"
color_enable = True

[XML]
#location where you store your app maps, should be an absolute path
#set this location wrong could cause your app map loading fail
app_map_location = abspath

#location where you store your schema, should be an absolute path
#usually you do not need to change the default schema
#so do not set this unless you know what your are doing
#schema_location = abspath

#global timeout for UI response
time_out = 5

#screenshot file location
#need abspath
screenshot_location = abspath

#enable screenshot when fail happens
#can only set to True or False, other value will be ignore
screenshot_on_failure = False

[image]
#if generate diff image
#can only set to True or False, other value will be ignore
gen_diff_image = True

#diff image location
#need abspath
diff_image_location = abspath

[driver]
#driver used in your UI automation
driver_used = windows
```

AXUI configuration sections

Just introduce some common used configure settings

logging

- logger_name: you can change logger_name to keep consistent with your app
- logging_file: AXUI log file, you can specify a abs path or relative path

XML

- app_map_location: you need to set your app map location, make sure AXUI can find an app map file has same name as your specified
- schema_location: default is `AXUI/XML/schema`, you usually do not need to change it

- time_out: global timeout for UI response, you can set it bigger for slow machine/website
- screenshot_on_failure: set this true will turn on screen shot when UI operation fails

image

- gen_diff_image: set this to true will generate diff image for image compare

driver

- driver_used: your driver module name used currently, like "windows" for windows driver

CHAPTER 3

AXUI AppMap

Page Status Development

Last Reviewed

AppMap overview

AppMap is a key feature of AXUI, use AppMap smart can make your automation task much easier. Basically, AppMap is to store all changeable UI element features, and provide a consistent AppMap element for use. Thus, AppMap could reduce the scripts mutability, and make script easier for maintenance. This feature also provide a direct solution for internationalization.

We will provide a detail introduce for each part of AppMap below, before that, here provide a sample AppMap containing all parts of AppMap:

```
<AXUI:app_map xmlns:AXUI="AXUI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"␣
↪xsi:schemaLocation="AXUI AXUI_app_map.xsd">

<AXUI:variables>
    <AXUI:variable name="" value="" />
</AXUI:variables>

<AXUI:includes>
    <AXUI:include name="" path="" />
</AXUI:includes>

<AXUI:funcs>
    <AXUI:func name="" description="">
        <AXUI:step type="" cmd=''/>
    </AXUI:func>
</AXUI:funcs>

<AXUI:UI_elements>
    <AXUI:Root_element name=""/>
```

```
    <AXUI:UI_element name="" parent="" identifier="" start_func="" stop_func="">
        <AXUI:UI_element name="" identifier="" start_func="">
            <AXUI:UI_element name="" identifier="" />
        </AXUI:UI_element>
    </AXUI:UI_element>
</AXUI:UI_elements>
```

</AXUI:app_map>

AppMap elements

Here we have an introduce for each element in AppMap, AppMap will be checked with a pre-defined schema `AXUI/XML/schemas/AXUI_app_map.xsd`

root element – AXUI:app_map

`AXUI:app_map` is the root element of AppMap, it should always be:

```
<AXUI:app_map xmlns:AXUI="AXUI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"␣
↪xsi:schemaLocation="AXUI AXUI_app_map.xsd">
    ......
</AXUI:app_map>
```

AXUI:variables

`AXUI:variables` can include multiple `AXUI:variable` elements, `AXUI:variables` has no attributes.

AXUI:variable

`AXUI:variable` element provide a dynamic way to change AppMap contents, currently only support change command line in `AXUI:func` element

typical usage:

```
<AXUI:variables>
    <AXUI:variable name="AppPath" value='"some_app_path"'/>
</AXUI:variables>

<AXUI:funcs>
    <AXUI:func name="start_some_element" description="">
        <AXUI:step type="CLI" cmd='{AppPath}'/>
    </AXUI:func>
</AXUI:funcs>
```

OR:

```
<AXUI:funcs>
    <AXUI:func name="start_some_element" description="">
        <AXUI:step type="CLI" cmd='{AppPath}'/>
    </AXUI:func>
</AXUI:funcs>
```

```
#in script
appmap.variables["AppPath"] = "some_app_path"
```

AXUI:includes

`AXUI:includes` can include multiple `AXUI:include` elements, `AXUI:includes` has no attributes.

AXUI:include

AXUI support include other AppMap to use other AppMap definitions by using `AXUI:include`, this is used when some app is used by other apps, or you want to extend original app UI elements.

typical usage:

```
<AXUI:includes>
    <AXUI:include name="namespace_name" path="included_appmap_name" />
    ...
</AXUI:includes>

#in script
appmap.namespace_name.element_name...
```

OR using it as element parent:

```
<AXUI:includes>
    <AXUI:include name="namespace_name" path="included_appmap_name" />
    ...
</AXUI:includes>

<AXUI:UI_elements>
    <AXUI:UI_element name="" parent="namespace_name.element_name" />
</AXUI:UI_elements>
```

AXUI:funcs

`AXUI:funcs` can include multiple `AXUI:func` element, `AXUI:funcs` has no attributes.

AXUI:func

`AXUI:func` can include multiple `AXUI:step` element, `AXUI:func` has two attributes "name" and "description". "name" attribute is to provide an identifier to use this function, this attribute is must have, "description" provide a description for this function, it's optional.

Note: it's recommend to use `AXUI:func` element for UI element start/stop function

not recommend to use `AXUI:func` to replace python functions

AXUI:step

`AXUI:step` has three attributes: "type", "cmd", "app_map". "type" specifies step type, could be "CLI" for command or "GUI" for UI operation "cmd" specifies the detail executing step, for "CLI" type, it's a string of command line; for "GUI" type, it's a string of GUI command just like AXUI command in appmap "app_map" is optional, indicate which AppMap to execute the step

typical usage:

```
<AXUI:funcs>
    <AXUI:func name="wmplayer_start_open_dialog" description="">
        <AXUI:step type="CLI" cmd='"C:\Program Files\Windows Media Player\wmplayer.exe
↪"'/>
        <AXUI:step type="GUI" cmd="wmplayer_Window.keyboard.Input '^o'"/>
        ...
    </AXUI:func>
    ...
</AXUI:funcs>

<AXUI:UI_elements>
    <AXUI:UI_element name="OpenDialog" parent="" start_func="wmplayer_start_open_
↪dialog"/>
</AXUI:UI_elements>
```

AXUI:UI_elements

`AXUI:UI_elements` can include multiple `AXUI:UI_element` \ `AXUI:Root_element` \ `AXUI:UI_element_group` elements, `AXUI:UI_elements` has no attributes.

AXUI:Root_element

`AXUI:Root_element` element represents the enter point of UI automation API, like desktop of windows UIA, web browser of `WebDriver`.

`AXUI:Root_element` element can include multiple `AXUI:UI_element` \ `AXUI:UI_element_group` elements, element included are treated as children of `AXUI:Root_element` element.

`AXUI:Root_element` element has a `name` attribute as identifier.

AXUI:UI_element

`AXUI:UI_element` element represents the normal UI elements,

`AXUI:UI_element` element can contain multiple `AXUI:UI_element` \ `AXUI:UI_element_group` elements, element included are treated as children of `AXUI:UI_element` element.

`AXUI:UI_element` element has six attributes:

```
- ``name`` attribute, must have, identifier of this element
- ``parent`` attribute, must have for elements of the direct children of ``AXUI:UI_
↪elements``
- ``identifier`` attribute, optional, a string for UI API to find the element, check␣
↪:ref:`AXUI built-in drivers` for detail identifier format
- ``start_func`` attribute, optional, how to start the element
```

```
- ``stop_func`` attribute, optional, how to stop the element
- ``timeout`` attribute, optional, element unique timeout time, will replace global␣
↪timeout in config file
```

AXUI:UI_element_group

`AXUI:UI_element_group` element represents the UI element list,

`AXUI:UI_element_group` element cannot contain any element,

`AXUI:UI_element_group` element also has six attributes:

```
- ``name`` attribute, must have, identifier of this element
- ``parent`` attribute, must have for elements of the direct children of ``AXUI:UI_
↪elements``
- ``identifier`` attribute, optional, a string for UI API to find the element, check␣
↪:ref:`AXUI built-in drivers` for detail identifier format
- ``start_func`` attribute, optional, how to start the element
- ``stop_func`` attribute, optional, how to stop the element
- ``timedelay`` attribute, optional, will find the element group after time specified
```

CHAPTER 4

AXUI built-in drivers

Page Status Development

Last Reviewed

AXUI has implement some drivers for common used platforms, this chapter will have a introduce about this three drivers

- windows desktop, based on windows native UIAutomation Client API
- web, based on selenium project
- mobile android/ios, based on appium project

driver for windows UIAutomation API

We use comtypes to access this windows COM API, thus to use AXUI to automate windows UI, you need install comtypes first.

Windows UIAutomation API separates operations for different kinds of UI into a set of control patterns, it's recommended to use these patterns to operate target UI, AXUI expose these patterns to end users, anyway end users should need to have a check of these patterns.

Note: I have tested windows driver on win8.1 and win10,

- it's works well with windows UI framework like win32, winform, WPF, windows store app
- not works well with Qt framework, UIA recognize all kinds of Qt controls as frames
- not support for directX, custom UI controls

UIA identifiers

UIA supports different ways to find UI elements, AXUI only uses FindFisrt and FindAll to find UI element.

For search scope, AXUI uses *TreeScope_Descendants* as default, only uses *TreeScope_Children* for find element under root element.

Basically, AXUI supports most of UIA search conditions, in a different flavour using AppMap. The search condition in AppMap has a structure like:

```
identifier="<key=value [AND key=value] [OR key=value]>"
```

the identifier key is from UIA property identifiers, we strip the repeat part and get our AXUI identifier key, like:

```
UIA_NamePropertyId  -> [UIA_]Name[PropertyId] -> Name
```

- single property condition, like: `"Name='element_name'"`
- simple and/or condition, like: `"Name='element_name' AND IsEnabled=True"`, `"Name='element_name_1' OR Name='element_name_1'"`
- multiple and/or condition, like: `"Name='element_name_1' OR Name='element_name_1' AND IsEnabled=True"`

Note: Suggest using inspect tool to retrieve UI identifier values

UIA element properties

UIA element has properties that we can retrieve and check their values, to get these properties value, we just need to append the property name after the element, just like normal python properties:

```
appmap.<element_1>.[element_2]...[element_n].property_name
```

property_name is from UIA property identifiers, just like AXUI identifier key:

```
UIA_NamePropertyId  -> [UIA_]Name[PropertyId] -> Name
```

UIA element patterns

As said before, UIA split interfaces for different UI element into different control patterns, AXUI porting these pattern interfaces untouched, you can use these pattern interfaces directly:

```
appmap.<element_1>.[element_2]...[element_n].pattern_name.[pattern_method][pattern_
↪property]
```

Notice pattern_name is from control patterns, with "IUIAutomation" prefix stripped:

```
IUIAutomationValuePattern -> [IUIAutomation]ValuePattern -> ValuePattern
```

driver for WebDriver compatible projects

`selenium webdriver` and `appium` all use a C/S structure to support multiple languages, the client side and server side use WebDriver protocol to communicate with each other. since `selenium webdriver` and `appium` already have python clients, we don't reinvent the wheel, but use these python clients to implement our drivers for AXUI

Note: Since I only little experience for web UI and mobile UI automation, these driver could be not good to use

Welcome if somebody to write better drivers to replace my reference drivers.

selenium webdriver

selenium identifiers

All selenium identifiers to search elements is in `selenium/webdriver/common/by.py`, we can use these identifiers to search elements in AXUI, like:

```
identifier="key=value"
```

It's similar with windows driver, but not supports and/or search condition. The identifier key is property names of `By` class. Like "ID", "XPATH", "TAG_NAME"...

selenium properties

We can retrieve the selenium element's properties just like normal python properties:

```
appmap.<element_1>.[element_2]...[element_n].property_name
```

`property_name` is same with selenium element property name

selenium patterns/interfaces

I have restructured the selenium element methods to different pattern class, so you cannot access selenium element methods directly. Currently there are 4 pattern interfaces:

`Keyboard` interface

This interface has one method "input", to replace selenium "send_keys" method. For input normal keys like [0~9][a~z][A~Z], input directly:

```
appmap.<element_1>.[element_2]...[element_n].Keyboard.input("123asdfADSD")
```

For special charactors like "space", "tab", "newline", "F1~F12", You use {key_name} to replace them, all support keys in "selenium/webdriver/common/keys".

appmap.<element_1>.[element_2]...[element_n].Keyboard.input("{space}", "{tab}", "{F1}")

`Mouse` interface

This interface has one method "left_click", to replace selenium "click" method:

```
appmap.<element_1>.[element_2]...[element_n].Mouse.left_click()
```

`WebUIElementPattern` interface

This interface wrap original selenium methods for normal web element:

```
interfaces = [
    "submit",
    "clear",

    "is_selected",
    "is_enabled",
    "is_displayed",

    "value_of_css_property",
]
```

Use like:

```
appmap.<element_1>.[element_2]...[element_n].WebUIElementPattern.is_enabled()
```

`BrowserPattern` interface

This interface wrap original selenium methods for browser element:

```
interfaces = [
    "get",
    "close",
    "maximize_window",

    "execute_script",
    "execute_async_script",
    "set_script_timeout",

    "back",
    "forward",
    "refresh",

    "get_cookies",
    "get_cookie",
    "delete_cookie",
    "delete_all_cookies",
    "add_cookie",

    "implicitly_wait",
    "set_page_load_timeout",

    "set_window_size",
    "get_window_size",
    "set_window_position",
    "get_window_position",

    "get_log",
]
```

Use like:

```
appmap.<element_1>.[element_2]...[element_n].BrowserPattern.get("http://www.bing.com
↪")
```

Note: I have tested selenium driver with some browsers on windows, seems selenium webdriver has some problems with IE 11.

Suggest use window driver to test IE's UI, windows UIA supports IE pretty well.

appium

Note: since I don't have an apple/android environment, the appium driver is not tested

I will be very glad someone can have a test for it :)

CHAPTER 5

Extend AXUI

Page Status Development

Last Reviewed

AXUI driver interface

AXUI is first developed for easy use of windows UIAutomation API, then restructure to add support for WebDriver API used by `selenium` and `appium`. So if your UI automation is similar to windows UIAutomation API or WebDriver API, it will be easy to add support for it in AXUI.

AXUI driver interface exposes some basic interface for used in other AXUI modules,

details interfaces definition is in `driver/template/UIElement.py`

```
class UIElement(object):
    '''This class defines interfaces for common UI element

    Every driver (Windows, Appium, Selenium) should implement this interfaces,
    provides independent interfaces for uplevel modules, so we transplant AXUI cross␣
↪different platform

    Attributes:
        find_element:           find the first descendant element which matches␣
↪parsed_identifier
        find_elements:          find all elements which match parsed_identifier
        verify:                 verify current element is valid

        get_keyboard:           class for keyboard related methods
        get_mouse:              class for mouse related methods
        get_touch:              class for touch related methods

        get_property:           get property value for current element
        get_pattern:            get pattern interface for current element
    '''
```

```
    def find_element(self, parsed_identifier):
        '''
        find the first child UI element via identifier, return one UIAElement if␣
↪success, return None if not find
        '''
        raise NotImplementedError("Not implement")

    def find_elements(self, parsed_identifier):
        '''
        find the child UI elements via identifier, return a list containing target UI␣
↪elements
        '''
        raise NotImplementedError("Not implement")

    def verify(self):
        '''
        verify UI element is still exist
        '''
        raise NotImplementedError("Not implement")

    def get_property(self, name):
        '''
        get property value
        '''
        raise NotImplementedError("Not implement")

    def get_pattern(self, name):
        '''
        pattern is a class support one kind of UI actions
        '''
        raise NotImplementedError("Not implement")

    def get_keyboard(self):
        '''
        get keyboard class to use keyboard related methods
        '''
        raise NotImplementedError("Not implement")

    def get_mouse(self):
        '''
        get mouse class to use mouse related methods
        '''
        raise NotImplementedError("Not implement")

    def get_touch(self):
        '''
        get touch class to use touch related methods
        '''
        raise NotImplementedError("Not implement")

    def __getattr__(self, name):
        if name == "Keyboard":
            return self.get_keyboard()
        elif name == "Mouse":
            return self.get_mouse()
        elif name == "Touch":
            return self.get_touch()
        else:
```

```
            attr = self.get_property(name)
            if attr is not None:
                return attr
            attr = self.get_pattern(name)
            if attr is not None:
                return attr
            raise AttributeError("Attribute not exist: %s" % name)

class Root(UIElement):
    '''
    root is the entry point to interact with UI
    like desktop of windows UIA, web browser of web driver API

    This class defines interfaces for root element

    Every driver (Windows, Appium, Selenium) should implement this interfaces,
    provides independent interfaces for uplevel modules, so we transplant AXUI cross␣
↪different platform

    Attributes:
        start:                  start root element
        stop:                   stop root element
        screenshot:             take a screen shot for root element

        find_element:           find the first descendant element which matches␣
↪parsed_identifier
        find_elements:          find all elements which match parsed_identifier
        verify:                 verify current element is valid

        get_keyboard:           class for keyboard related methods
        get_mouse:              class for mouse related methods
        get_touch:              class for touch related methods

        get_property:           get property value for current element
        get_pattern:            get pattern interface for current element
    '''
    def start(self, **kwargs):
        '''
        get root ready
        like get root element in windows UIA, get browser to target website
        '''
        raise NotImplementedError("Not implement")

    def stop(self, **kwargs):
        '''
        stop root
        like close browser for web driver API
        '''
        raise NotImplementedError("Not implement")

    def screenshot(self, absfile_path):
        '''
        take a screen shot for root
        '''
        raise NotImplementedError("Not implement")
```

Implement you own driver

Most of platforms already supported by AXUI built-in driver, anyway AXUI still open for new drivers. You can refer to upper built-in drivers to write your driver.

Note: Before you start to implement your own driver for your UI, always be sure you really need to do this

For Custom UI, if your developers have provided you a command line back door to control the UI, you do not need to write your own UI driver.

CHAPTER 6

Best practices for UI Automation

Page Status Development

Last Reviewed

1. 80% principle, 80% tests only cost you 20% effect, the rest 20% tests cost 80% effect, be wise of what tests need to be automated
2. Choose a good automation tool
3. Use a template for test scripts, call a meeting to sync with all testers what the template should be like
4. Use a case management system
5. Use a test framework

CHAPTER 7

Appendices

Page Status Development

Last Reviewed

Samples

sample codes in `AXUI/example`, here I give a step by step guide how to launch the examples

windows

My test environment is win8.1 64bit, anyway win8.1 32bit machine is OK. For win7/win8, you may need to change the AppMap, since UI changes for different windows version.

Prepare Environment

1. Install python 2.7
2. Install setuptools + pip. check python package management
3. Install comtypes
4. Install AXUI
5. Config windows path, add python path (usually `c:\python27`) to windows path

Run the example in `example/windows`

1. Modify the config file, check *AXUI configurations*
2. Run the script, `python wmplayer_wrapper.py`

selenium webdriver

My test environment is win8.1 64bit, with chrome browser. Other environment supported by selenium should also be OK.

Prepare Environment

1. Install python 2.7
2. Install setuptools + pip. check python package management
3. Install selenium
4. Install AXUI
5. Config windows path, add python path (usually `c:\python27`) to windows path

Run the example in `example/selenium`

1. Modify the config file, check *AXUI configurations*
2. Run the script, `python bing.py`

appium webdriver

Prepare Environment

My test environment is win7 64bit, here is my steps to prepare appium environment.

1. Install appium, two methods for windows
    1. Install node.js
    2. Install appium, `npm install -g appium`

    OR

    1. Download and install AppiumForWindow from github
2. Prepare Android develop environment, include SDK and AVD. Check this website for available mirrors for china
3. Install selenium
4. Install Appium-Python-Client
5. Install AXUI

Run the example in `example/appium`

1. Launch appium server, you need set the Android SDK path for appium server, should not contain white space.
2. Modify the config file, check *AXUI configurations*
3. Run the script, `python simple.py`, this sample is modified from appium sample code
4. Check the AVD is act as expected

ToDo list

- Multiple languages support, to support internationalization test
- Define an unified interfaces for different driver UI actions
- Support for python 3

CHAPTER 8

Indices and tables

- genindex
- modindex
- search