
Autopilot Documentation

Release 1.5

Thomi Richards

Jul 07, 2017

Contents

1	What is Autopilot, and what can it do?	1
2	Where is Autopilot used?	3
3	How does Autopilot fit with other test frameworks?	5
4	What do Autopilot Tests Contain?	7
5	Autopilot Tutorial	9
5.1	Writing Your First Test	9
5.2	Advanced Autopilot Features	16
6	Writing Good Autopilot Tests	29
6.1	Write Expressive Tests	29
6.2	Test One Thing Only	30
6.3	Fail Well	30
6.4	Think about design	31
6.5	Test Length	31
6.6	Good docstrings	32
6.7	Test Readability	33
6.8	Prefer <code>wait_for</code> and <code>Eventually to sleep</code>	35
6.9	Scenarios	36
6.10	Do Not Depend on Object Ordering	39
7	Running Autopilot	41
7.1	List Tests	41
7.2	Run Tests	42
7.3	Launching an Application to Introspect	43
7.4	Visualise Introspection Tree	43
8	Installing Autopilot	47
8.1	Ubuntu	47
8.2	Other Linux's	48
9	Autopilot API Documentation	49
9.1	<code>autopilot</code> - Global stuff	50
9.2	<code>autopilot.application</code> - Autopilot Application Launchers	50
9.3	<code>autopilot.display</code> - Get information about the current display(s)	50

9.4	<code>autopilot.emulators</code> - Backwards compatibility for autopilot v1.2	50
9.5	<code>autopilot.exceptions</code> - Autopilot Exceptions	50
9.6	<code>autopilot.gestures</code> - Gestural and multi-touch support	50
9.7	<code>autopilot.input</code> - Generate keyboard, mouse, and touch input events	50
9.8	<code>autopilot.introspection</code> - Retrieve proxy objects	50
9.9	<code>autopilot.introspection.types</code> - Introspection Type details	50
9.10	<code>autopilot.matchers</code> - Custom matchers for test assertions	50
9.11	<code>autopilot.platform</code> - Functions for platform detection	50
9.12	<code>autopilot.process</code> - Process Control	50
9.13	<code>autopilot.testcase</code> - Base class for all Autopilot Test Cases	50
10	Frequently Asked Questions	51
10.1	Autopilot: The Project	52
10.2	Autopilot Tests	52
10.3	Autopilot Tests and Launching Applications	54
10.4	Autopilot Qt & Gtk Support	55
11	Porting Autopilot Tests	57
11.1	A note on Versions	57
11.2	Porting to Autopilot v1.4.x	58
11.3	Porting to Autopilot v1.3.x	59
12	Appendices	61
12.1	XPathSelect Query Protocol	61
13	Autopilot Man Page	71
13.1	SYNOPSIS	71
13.2	DESCRIPTION	71
13.3	OPTIONS	71
13.4	SPECIFYING SUITES	72

What is Autopilot, and what can it do?

Autopilot is a tool for writing functional tests. Functional tests are tests that:

- Run out-of-process. I.e.- the tests run in a separate process to the application under test.
- Simulate user interaction. Autopilot provides methods to generate keyboard, mouse, and touch events. These events are delivered to the application under test in exactly the same way as normal input events. The application under test therefore cannot distinguish between a “real” user and an autopilot test case.
- Validate design decisions. The primary function of a functional test is to determine whether or not an application has met the design criteria. Functional tests evaluate high-level design correctness.

CHAPTER 2

Where is Autopilot used?

Autopilot was designed to test the [Unity 3D](#) shell. However, since then it has been used to test a number of other applications, including:

- Core Ubuntu GUI applications.
- Mobile phone applications for the Ubuntu Phone & Ubuntu Tablet.

How does Autopilot fit with other test frameworks?

Autopilot exists at the apex of the “testing pyramid”. It is designed to test high-level functionality, and complement a solid base of unit and integration tests. *Using autopilot is not a substitute for testing your application with unit and integration tests!*. Autopilot is a very capable tool for testing high-level feature functionality. It is not an appropriate tool for testing low-level implementation details.

Autopilot is built on top of several other python test frameworks, including:

- **Python Testtools** - `AutopilotTestCase` derives from the `testtools.TestCase` class, which allows test author to use all the extended features found in `testtools`. Specifically, Autopilot includes the `Eventually` matcher class, which allows test authors to make assertions about the application under test without having to worry about the timing between the tests and the application under test.
- **Python Test Scenarios** - `AutopilotTestCase` contains the necessary plumbing in order to allow test authors to use test scenarios out of the box. This is extremely useful when you want to test several different modes of operation.
- **Python Test Fixtures** - Several parts of autopilot are built as fixtures. While this is rarely exposed to the test author, it can be useful to know that this functionality is always present whenever autopilot is installed.

What do Autopilot Tests Contain?

A typical autopilot test has three distinct stages:

The Setup Stage

There are several concerns that must be addressed in the setup Phase. The most important step is to launch the application to be tested. Most autopilot test suites launch the application under test anew for each test. This ensures that the test starts with the application under test in a known, clean state. Autopilot can launch normal applications, launch applications via upstart, or launch apps contained within a click package.

Tests may also wish to take other actions in the setup stage, including:

- Setting environment variables to certain values.
- Starting external applications that are required for the test to run.
- Creating files or folders (or any kind of external data) on disk.

The purpose of the setup stage is to make sure that everything that is required for the test to run is in place.

The Interaction Stage

Once the setup has been completed, it's time to start interacting with your application. This typically involves generating input events. For example, if you are testing a text editor you might have a test whose specification is similar to the following:

Type some text into the document area, open the 'Edit' menu and click the 'Search and Replace' menu item.

During this stage you will most likely need to read the applications internal state. For example, your test will need to know where the 'Edit' menu is on the screen. Thankfully, autopilot takes care of the details, allowing you to write expressive tests.

The Assertion Stage

The final stage is where you determine if your test should pass or fail. Most tests will contain more than one assertion (*why?*). Autopilot contains several custom assertions that make testing high-level concepts easier.

This tutorial will guide users new to Autopilot through creating a minimal autopilot test.

Writing Your First Test

This document contains everything you need to know to write your first autopilot test. It covers writing several simple tests for a sample Qt5/Qml application. However, it's important to note that nothing in this tutorial is specific to Qt5/Qml, and will work equally well with any other kind of application.

Files and Directories

Your autopilot test suite will grow to several files, possibly spread across several directories. We recommend that you follow this simple directory layout:

```
autopilot/  
autopilot/<projectname>/  
autopilot/<projectname>/tests/
```

The `autopilot` folder can be anywhere within your project's source tree. It will likely contain a `setup.py` file.

The `autopilot/<projectname>/` folder is the base package for your autopilot tests. This folder, and all child folders, are python packages, and so must contain an `__init__.py` file. If you ever find yourself writing custom proxy classes (This is an advanced topic, and is covered here: [Writing Custom Proxy Classes](#)), they should be imported from this top-level package.

Each test file should be named `test_<component>.py`, where `<component>` is the logical component you are testing in that file. Test files must be written in the `autopilot/<projectname>/tests/` folder.

A Minimal Test Case

Autopilot tests follow a similar pattern to other python test libraries: you must declare a class that derives from `AutopilotTestCase`. A minimal test case looks like this:

```
from autopilot.testcase import AutopilotTestCase

class MyTests(AutopilotTestCase):

    def test_something(self):
        """An example test case that will always pass."""
        self.assertTrue(True)
```

The Setup Phase

Before each test is run, the `setUp` method is called. Test authors may override this method to run any setup that needs to happen before the test is run. However, care must be taken when using the `setUp` method: it tends to hide code from the test case, which can make your tests less readable. It is our recommendation, therefore, that you use this feature sparingly. A more suitable alternative is often to put the setup code in a separate function or method and call it from the test function.

Should you wish to put code in a setup method, it looks like this:

```
from autopilot.testcase import AutopilotTestCase

class MyTests(AutopilotTestCase):

    def setUp(self):
        super(MyTests, self).setUp()
        # This code gets run before every test!

    def test_something(self):
        """An example test case that will always pass."""
        self.assertTrue(True)
```

Note: Any action you take in the setup phase must be undone if it alters the system state. See [Cleaning Up](#) for more details.

Starting the Application

At the start of your test, you need to tell autopilot to launch your application. To do this, call `launch_test_application`. The minimum required argument to this method is the application name or path. If you pass in the application name, autopilot will look in the current working directory, and then will search the `PATH` environment variable. Otherwise, autopilot looks for the executable at the path specified. Positional arguments to this method are passed to the executable being launched.

Autopilot will try and guess what type of application you are launching, and therefore what kind of introspection libraries it should load. Sometimes autopilot will need some assistance however. For example, at the time of writing, autopilot cannot automatically detect the introspection type for python / Qt4 applications. In that case, a `RuntimeError` will be raised. To provide autopilot with a hint as to which introspection type to load, you can provide the `app_type` keyword argument. For example:

```
class MyTests(AutopilotTestCase):

    def test_python_qt4_application(self):
```

```
self.app = self.launch_test_application(
    'my-pyqt4-app',
    app_type='qt'
)
```

See the documentation for `launch_test_application` for more details.

The return value from `launch_test_application` is a proxy object representing the root of the introspection tree of the application you just launched.

A Simple Test

To demonstrate the material covered so far, this selection will outline a simple application, and a single test for it. Instead of testing a third-party application, we will write the simplest possible application in Python and Qt4. The application, named `testapp.py`, is listed below:

```
#!/usr/bin/env python

from PyQt4 import QtGui
from sys import argv

def main():
    app = QtGui.QApplication(argv)
    win = QtGui.QMainWindow()
    win.show()
    win.setWindowTitle("Hello World")
    app.exec_()

if __name__ == '__main__':
    main()
```

As you can see, this is a trivial application, but it serves our purpose. For the upcoming tests to run this file must be executable:

```
$ chmod u+x testapp.py
```

We will write a single autopilot test that asserts that the title of the main window is equal to the string “Hello World”. Our test file is named `test_window.py`, and contains the following code:

```
from autopilot.testcase import AutopilotTestCase
from os.path import abspath, dirname, join
from testtools.matchers import Equals

class MainWindowTitleTests(AutopilotTestCase):

    def launch_application(self):
        """Work out the full path to the application and launch it.

        This is necessary since our test application will not be in $PATH.

        :returns: The application proxy object.

        """
        full_path = abspath(join(dirname(__file__), '..', '..', 'testapp.py'))
        return self.launch_test_application(full_path, app_type='qt')
```

```
def test_main_window_title_string(self):
    """The main window title must be 'Hello World'."""
    app_root = self.launch_application()
    main_window = app_root.select_single('QMainWindow')

    self.assertEqual(main_window.windowTitle, Equals("Hello World"))
```

Note that we have made the test method as readable as possible by hiding the complexities of finding the full path to the application we want to test. Of course, if you can guarantee that the application is in PATH, then this step becomes a lot simpler.

The entire directory structure looks like this:

```
./example/___init___py
./example/tests/___init___py
./example/tests/test_window.py
./testapp.py
```

The `___init___py` files are empty, and are needed to make these directories importable by python.

Running Autopilot

From the root of this directory structure, we can ask autopilot to list all the tests it can find:

```
$ autopilot3 list example
Loading tests from: /home/thomi/code/canonical/autopilot/example_test

    example.tests.test_window.MainWindowTitleTests.test_main_window_title_string

1 total tests.
```

Note that on the first line, autopilot will tell you where it has loaded the test definitions from. Autopilot will look in the current directory for a python package that matches the package name specified on the command line. If it does not find any suitable packages, it will look in the standard python module search path instead.

To run our test, we use the autopilot ‘run’ command:

```
$ autopilot3 run example
Loading tests from: /home/thomi/code/canonical/autopilot/example_test

Tests running...

Ran 1 test in 2.342s
OK
```

You will notice that the test application launches, and then disappears shortly afterwards. Since this test doesn’t manipulate the application in any way, this is a rather boring test to look at. If you ever want more output from the run command, you may specify the ‘-v’ flag:

```
$ autopilot3 run -v example
Loading tests from: /home/thomi/code/canonical/autopilot/example_test

Tests running...
13:41:11.614 INFO globals:49 -
↳*****
13:41:11.614 INFO globals:50 - Starting test example.tests.test_window.
↳MainWindowTitleTests.test_main_window_title_string
```



```

13:41:11.693 INFO __init__:136 - Launching process: ['/home/thomi/code/canonical/
↳ autopilot/example_test/testapp.py', '-testability']
13:41:11.699 INFO __init__:169 - Looking for autopilot interface for PID 12013 (and
↳ children)
13:41:11.727 WARNING __init__:185 - Caught exception while searching for autopilot
↳ interface: 'DBusException("Could not get PID of name 'org.freedesktop.DBus': no
↳ such name",)'
13:41:12.773 WARNING __init__:185 - Caught exception while searching for autopilot
↳ interface: 'DBusException("Could not get PID of name 'org.freedesktop.DBus': no
↳ such name",)'
13:41:12.848 WARNING __init__:185 - Caught exception while searching for autopilot
↳ interface: 'RuntimeError("Could not find Autopilot interface on DBus backend '
↳ <session bus :1.5967 /com/canonical/Autopilot/Introspection>",)'
13:41:12.852 WARNING __init__:185 - Caught exception while searching for autopilot
↳ interface: 'RuntimeError("Could not find Autopilot interface on DBus backend '
↳ <session bus :1.5968 /com/canonical/Autopilot/Introspection>",)'
13:41:12.863 WARNING dbus:464 - Generating introspection instance for type 'Root'
↳ based on generic class.
13:41:12.864 DEBUG dbus:338 - Selecting objects of type QMainWindow with attributes:
↳ {}
13:41:12.871 WARNING dbus:464 - Generating introspection instance for type
↳ 'QMainWindow' based on generic class.
13:41:12.886 INFO testcase:380 - waiting for process to exit.
13:41:13.983 INFO testresult:35 - OK: example.tests.test_window.MainWindowTitleTests.
↳ test_main_window_title_string

Ran 1 test in 2.370s
OK

```

You may also specify ‘-v’ twice for even more output (this is rarely useful for test authors however).

Both the ‘list’ and ‘run’ commands take a test id as an argument. You may be as generic, or as specific as you like. In the examples above, we will list and run all tests in the ‘example’ package (i.e.- all tests), but we could specify a more specific run criteria if we only wanted to run some of the tests. For example, to only run the single test we’ve written, we can execute:

```

$ autopilot3 run example.tests.test_window.MainWindowTitleTests.test_main_window_
↳ title_string

```

A Test with Interaction

Now lets take a look at some simple tests with some user interaction. First, update the test application with some input and output controls:

```

#!/usr/bin/env python
# File: testapp.py

from PyQt4 import QtGui
from sys import argv

class AutopilotHelloWorld(QtGui.QWidget):
    def __init__(self):
        super(AutopilotHelloWorld, self).__init__()

        self.hello = QtGui.QPushButton("Hello")
        self.hello.clicked.connect(self.say_hello)

```

```
self.goodbye = QtGui.QPushButton("Goodbye")
self.goodbye.clicked.connect(self.say_goodbye)

self.response = QtGui.QLabel("Response: None")

grid = QtGui.QGridLayout()
grid.addWidget(self.hello, 0, 0)
grid.addWidget(self.goodbye, 0, 1)
grid.addWidget(self.response, 1, 0, 1, 2)
self.setLayout(grid)
self.show()
self.setWindowTitle("Hello World")

def say_hello(self):
    self.response.setText('Response: Hello')

def say_goodbye(self):
    self.response.setText('Response: Goodbye')

def main():
    app = QtGui.QApplication(argv)
    ahw = AutopilotHelloWorld()
    app.exec_()

if __name__ == '__main__':
    main()
```

We’ve reorganized the application code into a class to make the event handling easier. Then we added two input controls, the hello and goodbye buttons and an output control, the response label.

The operation of the application is still very trivial, but now we can test that it actually does something in response to user input. Clicking either of the two buttons will cause the response text to change. Clicking the Hello button should result in Response: Hello while clicking the Goodbye button should result in Response: Goodbye.

Since we’re adding a new category of tests, button response tests, we should organize them into a new class. Our tests module now looks like:

```
from autopilot.testcase import AutopilotTestCase
from os.path import abspath, dirname, join
from testtools.matchers import Equals

from autopilot.matchers import Eventually

class HelloWorldTestBase(AutopilotTestCase):

    def launch_application(self):
        """Work out the full path to the application and launch it.

        This is necessary since our test application will not be in $PATH.

        :returns: The application proxy object.

        """
        full_path = abspath(join(dirname(__file__), '..', '..', 'testapp.py'))
        return self.launch_test_application(full_path, app_type='qt')
```

```

class MainWindowTitleTests (HelloWorldTestBase):

    def test_main_window_title_string(self):
        """The main window title must be 'Hello World'."""
        app_root = self.launch_application()
        main_window = app_root.select_single('AutopilotHelloWorld')

        self.assertEqual(main_window.windowTitle, Equals("Hello World"))

class ButtonResponseTests (HelloWorldTestBase):

    def test_hello_response(self):
        """The response text must be 'Response: Hello' after a Hello click."""
        app_root = self.launch_application()
        response = app_root.select_single('QLabel')
        hello = app_root.select_single('QPushButton', text='Hello')

        self.mouse.click_object(hello)

        self.assertEqual(response.text, Eventually(Equals('Response: Hello')))

    def test_goodbye_response(self):
        """The response text must be 'Response: Goodbye' after a Goodbye
        click."""
        app_root = self.launch_application()
        response = app_root.select_single('QLabel')
        goodbye = app_root.select_single('QPushButton', text='Goodbye')

        self.mouse.click_object(goodbye)

        self.assertEqual(response.text, Eventually(Equals('Response: Goodbye')))

```

In addition to the new class, `ButtonResponseTests`, you'll notice a few other changes. First, two new import lines were added to support the new tests. Next, the existing `MainWindowTitleTests` class was refactored to subclass from a base class, `HelloWorldTestBase`. The base class contains the `launch_application` method which is used for all test cases. Finally, the object type of the main window changed from `QMainWindow` to `AutopilotHelloWorld`. The change in object type is a result of our test application being refactored into a class called `AutopilotHelloWorld`.

The `ButtonResponseTests` class adds two new tests, one for each input control. Each test identifies the user interface controls that need to be used, performs a single, specific action, and then verifies the outcome. In `test_hello_response`, we first identify the `QLabel` control which contains the output we need to check. We then identify the Hello button. As the application has two `QPushButton` controls, we must further refine the `select_single` call by specifying an additional property. In this case, we use the button text. Next, an input action is triggered by instructing the mouse to click the Hello button. Finally, the test asserts that the response label text matches the expected string. The second test repeats the same process with the Goodbye button.

The Eventually Matcher

Notice that in the `ButtonResponseTests` tests above, the autopilot method `Eventually` is used in the assertion. This allows the assertion to be retried continuously until it either becomes true, or times out (the default timeout is 10 seconds). This is necessary because the application and the autopilot tests run in different processes. Autopilot could test the assert before the application has completed its action. Using `Eventually` allows the application to complete its action without having to explicitly add delays to the tests.

Advanced Autopilot Features

This document covers advanced features in autopilot.

Cleaning Up

It is vitally important that every test you run leaves the system in exactly the same state as it found it. This means that:

- Any files written to disk need to be removed.
- Any environment variables set during the test run need to be un-set.
- Any applications opened during the test run need to be closed again.
- Any Keyboard keys pressed during the test need to be released again.

All of the methods on `AutopilotTestCase` that alter the system state will automatically revert those changes at the end of the test. Similarly, the various input devices will release any buttons or keys that were pressed during the test. However, for all other changes, it is the responsibility of the test author to clean up those changes.

For example, a test might require that a file with certain content be written to disk at the start of the test. The test case might look something like this:

```
class MyTests(AutopilotTestCase):

    def make_data_file(self):
        open('/tmp/datafile', 'w').write("Some data...")

    def test_application_opens_data_file(self):
        """Our application must be able to open a data file from disk."""
        self.make_data_file()
        # rest of the test code goes here
```

However this will leave the `/tmp/datafile` on disk after the test has finished. To combat this, use the `addCleanup` method. The arguments to `addCleanup` are a callable, and then zero or more positional or keyword arguments. The Callable will be called with the positional and keyword arguments after the test has ended.

Cleanup actions are called in the reverse order in which they are added, and are called regardless of whether the test passed, failed, or raised an uncaught exception. To fix the above test, we might write something similar to:

```
import os

class MyTests(AutopilotTestCase):

    def make_data_file(self):
        open('/tmp/datafile', 'w').write("Some data...")
        self.addCleanup(os.remove, '/tmp/datafile')

    def test_application_opens_data_file(self):
        """Our application must be able to open a data file from disk."""
        self.make_data_file()
        # rest of the test code goes here
```

Note that by having the code to generate the `/tmp/datafile` file on disk in a separate method, the test itself can ignore the fact that these resources need to be cleaned up. This makes the tests cleaner and easier to read.

Test Scenarios

Occasionally test authors will find themselves writing multiple tests that differ in one or two subtle ways. For example, imagine a hypothetical test case that tests a dictionary application. The author wants to test that certain words return no results. Without using test scenarios, there are two basic approaches to this problem. The first is to create many test cases, one for each specific scenario (*don't do this*):

```
class DictionaryResultsTests (AutopilotTestCase) :

    def test_empty_string_returns_no_results(self):
        self.dictionary_app.enter_search_term("")
        self.assertEqual(len(self.dictionary_app.results), Equals(0))

    def test_whitespace_string_returns_no_results(self):
        self.dictionary_app.enter_search_term(" \t ")
        self.assertEqual(len(self.dictionary_app.results), Equals(0))

    def test_punctuation_string_returns_no_results(self):
        self.dictionary_app.enter_search_term(".-?<>{}[]")
        self.assertEqual(len(self.dictionary_app.results), Equals(0))

    def test_garbage_string_returns_no_results(self):
        self.dictionary_app.enter_search_term("ljdzgfhdsgjfhgdgjh")
        self.assertEqual(len(self.dictionary_app.results), Equals(0))
```

The main problem here is that there's a lot of typing in order to change exactly one thing (and this hypothetical test is deliberately short, to ease clarity. Imagine a 100 line test case!). Another approach is to make the entire thing one large test (*don't do this either*):

```
class DictionaryResultsTests (AutopilotTestCase) :

    def test_bad_strings_returns_no_results(self):
        bad_strings = ("",
            " \t ",
            ".-?<>{}[]",
            "ljdzgfhdsgjfhgdgjh",
        )
        for input in bad_strings:
            self.dictionary_app.enter_search_term(input)
            self.assertEqual(len(self.dictionary_app.results), Equals(0))
```

This approach makes it easier to add new input strings, but what happens when just one of the input strings stops working? It becomes very hard to find out which input string is broken, and the first string that breaks will prevent the rest of the test from running, since tests stop running when the first assertion fails.

The solution is to use test scenarios. A scenario is a class attribute that specifies one or more scenarios to run on each of the tests. This is best demonstrated with an example:

```
class DictionaryResultsTests (AutopilotTestCase) :

    scenarios = [
        ('empty string', {'input': ""}),
        ('whitespace', {'input': " \t "}),
        ('punctuation', {'input': ".-?<>{}[]"}),
        ('garbage', {'input': "ljdzgfhdsgjfhgdgjh"}),
    ]

    def test_bad_strings_return_no_results(self):
```

```
self.dictionary_app.enter_search_term(self.input)
self.assertThat(len(self.dictionary_app.results), Equals(0))
```

Autopilot will run the `test_bad_strings_return_no_results` once for each scenario. On each test, the values from the scenario dictionary will be mapped to attributes of the test case class. In this example, that means that the ‘input’ dictionary item will be mapped to `self.input`. Using scenarios has several benefits over either of the other strategies outlined above:

- Tests that use strategies will appear as separate tests in the test output. The test id will be the normal test id, followed by the strategy name in parenthesis. So in the example above, the list of test ids will be:

```
DictionaryResultsTests.test_bad_strings_return_no_results(empty_string)
DictionaryResultsTests.test_bad_strings_return_no_results(whitespace)
DictionaryResultsTests.test_bad_strings_return_no_results(punctuation)
DictionaryResultsTests.test_bad_strings_return_no_results(garbage)
```

- Since scenarios are treated as separate tests, it’s easier to debug which scenario has broken, and re-run just that one scenario.
- Scenarios get applied before the `setUp` method, which means you can use scenario values in the `setUp` and `tearDown` methods. This makes them more flexible than either of the approaches listed above.

Test Logging

Autopilot integrates the [python logging framework](#) into the `AutopilotTestCase` class. Various autopilot components write log messages to the logging framework, and all these log messages are attached to each test result when the test completes. By default, these log messages are shown when a test fails, or if autopilot is run with the `-v` option.

Test authors are encouraged to write to the python logging framework whenever doing so would make failing tests clearer. To do this, there are a few simple steps to follow:

1. Import the logging module:

```
import logging
```

2. Create a logger object. You can either do this at the file level scope, or within a test case class:

```
logger = logging.getLogger(__name__)
```

3. Log some messages. You may choose which level the messages should be logged at. For example:

```
logger.debug("This is debug information, not shown by default.")
logger.info("This is some information")
logger.warning("This is a warning")
logger.error("This is an error")
```

Note: To view log messages when using debug level of logging pass `-vv` when running autopilot.

For more information on the various logging levels, see the [python documentation on Logger objects](#). All messages logged in this way will be picked up by the autopilot test runner. This is a valuable tool when debugging failing tests.

Environment Patching

Sometimes you need to change the value of an environment variable for the duration of a single test. It is important that the variable is changed back to its original value when the test has ended, so future tests are run in a pristine environment. The `fixtures` module includes a `fixtures.EnvironmentVariable` fixture which takes care of this for you. For example, to set the `FOO` environment variable to "Hello World" for the duration of a single test, the code would look something like this:

```
from fixtures import EnvironmentVariable
from autopilot.testcase import AutopilotTestCase

class MyTests(AutopilotTestCase):

    def test_that_needs_custom_environment(self):
        self.useFixture(EnvironmentVariable("FOO", "Hello World"))
        # Test code goes here.
```

The `fixtures.EnvironmentVariable` fixture will revert the value of the environment variable to its initial value, or will delete it altogether if the environment variable did not exist when `fixtures.EnvironmentVariable` was instantiated. This happens in the cleanup phase of the test execution.

Custom Assertions

Autopilot provides additional custom assertion methods within the `AutopilotTestCase` base class. These assertion methods can be used for validating the visible window stack and also properties on objects whose attributes do not have the `wait_for` method, such as `Window` objects (See [In Proxy Classes](#) for more information about `wait_for`).

```
autopilot.testcase.AutopilotTestCase.assertVisibleWindowStack
```

This assertion allows the test to check the start of the visible window stack by passing an iterable item of `Window` instances. Minimised windows will be ignored:

```
from autopilot.process import ProcessManager
from autopilot.testcase import AutopilotTestCase

class WindowTests(AutopilotTestCase):

    def test_window_stack(self):
        self.launch_some_test_apps()
        pm = ProcessManager.create()
        test_app_windows = []
        for window in pm.get_open_windows():
            if self.is_test_app(window.name):
                test_app_windows.append(window)
        self.assertVisibleWindowStack(test_app_windows)
```

Note: The process manager is only available on environments that use `bamf`, i.e. desktop running Unity 7. There is currently no process manager for any other platform.

```
autopilot.testcase.AutopilotTestCase.assertProperty
```

This assertion allows the test to check properties of an object that does not have a **wait_for** method (i.e.- objects that do not come from the autopilot DBus interface). For example the Window object:

```
from autopilot.process import ProcessManager
from autopilot.testcase import AutopilotTestCase

class WindowTests(AutopilotTestCase):

    def test_window_stack(self):
        self.launch_some_test_apps()
        pm = ProcessManager.create()
        for window in pm.get_open_windows():
            if self.is_test_app(window.name):
                self.assertProperty(window, is_maximized=True)
```

Note: `assertProperties` is a synonym for this method.

Note: The process manager is only available on environments that use bamf, i.e. desktop running Unity 7. There is currently no process manager for any other platform.

`autopilot.testcase.AutopilotTestCase.assertProperties`

See [*autopilot.testcase.AutopilotTestCase.assertProperty*](#).

Note: `assertProperty` is a synonym for this method.

Platform Selection

Autopilot provides functionality that allows the test author to determine which platform a test is running on so that they may either change behaviour within the test or skipping the test all together.

For examples and API documentaion please see `autopilot.platform`.

Gestures and Multi-touch

Autopilot provides API support for both *single-touch* and *multi-touch* gestures which can be used to simulate user input required to drive an application or system under test. These APIs should be used in conjunction with *Platform Selection* to detect platform capabilities and ensure the correct input API is being used.

Single-Touch

`autopilot.input.Touch` provides single-touch input gestures, which includes:

- `tap` which can be used to tap a specified `[x,y]` point on the screen
- `drag` which will drag between 2 `[x,y]` points and can be customised by altering the speed of the action
- `press`, `release` and `move` operations which can be combined to create custom gestures

- `tap_object` can be used to tap the center point of a given introspection object, where the screen co-ordinates are taken from one of several properties of the object

Autopilot additionally provides the class `autopilot.input.Pointer` as a means to provide a single unified API that can be used with both `Mouse` input and `Touch` input. See the documentation for this class for further details of this, as not all operations can be performed on both of these input types.

This example demonstrates swiping from the center of the screen to the left edge, which could for example be used in [Ubuntu Touch](#) to swipe a new scope into view.

1. First calculate the center point of the screen (see: [Display Information](#)):

```
>>> from autopilot.display import Display
>>> display = Display.create()
>>> center_x = display.get_screen_width() // 2
>>> center_y = display.get_screen_height() // 2
```

2. Then perform the swipe operation from the center of the screen to the left edge, using `autopilot.input.Pointer.drag`:

```
>>> from autopilot.input import Touch, Pointer
>>> pointer = Pointer(Touch.create())
>>> pointer.drag(center_x, center_y, 0, center_y)
```

Multi-Touch

`autopilot.gestures` provides support for multi-touch input which includes:

- `autopilot.gestures.pinch` provides a 2-finger pinch gesture centered around an `[x,y]` point on the screen

This example demonstrates how to use the pinch gesture, which for example could be used on [Ubuntu Touch](#) web-browser, or gallery application to zoom in or out of currently displayed content.

1. To zoom in, pinch vertically outwards from the center point by 100 pixels:

```
>>> from autopilot import gestures
>>> gestures.pinch([center_x, center_y], [0, 0], [0, 100])
```

2. To zoom back out, pinch vertically 100 pixels back towards the center point:

```
>>> gestures.pinch([center_x, center_y], [0, 100], [0, 0])
```

Note: The multi-touch `pinch` method is intended for use on a touch enabled device. However, if run on a desktop environment it will behave as if the mouse select button is pressed whilst moving the mouse pointer. For example to select some text in a document.

Advanced Backend Picking

Several features in autopilot are provided by more than one backend. For example, the `autopilot.input` module contains the `Keyboard`, `Mouse` and `Touch` classes, each of which can use more than one implementation depending on the platform the tests are being run on.

For example, when running autopilot on a traditional ubuntu desktop platform, Keyboard input events are probably created using the X11 client libraries. On a phone platform, X11 is not present, so autopilot will instead choose to generate events using the kernel UInput device driver instead.

Other autopilot systems that make use of multiple backends include the `autopilot.display` and `autopilot.process` modules. Every class in these modules follows the same construction pattern:

Default Creation

By default, calling the `create()` method with no arguments will return an instance of the class that is appropriate to the current platform.

```
>>> from autopilot.input import Keyboard
>>> kbd = Keyboard.create()
```

The code snippet above will create an instance of the Keyboard class that uses X11 on Desktop systems, and UInput on other systems. On the rare occasion when test authors need to construct these objects themselves, we expect that the default creation pattern to be used.

Picking a Backend

Test authors may sometimes want to pick a specific backend. The possible backends are documented in the API documentation for each class. For example, the documentation for the `autopilot.input.Keyboard.create` method says there are three backends available: the X11 backend, the UInput backend, and the OSK backend. These backends can be specified in the create method. For example, to specify that you want a Keyboard that uses X11 to generate its input events:

```
>>> from autopilot.input import Keyboard
>>> kbd = Keyboard.create("X11")
```

Similarly, to specify that a UInput keyboard should be created:

```
>>> from autopilot.input import Keyboard
>>> kbd = Keyboard.create("UInput")
```

Finally, for the Onscreen Keyboard:

```
>>> from autopilot.input import Keyboard
>>> kbd = Keyboard.create("OSK")
```

Warning: Care must be taken when specifying specific backends. There is no guarantee that the backend you ask for is going to be available across all platforms. For that reason, using the default creation method is encouraged.

Warning: The **OSK** backend has some known implementation limitations, please see `autopilot.input.Keyboard.create` method documentation for further details.

Possible Errors when Creating Backends

Lots of things can go wrong when creating backends with the `create` method.

If autopilot is unable to create any backends for your current platform, a `RuntimeError` exception will be raised. Its `message` attribute will contain the error message from each backend that autopilot tried to create.

If a preferred backend was specified, but that backend doesn't exist (probably the test author mis-spelled it), a `RuntimeError` will be raised:

```
>>> from autopilot.input import Keyboard
>>> try:
...     kbd = Keyboard.create("uinput")
... except RuntimeError as e:
...     print("Unable to create keyboard: " + str(e))
...
Unable to create keyboard: Unknown backend 'uinput'
```

In this example, `uinput` was mis-spelled (backend names are case sensitive). Specifying the correct backend name works as expected:

```
>>> from autopilot.input import Keyboard
>>> kbd = Keyboard.create("UInput")
```

Finally, if the test author specifies a preferred backend, but that backend could not be created, a `autopilot.BackendException` will be raised. This is an important distinction to understand: While calling `create()` with no arguments will try more than one backend, specifying a backend to create will only try and create that one backend type. The `BackendException` instance will contain the original exception raised by the backend in its `original_exception` attribute. In this example, we try and create a `UInput` keyboard, which fails because we don't have the correct permissions (this is something that autopilot usually handles for you):

```
>>> from autopilot.input import Keyboard
>>> from autopilot import BackendException
>>> try:
...     kbd = Keyboard.create("UInput")
... except BackendException as e:
...     repr(e.original_exception)
...     repr(e)
...
'UInputError(\'"/dev/uinput" cannot be opened for writing\',)'
'BackendException(\'Error while initialising backend. Original exception was: "/dev/
↪uinput" cannot be opened for writing\',)'
```

Keyboard Backends

A quick introduction to the Keyboard backends

Each backend has a different method of operating behind the scenes to provide the Keyboard interface.

Here is a quick overview of how each backend works.

Back-end	Description
X11	The X11 backend generates X11 events using a mock input device which it then syncs with X to actually action the input.
Uinput	The UInput backend injects events directly in to the kernel using the UInput device driver to produce input.
OSK	The Onscreen Keyboard backend uses the GUI pop-up keyboard to enter input. Using a pointer object it taps on the required keys to get the expected output.

Limitations of the different Keyboard backends

While every effort has been made so that the Keyboard devices act the same regardless of which backend or platform is in use, the simple fact is that there can be some technical limitations for some backends.

Some of these limitations are hidden when using the “create” method and won’t cause any concern (e.g. X11 backend on desktop, UInput on an Ubuntu Touch device.) while others will raise exceptions (that are fully documented in the API docs).

Here is a list of known limitations:

X11

- Only available on desktop platforms
 - X11 isn’t available on Ubuntu Touch devices

UInput

- Requires correct device access permissions
 - The user (or group) that are running the autopilot tests need read/write access to the UInput device (usually /dev/uinput).
- Specific kernel support is required
 - The kernel on the system running the tests must be running a kernel that includes UInput support (as well as have the module loaded).

OSK

- Currently only available on Ubuntu Touch devices
 - At the time of writing this the OSK/Ubuntu Keyboard is only supported/available on the Ubuntu Touch devices. It is possible that it will be available on the desktop in the near future.
- Unable to type ‘special’ keys e.g. Alt
 - This shouldn’t be an issue as applications running on Ubuntu Touch devices will be using the expected patterns of use on these platforms.
- The following methods have limitations or are not implemented:
 - `autopilot.input.Keyboard.press`: Raises `NotImplementedError` if called.
 - `autopilot.input.Keyboard.release`: Raises `NotImplementedError` if called.
 - `autopilot.input.Keyboard.press_and_release`: can only handle single keys/characters. Raises either `ValueError` if passed more than a single character key or `UnsupportedKey` if passed a key that is not supported by the OSK backend (or the current language layout).

Process Control

The `autopilot.process` module provides the `ProcessManager` class to provide a high-level interface for managing applications and windows during testing. Features of the `ProcessManager` allow the user to start and stop applications easily and to query the current state of an application and its windows. It also provides automatic cleanup for apps that have been launched during testing.

Note: `ProcessManager` is not intended for introspecting an application’s object tree, for this see [Launching Applications](#). Also it does not provide a method for interacting with an application’s UI or specific features.

Properties of an application and its windows can be accessed using the classes `Application` and `Window`, which also allows the window instance to be focused and closed.

A list of known applications is defined in `KNOWN_APPS` and these can easily be referenced by name. This list can also be updated using `register_known_application` and `unregister_known_application` for easier use during the test.

To use the `ProcessManager` the static `create` method should be called, which returns an initialised object instance.

A simple example to launch the `gedit` text editor and check it is in focus:

```
from autopilot.process import ProcessManager
from autopilot.testcase import AutopilotTestCase

class ProcessManagerTestCase(AutopilotTestCase):

    def test_launch_app(self):
        pm = ProcessManager.create()
        app_window = pm.start_app_window('Text Editor')
        app_window.set_focus()
        self.assertTrue(app_window.is_focused)
```

Note: `ProcessManager` is only available on environments that use `bamf`, i.e. desktop running Unity 7. There is currently no process manager for any other platform.

Display Information

Autopilot provides the `autopilot.display` module to get information about the displays currently being used. This information can be used in tests to implement gestures or input events that are specific to the current test environment. For example a test could be run on a desktop environment with multiple screens, or on a variety of touch devices that have different screen sizes.

The user must call the static `create` method to get an instance of the `Display` class.

This example shows how to get the size of each available screen, which could be used to calculate coordinates for a swipe or input event (See the `autopilot.input` module for more details about generating input events):

```
from autopilot.display import Display

display = Display.create()
for screen in range(0, display.get_num_screens()):
    width = display.get_screen_width(screen)
    height = display.get_screen_height(screen)
    print('screen {0}: {1}x{2}'.format(screen, width, height))
```

Writing Custom Proxy Classes

By default, autopilot will generate an object for every introspectable item in your application under test. These are generated on the fly, and derive from `ProxyBase`. This gives you the usual methods of selecting other nodes in the object tree, as well as the means to inspect all the properties in that class.

However, sometimes you want to customize the class used to create these objects. The most common reason to want to do this is to provide methods that make it easier to inspect or interact with these objects. Autopilot allows test authors to provide their own custom classes, through a couple of simple steps:

1. First, you must define your own base class, to be used by all custom proxy objects in your test suite. This base class can be empty, but must derive from `ProxyBase`. An example class might look like this:

```
from autopilot.introspection import ProxyBase

class CustomProxyObjectBase(ProxyBase):
    """A base class for all custom proxy objects within this test suite."""
```

For Ubuntu applications using Ubuntu UI Toolkit objects, you should derive your custom proxy object from `UbuntuUIToolkitCustomProxyObjectBase`. This base class is also derived from `ProxyBase` and is used for all Ubuntu UI Toolkit custom proxy objects. So if you are introspecting objects from Ubuntu UI Toolkit then this is the base class to use.

2. Define the classes you want autopilot to use, instead of the default. The simplest method is to give the class the same name as the type you wish to override. For example, if you want to define your own custom class to be used every time autopilot generates an instance of a 'QLabel' object, the class definition would look like this:

```
class QLabel(CustomProxyObjectBase):

    # Add custom methods here...
```

If you wish to implement more specific selection criteria, your class can override the `validate_dbus_object` method, which takes as arguments the dbus path and state. For example:

```
class SpecificQLabel(CustomProxyObjectBase):

    def validate_dbus_object(path, state):
        return (path.endswith('object_we_want') or
                state['some_property'] == 'desired_value')
```

This method should return `True` if the object matches this custom proxy class, and `False` otherwise. If more than one custom proxy class matches an object, a `ValueError` will be raised at runtime.

An example using Ubuntu UI Toolkit which would be used to swipe up a `PageWithBottomEdge` object to reveal it's bottom edge menu could look like this:

```
import ubuntuuitoolkit

class PageWithBottomEdge(ubuntuuitoolkit.UbuntuUIToolkitCustomProxyObjectBase):
    """An emulator class that makes it easy to interact with the bottom edge
    swipe page"""

    def reveal_bottom_edge_page(self):
        """Swipe up from the bottom edge of the Page
        to reveal it's bottom edge menu."""
```

3. Pass the custom proxy base class as an argument to the `launch_test_application` method on your test class. This base class should be the same base class that is used to write all of your custom proxy objects:

```
from autopilot.testcase import AutopilotTestCase

class TestCase(AutopilotTestCase):

    def setUp(self):
        super().setUp()
        self.app = self.launch_test_application(
```

```

'/path/to/the/application',
emulator_base=CustomProxyObjectBase)

```

For applications using objects from Ubuntu UI Toolkit, the `emulator_base` parameter should be:

```
emulator_base=ubuntuitoolkit.UbuntuUIToolkitCustomProxyObjectBase
```

4. You can pass the custom proxy class to methods like `select_single` instead of a string. So, for example, the following is a valid way of selecting the `QLabel` instances in an application:

```

# Get all QLabel instances in the application:
labels = self.app.select_many(QLabel)

```

If you are introspecting an application that already has a custom proxy base class defined, then this class can simply be imported and passed to the appropriate application launcher method. See [launching applications](#) for more details on launching an application for introspection. This will allow you to call all of the public methods of the application's proxy base class directly in your test.

This example will run on desktop and uses the `webbrowser` application to navigate to a url using the base class `go_to_url()` method:

```

from autopilot.testcase import AutopilotTestCase
from webbrowser_app.emulators import browser

class ClickAppTestCase(AutopilotTestCase):

    def test_go_to_url(self):
        app = self.launch_test_application(
            'webbrowser-app',
            emulator_base=browser.Webbrowser)
        # main_window is a property of the Webbrowser class
        app.main_window.go_to_url('http://www.ubuntu.com')

```

Launching Applications

Applications can be launched inside of a testcase using the application launcher methods from the `AutopilotTestCase` class. The exact method required will depend upon the type of application being launched:

- `launch_test_application` is used to launch regular executables
- `launch_upstart_application` is used to launch upstart-based applications
- `launch_click_package` is used to launch applications inside a [click package](#)

This example shows how to launch an installed click application from within a test case:

```

from autopilot.testcase import AutopilotTestCase

class ClickAppTestCase(AutopilotTestCase):

    def test_something(self):
        app_proxy = self.launch_click_package('com.ubuntu.calculator')

```

Outside of testcase classes, the `NormalApplicationLauncher`, `UpstartApplicationLauncher`, and `ClickApplicationLauncher` fixtures can be used, e.g.:

```
from autopilot.application import NormalApplicationLauncher

with NormalApplicationLauncher() as launcher:
    launcher.launch('gedit')
```

or a similar example for an installed click package:

```
from autopilot.application import ClickApplicationLauncher

with ClickApplicationLauncher() as launcher:
    app_proxy = launcher.launch('com.ubuntu.calculator')
```

Within a fixture or a testcase, `self.useFixture` can be used:

```
launcher = self.useFixture(NormalApplicationLauncher())
launcher.launch('gedit', ['--new-window', '/path/to/file'])
```

or for an installed click package:

```
launcher = self.useFixture(ClickApplicationLauncher())
app_proxy = launcher.launch('com.ubuntu.calculator')
```

Additional options can also be specified to set a custom `addDetail` method, a custom proxy base, or a custom dbus bus with which to patch the environment:

```
launcher = self.useFixture(NormalApplicationLauncher(
    case_addDetail=self.addDetail,
    dbus_bus='some_other_bus',
    proxy_base=my_proxy_class,
))
```

Note: You must pass the test case's `addDetail` method to these application launch fixtures if you want application logs to be attached to the test result. This is due to the way fixtures are cleaned up, and is unavoidable.

The main qml file of some click applications can also be launched directly from source. This can be done using the `qmlscene` application directly on the target application's main qml file. This example uses `launch_test_application` method from within a test case:

```
app_proxy = self.launch_test_application('qmlscene', 'application.qml', app_type='qt')
```

However, using this method it will not be possible to return an application specific custom proxy object, see [Writing Custom Proxy Classes](#).

Writing Good Autopilot Tests

This document is an introduction to writing good autopilot tests. This should be treated as additional material on top of all the things you'd normally do to write good code. Put another way: test code follows all the same rules as production code - it must follow the coding standards, and be of a professional quality.

Several points in this document are written with respect to the unity autopilot test suite. This is incidental, and doesn't mean that these points do not apply to other test suites!

Write Expressive Tests

Unit tests are often used as a reference for how your public API should be used. Functional (Autopilot) tests are no different: they can be used to figure out how your application should work from a functional standpoint. However, this only works if your tests are written in a clear, concise, and most importantly expressive style. There are many things you can do to make your tests easier to read:

Pick Good Test Case Class Names

Pick a name that encapsulates all the tests in the class, but is as specific as possible. If necessary, break your tests into several classes, so your class names can be more specific. This is important because when a test fails, the test id is the primary means of identifying the failure. The more descriptive the test id is, the easier it is to find the fault and fix the test.

Pick Good Test Case Method Names

Similar to picking good test case class names, picking good method names makes your test id more descriptive. We recommend writing very long test method names, for example:

```
# bad example:
def test_save_dialog(self):
    # test goes here

# better example:
def test_save_dialog_can_cancel(self):
    # test goes here
```

```
# best example:
def test_save_dialog_cancels_on_escape_key(self):
    # test goes here
```

Write Docstrings

You should write docstrings for your tests. Often the test method is enough to describe what the test does, but an English description is still useful when reading the test code. For example:

```
def test_save_dialog_cancels_on_escape_key(self):
    """The Save dialog box must cancel when the escape key is pressed."""
```

We recommend following [PEP 257](#) when writing all docstrings.

Test One Thing Only

Tests should test one thing, and one thing only. Since we're not writing unit tests, it's fine to have more than one assert statement in a test, but the test should test one feature only. How do you tell if you're testing more than one thing? There's two primary ways:

1. Can you describe the test in a single sentence without using words like 'and', 'also', etc? If not, you should consider splitting your tests into multiple smaller tests.
2. Tests usually follow a simple pattern:
 1. Set up the test environment.
 2. Perform some action.
 3. Test things with assert statements.

If you feel you're repeating steps 'b' and 'c' you're likely testing more than one thing, and should consider splitting your tests up.

Good Example:

```
def test_alt_f4_close_dash(self):
    """Dash must close on alt+F4."""
    self.dash.ensure_visible()
    self.keyboard.press_and_release("Alt+F4")
    self.assertThat(self.dash.visible, Eventually(Equals(False)))
```

This test tests one thing only. Its three lines match perfectly with the typical three stages of a test (see above), and it only tests for things that it's supposed to. Remember that it's fine to assume that other parts of unity work as expected, as long as they're covered by an autopilot test somewhere else - that's why we don't need to verify that the dash really did open when we called `self.dash.ensure_visible()`.

Fail Well

Make sure your tests test what they're supposed to. It's very easy to write a test that passes. It's much more difficult to write a test that only passes when the feature it's testing is working correctly, and fails otherwise. There are two main ways to achieve this:

- Write the test first. This is easy to do if you're trying to fix a bug in Unity. In fact, having a test that's exploitable via an autopilot test will help you fix the bug as well. Once you think you have fixed the bug, make sure the autopilot test you wrote now passed. The general workflow will be:

0. Branch unity trunk.
1. Write autopilot test that reproduces the bug.
2. Commit.
3. Write code that fixes the bug.
4. Verify that the test now passes.
5. Commit. Push. Merge.
6. Celebrate!
 - If you're writing tests for a bug-fix that's already been written but is waiting on tests before it can be merged, the workflow is similar but slightly different:
0. Branch unity trunk.
1. Write autopilot test that reproduces the bug.
2. Commit.
3. Merge code that supposedly fixes the bug.
4. Verify that the test now passes.
5. Commit. Push. Superseed original merge proposal with your branch.
6. Celebrate!

Think about design

Much in the same way you might choose a functional or objective-oriented paradigm for a piece of code, a testsuite can benefit from choosing a good design pattern. One such design pattern is the page object model. The page object model can reduce testcase complexity and allow the testcase to grow and easily adapt to changes within the underlying application. Check out [page_object_guide](#).

Test Length

Tests should be short - as short as possible while maintaining readability. Longer tests are harder to read, harder to understand, and harder to debug. Long tests are often symptomatic of several possible problems:

- Your test requires complicated setup that should be encapsulated in a method or function.
- Your test is actually several tests all jammed into one large test.

Bad Example:

```
def test_panel_title_switching_active_window(self):
    """Tests the title shown in the panel with a maximized application."""
    # Locked Launchers on all monitors
    self.set_unity_option('num_launchers', 0)
    self.set_unity_option('launcher_hide_mode', 0)

    text_win = self.open_new_application_window("Text Editor", maximized=True)

    self.assertTrue(text_win.is_maximized)
    self.assertEqual(self.panel.title, Equals(text_win.title))
    sleep(.25)
```

```
calc_win = self.open_new_application_window("Calculator")
self.assertThat(self.panel.title, Equals(calc_win.application.name))

icon = self.launcher.model.get_icon_by_desktop_id(text_win.application.desktop_
↪file)
launcher = self.launcher.get_launcher_for_monitor(self.panel_monitor)
launcher.click_launcher_icon(icon)

self.assertTrue(text_win.is_focused)
self.assertThat(self.panel.title, Equals(text_win.title))
```

This test can be simplified into the following:

```
def test_panel_title_switching_active_window(self):
    """Tests the title shown in the panel with a maximized application."""
    text_win = self.open_new_application_window("Text Editor", maximized=True)
    self.open_new_application_window("Calculator")

    icon = self.launcher.model.get_icon_by_desktop_id(text_win.application.desktop_
↪file)
    launcher = self.launcher.get_launcher_for_monitor(self.panel_monitor)
    launcher.click_launcher_icon(icon)

    self.assertTrue(text_win.is_focused)
    self.assertThat(self.panel.title, Equals(text_win.title))
```

Here's what we changed:

- Removed the `set_unity_option` lines, as they didn't affect the test results at all.
- Removed assertions that were duplicated from other tests. For example, there's already an autopilot test that ensures that new applications have their title displayed on the panel.

With a bit of refactoring, this test could be even smaller (the launcher proxy classes could have a method to click an icon given a desktop id), but this is now perfectly readable and understandable within a few seconds of reading.

Good docstrings

Test docstrings are used to communicate to other developers what the test is supposed to be testing. Test Docstrings must:

1. Conform to [PEP8](#) and [PEP257](#) guidelines.
2. Avoid words like “should” in favor of stronger words like “must”.
3. Contain a one-line summary of the test.

Additionally, they should:

1. Include the launchpad bug number (if applicable).

Good Example:

```
def test_launcher_switcher_next_keeps_shortcuts(self):
    """Launcher switcher next action must keep shortcuts after they've been shown."""
```

Within the context of the test case, the docstring is able to explain exactly what the test does, without any ambiguity. In contrast, here's a poorer example:

Bad Example:

```
def test_switcher_all_mode_shows_all_apps(self):
    """Test switcher 'show_all' mode shows apps from all workspaces."""
```

The docstring explains what the desired outcome is, but without how we're testing it. This style of sentence assumes test success, which is not what we want! A better version of this code might look like this:

```
def test_switcher_all_mode_shows_all_apps(self):
    """Switcher 'show_all' mode must show apps from all workspaces."""
```

The difference between these two are subtle, but important.

Test Readability

The most important attribute for a test is that it is correct - it must test what's it's supposed to test. The second most important attribute is that it is readable. Tests should be able to be examined by themselves by someone other than the test author without any undue hardship. There are several things you can do to improve test readability:

1. Don't abuse the `setUp()` method. It's tempting to put code that's common to every test in a class into the `setUp` method, but it leads to tests that are not readable by themselves. For example, this test uses the `setUp` method to start the launcher switcher, and `tearDown` to cancel it:

Bad Example:

```
def test_launcher_switcher_next(self):
    """Moving to the next launcher item while switcher is activated must work.
    ↪ """
    self.launcher_instance.switcher_next()
    self.assertThat(self.launcher.key_nav_selection,
    ↪Eventually(GreaterThan(0)))
```

This leads to a shorter test (which we've already said is a good thing), but the test itself is incomplete. Without scrolling up to the `setUp` and `tearDown` methods, it's hard to tell how the launcher switcher is started. The situation gets even worse when test classes derive from each other, since the code that starts the launcher switcher may not even be in the same class!

A much better solution in this example is to initiate the switcher explicitly, and use `addCleanup()` to cancel it when the test ends, like this:

Good Example:

```
def test_launcher_switcher_next(self):
    """Moving to the next launcher item while switcher is activated must work.
    ↪ """
    self.launcher_instance.switcher_start()
    self.addCleanup(self.launcher_instance.switcher_cancel)

    self.launcher_instance.switcher_next()
    self.assertThat(self.launcher.key_nav_selection,
    ↪Eventually(GreaterThan(0)))
```

The code is longer, but it's still very readable. It also follows the setup/action/test convention discussed above.

Appropriate uses of the `setUp()` method include:

- Initialising test class member variables.

- Setting unity options that are required for the test. For example, many of the switcher autopilot tests set a unity option to prevent the switcher going into details mode after a timeout. This isn't part of the test, but makes the test easier to write.
 - Setting unity log levels. The unity log is captured after each test. Some tests may adjust the verbosity of different parts of the Unity logging tree.
2. Put common setup code into well-named methods. If the “setup” phase of a test is more than a few lines long, it makes sense to put this code into it's own method. Pay particular attention to the name of the method you use. You need to make sure that the method name is explicit enough to keep the test readable. Here's an example of a test that doesn't do this:

Bad Example:

```
def test_showdesktop_hides_apps(self):
    """Show Desktop keyboard shortcut must hide applications."""
    self.start_app('Character Map', locale='C')
    self.start_app('Calculator', locale='C')
    self.start_app('Text Editor', locale='C')

    # show desktop, verify all windows are hidden:
    self.keybinding("window/show_desktop")
    self.addCleanup(self.keybinding, "window/show_desktop")

    open_wins = self.bamf.get_open_windows()
    for win in open_wins:
        self.assertTrue(win.is_hidden)
```

In contrast, we can refactor the test to look a lot nicer:

Good Example:

```
def test_showdesktop_hides_apps(self):
    """Show Desktop keyboard shortcut must hide applications."""
    self.launch_test_apps()

    # show desktop, verify all windows are hidden:
    self.keybinding("window/show_desktop")
    self.addCleanup(self.keybinding, "window/show_desktop")

    open_wins = self.bamf.get_open_windows()
    for win in open_wins:
        self.assertTrue(win.is_hidden)
```

The test is now shorter, and the `launch_test_apps` method can be re-used elsewhere. Importantly - even though I've hidden the implementation of the `launch_test_apps` method, the test still makes sense.

3. Hide complicated assertions behind custom `assertXXX` methods or custom matchers. If you find that you frequently need to use a complicated assertion pattern, it may make sense to either:
- Write a custom matcher. As long as you follow the protocol laid down by the `testtools.matchers.Matcher` class, you can use a hand-written `Matcher` just like you would use an ordinary one. Matchers should be written in the `autopilot.matchers` module if they're likely to be reusable outside of a single test, or as local classes if they're specific to one test.
 - Write custom assertion methods. For example:

```
def test_multi_key_copyright(self):
    """Pressing the sequences 'Multi_key' + 'c' + 'o' must produce '@'
    """
```

```
self.dash.reveal_application_lens()
self.keyboard.press_and_release('Multi_key')
self.keyboard.type("oc")
self.assertSearchText("@")
```

This test uses a custom method named `assertSearchText` that hides the complexity involved in getting the dash search text and comparing it to the given parameter.

Prefer `wait_for` and `Eventually` to sleep

Early autopilot tests relied on extensive use of the python `sleep` call to halt tests long enough for unity to change its state before the test continued. Previously, an autopilot test might have looked like this:

Bad Example:

```
def test_alt_f4_close_dash(self):
    """Dash must close on alt+F4."""
    self.dash.ensure_visible()
    sleep(2)
    self.keyboard.press_and_release("Alt+F4")
    sleep(2)
    self.assertThat(self.dash.visible, Equals(False))
```

This test uses two `sleep` calls. The first makes sure the dash has had time to open before the test continues, and the second makes sure that the dash has had time to respond to our key presses before we start testing things.

There are several issues with this approach:

1. On slow machines (like a jenkins instance running on a virtual machine), we may not be sleeping long enough. This can lead to tests failing on jenkins that pass on developers machines.
2. On fast machines, we may be sleeping too long. This won't cause the test to fail, but it does make running the test suite longer than it has to be.

There are two solutions to this problem:

In Tests

Tests should use the `Eventually` matcher. This can be imported as follows:

```
from autopilot.matchers import Eventually
```

The `Eventually` matcher works on all attributes in a proxy class that derives from `UnityIntrospectableObject` (at the time of writing that is almost all the autopilot unity proxy classes).

The `Eventually` matcher takes a single argument, which is another testtools matcher instance. For example, the bad assertion from the example above could be rewritten like so:

```
self.assertThat(self.dash.visible, Eventually(Equals(False)))
```

Since we can use any testtools matcher, we can also write code like this:

```
self.assertThat(self.launcher.key_nav_selection, Eventually(GreaterThan(prev_icon)))
```

Note that you can pass any object that follows the testtools matcher protocol (so you can write your own matchers, if you like).

In Proxy Classes

Proxy classes are not test cases, and do not have access to the `self.assertThat` method. However, we want proxy class methods to block until unity has had time to process the commands given. For example, the `ensure_visible` method on the Dash controller should block until the dash really is visible.

To achieve this goal, all attributes on unity proxy classes have been patched with a `wait_for` method that takes a testtools matcher (just like `Eventually` - in fact, the `Eventually` matcher just calls `wait_for` under the hood). For example, previously the `ensure_visible` method on the Dash controller might have looked like this:

Bad Example:

```
def ensure_visible(self):
    """Ensures the dash is visible."""
    if not self.visible:
        self.toggle_reveal()
        sleep(2)
```

In this example we're assuming that two seconds is long enough for the dash to open. To use the `wait_for` feature, the code looks like this:

Good Example:

```
def ensure_visible(self):
    """Ensures the dash is visible."""
    if not self.visible:
        self.toggle_reveal()
        self.visible.wait_for(True)
```

Note that `wait_for` assumes you want to use the `Equals` matcher if you don't specify one. Here's another example where we're using it with a testtools matcher:

```
key_nav_selection.wait_for(NotEquals(old_selection))
```

Scenarios

Autopilot uses the `python-testscenarios` package to run a test multiple times in different scenarios. A good example of scenarios in use is the launcher keyboard navigation tests: each test is run once with the launcher hide mode set to 'always show launcher', and again with it set to 'autohide launcher'. This allows test authors to write their test once and have it execute in multiple environments.

In order to use test scenarios, the test author must create a list of scenarios and assign them to the test case's `scenarios` class attribute. The autopilot ibus test case classes use scenarios in a very simple fashion:

Good Example:

```
class IBusTestsPinyin(IBusTests):
    """Tests for the Pinyin(Chinese) input engine."""

    scenarios = [
        ('basic', {'input': 'abc1', 'result': u'\u963f\u5e03\u4ece'}),
        ('photo', {'input': 'zhaopian ', 'result': u'\u7167\u7247'}),
        ('internet', {'input': 'hulianwang ', 'result': u'\u4e92\u8054\u7f51'}),
        ('disk', {'input': 'cipan ', 'result': u'\u78c1\u76d8'}),
        ('disk_management', {'input': 'cipan guanli ', 'result': u
    ↪ '\u78c1\u76d8\u7ba1\u7406'}),
    ]
```



```
def test_simple_input_dash(self):
    self.dash.ensure_visible()
    self.addCleanup(self.dash.ensure_hidden)
    self.activate_ibus(self.dash.searchbar)
    self.keyboard.type(self.input)
    self.deactivate_ibus(self.dash.searchbar)
    self.assertThat(self.dash.search_string, Eventually(Equals(self.result)))
```

This is a simplified version of the IBus tests. In this case, the `test_simple_input_dash` test will be called 5 times. Each time, the `self.input` and `self.result` attribute will be set to the values in the scenario list. The first part of the scenario tuple is the scenario name - this is appended to the test id, and can be whatever you want.

Important: It is important to notice that the test does not change its behavior depending on the scenario it is run under. Exactly the same steps are taken - the only difference in this case is what gets typed on the keyboard, and what result is expected.

Scenarios are applied before the test's `setUp` or `tearDown` methods are called, so it's safe (and indeed encouraged) to set up the test environment based on these attributes. For example, you may wish to set certain unity options for the duration of the test based on a scenario parameter.

Multiplying Scenarios

Scenarios are very helpful, but only represent a single-dimension of parameters. For example, consider the launcher keyboard navigation tests. We may want several different scenarios to come into play:

1. A scenario that controls whether the launcher is set to 'autohide' or 'always visible'.
2. A scenario that controls which monitor the test is run on (in case we have multiple monitors configured).

We can generate two separate scenario lists to represent these two scenario axis, and then produce the dot-product of the two lists like this:

```
from autopilot.tests import multiply_scenarios

class LauncherKeynavTests(AutopilotTestCase):

    hide_mode_scenarios = [
        ('autohide', {'hide_mode': 1}),
        ('neverhide', {'hide_mode': 0}),
    ]

    monitor_scenarios = [
        ('monitor_0', {'launcher_monitor': 0}),
        ('monitor_1', {'launcher_monitor': 1}),
    ]

    scenarios = multiply_scenarios(hide_mode_scenarios, monitor_scenarios)
```

(please ignore the fact that we're assuming that we always have two monitors!)

In the test classes `setUp` method, we can then set the appropriate unity option and make sure we're using the correct launcher:

```
def setUp(self):
    self.set_unity_option('launcher_hide_mode', self.hide_mode)
    self.launcher_instance = self.launcher.get_launcher_for_monitor(self.launcher_
↪monitor)
```

Which allows us to write tests that work automatically in all the scenarios:

```
def test_keynav_initiates(self):
    """Launcher must start keyboard navigation mode."""
    self.launcher.keynav_start()
    self.assertThat(self.launcher.kaynav_mode, Eventually(Equals(True)))
```

This works fine. So far we've not done anything to cause undue pain.... until we decide that we want to extend the scenarios with an additional axis:

```
from autopilot.tests import multiply_scenarios

class LauncherKeynavTests(AutopilotTestCase):

    hide_mode_scenarios = [
        ('autohide', {'hide_mode': 1}),
        ('neverhide', {'hide_mode': 0}),
    ]

    monitor_scenarios = [
        ('monitor_0', {'launcher_monitor': 0}),
        ('monitor_1', {'launcher_monitor': 1}),
    ]

    launcher_monitor_scenarios = [
        ('launcher on all monitors', {'monitor_mode': 0}),
        ('launcher on primary monitor only', {'monitor_mode': 1}),
    ]

    scenarios = multiply_scenarios(hide_mode_scenarios, monitor_scenarios, launcher_
↪monitor_scenarios)
```

Now we have a problem: Some of the generated scenarios won't make any sense. For example, one such scenario will be (autohide, monitor_1, launcher on primary monitor only). If monitor 0 is the primary monitor, this will leave us running launcher tests on a monitor that doesn't contain a launcher!

There are two ways to get around this problem, and they both lead to terrible tests:

1. Detect these situations and skip the test. This is bad for several reasons - first, skipped tests should be viewed with the same level of suspicion as commented out code. Test skips should only be used in exceptional circumstances. A test skip in the test results is just as serious as a test failure.
2. Detect the situation in the test, and run different code using an if statement. For example, we might decode to do this:

```
def test_something(self):
    # ... setup code here ...
    if self.monitor_mode == 1 and self.launcher_monitor == 1:
        # test something else
    else:
        # test the original thing.
```

As a general rule, tests shouldn't have assert statements inside an if statement unless there's a very good reason for doing so.

Scenarios can be useful, but we must be careful not to abuse them. It is far better to spend more time typing and end up with clear, readable tests than it is to end up with fewer, less readable tests. Like all code, tests are read far more often than they're written.

Do Not Depend on Object Ordering

Calls such as `select_many` return several objects at once. These objects are explicitly unordered, and test authors must take care not to make assumptions about their order.

Bad Example:

```
buttons = self.select_many('Button')
save_button = buttons[0]
print_button = buttons[1]
```

This code may work initially, but there's absolutely no guarantee that the order of objects won't change in the future. A better approach is to select the individual components you need:

Good Example:

```
save_button = self.select_single('Button', objectName='btnSave')
print_button = self.select_single('Button', objectName='btnPrint')
```

This code will continue to work in the future.

Running Autopilot

Autopilot test suites can be run with any python test runner (for example, the built-in testtools runner). However, several autopilot features are only available if you use the autopilot runner.

List Tests

Autopilot can list all tests found within a particular module:

```
$ autopilot3 list <modulename>
```

where *<modulename>* is the base name of the module you want to look at. The module must either be in the current working directory, or be importable by python. For example, to list the tests inside autopilot itself, you can run:

```
$ autopilot3 list autopilot
autopilot.tests.test_ap_apps.GtkTests.test_can_launch_qt_app
autopilot.tests.test_ap_apps.QtTests.test_can_launch_qt_app
autopilot.tests.test_application_mixin.ApplicationSupportTests.test_can_create
autopilot.tests.test_application_mixin.ApplicationSupportTests.test_launch_raises_
↳ValueError_on_unknown_kwargs
autopilot.tests.test_application_mixin.ApplicationSupportTests.test_launch_raises_
↳ValueError_on_unknown_kwargs_with_known
autopilot.tests.test_application_mixin.ApplicationSupportTests.test_launch_with_
↳bad_types_raises_typeerror
autopilot.tests.test_application_registration.ApplicationRegistrationTests.test_
↳can_register_new_application
autopilot.tests.test_application_registration.ApplicationRegistrationTests.test_
↳can_unregister_application
autopilot.tests.test_application_registration.ApplicationRegistrationTests.test_
↳registering_app_twice_raises_KeyError
autopilot.tests.test_application_registration.ApplicationRegistrationTests.test_
↳unregistering_unknown_application_raises_KeyError
...
81 total tests.
```

Some results have been omitted for clarity.

The list command takes only one option:

-ro, --run-order Display tests in the order in which they will be run, rather than alphabetical order (which is the default).

Run Tests

Running autopilot tests is very similar to listing tests:

```
$ autopilot3 run <module name>
```

However, the run command has many more options to customize the run behavior:

-h, --help show this help message and exit

-o OUTPUT, --output OUTPUT Write test result report to file. Defaults to stdout. If given a directory instead of a file will write to a file in that directory named: <hostname>_<dd.mm.yyy_HHMMSS>.log

-f FORMAT, --format FORMAT Specify desired output format. Default is “text”. Other option is ‘xml’ to produce junit xml format.

-r, --record Record failing tests. Required ‘recordmydesktop’ app to be installed. Videos are stored in /tmp/autopilot.

-rd PATH, --record-directory PATH Directory to put recorded tests (only if -r) specified.

-v, --verbose If set, autopilot will output test log data to stderr during a test run.

Common use cases

1. Run autopilot and save the test log:

```
$ autopilot3 run -o . <module name>
```

Autopilot will create a text log file named <hostname>_<dd.mm.yyy_HHMMSS>.log with the contents of the test log.

2. Run autopilot and record failing tests:

```
$ autopilot3 run -r --rd . <module name>
```

Videos are recorded as *ogg-vorbis* files, with an .ogv extension. They will be named with the test id that failed. All videos will be placed in the directory specified by the `-rd` option - in this case the current directory. If this option is omitted, videos will be placed in `/tmp/autopilot/`.

3. Save the test log as jUnitXml format:

```
$ autopilot3 run -o results.xml -f xml <module name>
```

The file ‘results.xml’ will be created when all the tests have completed, and will be in the jUnitXml file format. This is useful when running the autopilot tests within a Jenkins environment.

Launching an Application to Introspect

In order to be able to introspect an application, it must first be launched with introspection enabled. Autopilot provides the **launch** command to enable this:

```
$ autopilot3 launch <application> <app_parameters>
```

The *<application>* parameter could be the full path to the application, or the name of an application located somewhere on `$PATH`. *<app_parameter>* is passed on to the application being launched.

A simple Gtk example to launch gedit:

```
$ autopilot3 launch gedit
```

A Qt example which passes on parameters to the application being launched:

```
$ autopilot3 launch qmlscene my_app.qml
```

Autopilot launch attempts to detect if you are launching either a Gtk or Qt application so that it can enable the correct libraries. If it is unable to determine this you will need to specify the type of application it is by using the `-i` argument. This allows “Gtk” or “Qt” frameworks to be specified when launching the application. The default value (“Auto”) will try to detect which interface to load automatically.

A typical error in this situation will be “Error: Could not determine introspection type to use for application”. In which case the `-i` option should be specified with the correct application framework type to fix the problem:

```
$ autopilot3 launch -i Qt address-book-app
```

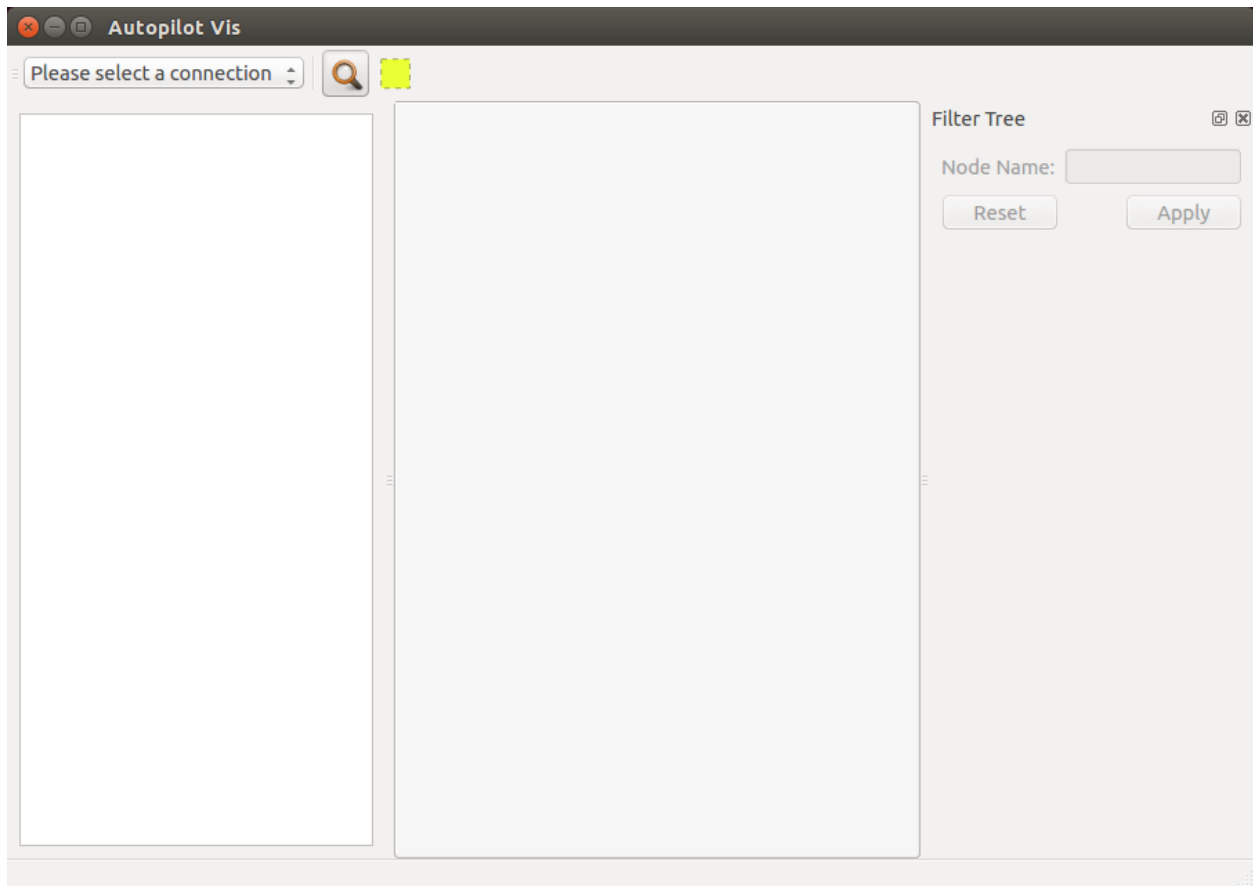
Once an application has launched with introspection enabled, it will be possible to launch autopilot vis and view the introspection tree, see: [Visualise Introspection Tree](#).

Visualise Introspection Tree

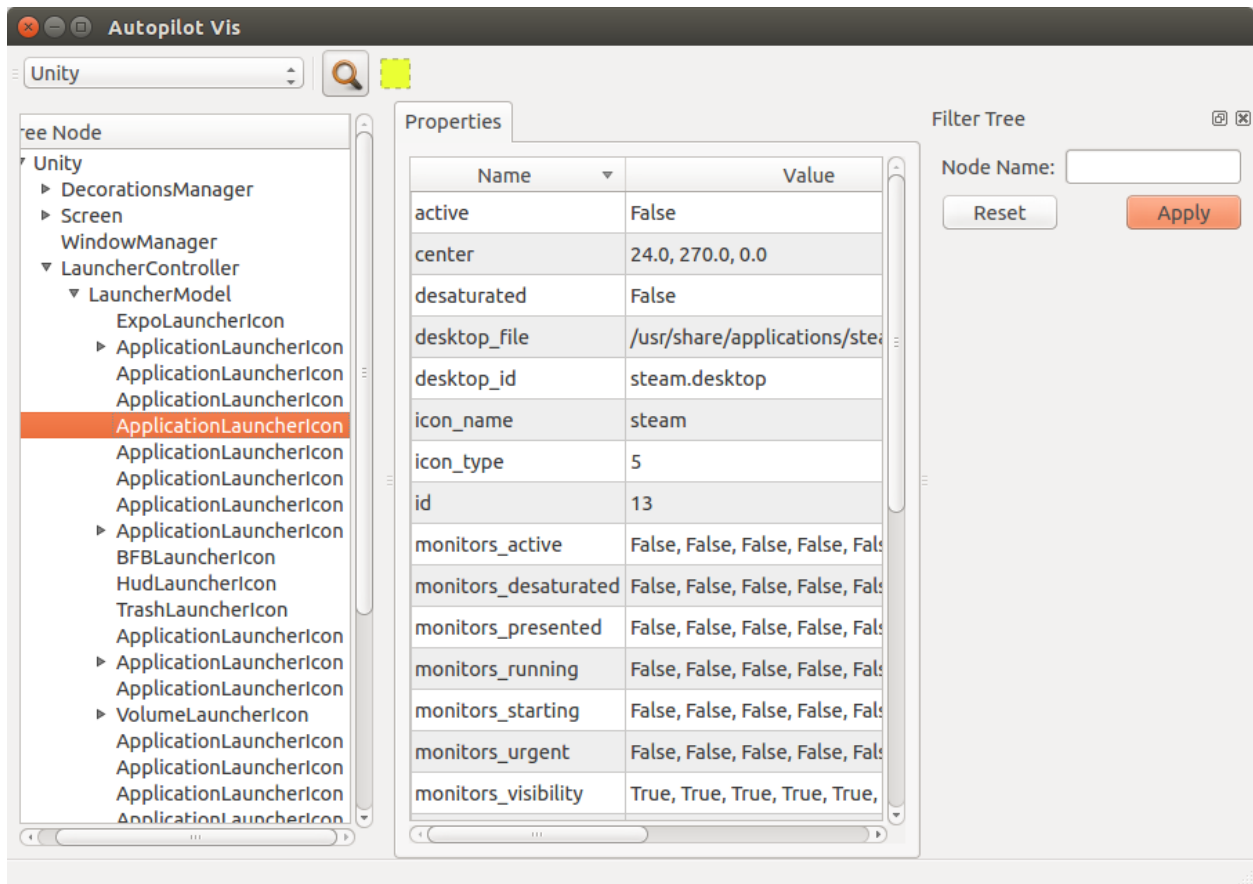
A very common thing to want to do while writing autopilot tests is see the structure of the application being tested. To support this, autopilot includes a simple application to help visualize the introspection tree. To start it, make sure the application you wish to test is running (see: [Launching an Application to Introspect](#)), and then run:

```
$ autopilot3 vis
```

The result should be a window similar to below:



Selecting a connection from the drop-down box allows you to inspect different autopilot-supporting applications. If Unity is running, the Unity connection should always be present. If other applications have been started with the autopilot support enabled, they should appear in this list as well. Once a connection is selected, the introspection tree is rendered in the left-hand pane, and the details of each object appear in the right-hand pane.



Autopilot vis also has the ability to search the object tree for nodes that match a given name (such as “LauncherController”, for example), and draw a transparent overlay over a widget if it contains position information. These tools, when combined can make finding certain parts of an application introspection tree much easier.

Installing Autopilot

Contents

- *Installing Autopilot*
 - *Ubuntu*
 - *Other Linux's*

Autopilot is in continuous development, and the best way to get the latest version of autopilot is to run the latest Ubuntu development image. The autopilot developers traditionally support the Ubuntu release immediately prior to the development release via the autopilot PPA.

Ubuntu

I am running the latest development image!

In that case you can install autopilot directly from the repository and know you are getting the latest release. Check out the packages below.

I am running a stable version of Ubuntu!

You may install the version of autopilot in the archive directly, however it will not be up to date. Instead, you should add the latest autopilot ppa to your system (as of this writing, that is autopilot 1.5).

To add the PPA to your system, run the following command:

```
sudo add-apt-repository ppa:autopilot/1.5 && sudo apt-get update
```

Once the PPA has been added to your system, you should be able to install the autopilot packages below.

Which packages should I install?

Are you working on ubuntu touch applications? The `autopilot-touch` metapackage is for you:

```
sudo apt-get install autopilot-touch
```

If you are sticking with gtk desktop applications, install the `autopilot-desktop` metapackage instead:

```
sudo apt-get install autopilot-desktop
```

Feel free to install both metapackages to ensure you have support for all autopilot tests.

Other Linux's

You may have to download the source code, and either run from source, or build the packages locally. Your best bet is to ask in the autopilot IRC channel ([Q. Where can I get help / support?](#)).

CHAPTER 9

Autopilot API Documentation

`autopilot` - Global stuff

`autopilot.application` - Autopilot Application Launchers

`autopilot.display` - Get information about the current display(s)

`autopilot.emulators` - Backwards compatibility for autopilot v1.2

`autopilot.exceptions` - Autopilot Exceptions

`autopilot.gestures` - Gestural and multi-touch support

`autopilot.input` - Generate keyboard, mouse, and touch input events

`autopilot.introspection` - Retrieve proxy objects

`autopilot.introspection.types` - Introspection Type details

`autopilot.matchers` - Custom matchers for test assertions

`autopilot.platform` - Functions for platform detection

`autopilot.process` - Process Control

`autopilot.testcase` - Base class for all Autopilot Test Cases

Frequently Asked Questions

Contents

- *Frequently Asked Questions*
 - *Autopilot: The Project*
 - * *Q. Where can I get help / support?*
 - * *Q. Which version of autopilot should I install?*
 - * *Q. Should I write my tests in python2 or python3?*
 - * *Q. Should I convert my existing tests to python3?*
 - * *Q. Where can I report a bug?*
 - * *Q. What type of applications can autopilot test?*
 - *Autopilot Tests*
 - * *Q. Autopilot tests often include multiple assertions. Isn't this bad practise?*
 - * *Q. How do I write a test that uses either a Mouse or a Touch device interchangeably?*
 - * *Q. How do I use the Onscreen Keyboard (OSK) to input text in my test?*
 - *Autopilot Tests and Launching Applications*
 - * *Q. How do I launch a Click application from within a test so I can introspect it?*
 - * *Q. How do I access an already running application so that I can test/introspect it?*
 - *Autopilot Qt & Gtk Support*
 - * *Q. How do I launch my application so that I can explore it with the vis tool?*
 - * *Q. What is the impact on memory of adding objectNames to QML items?*

Autopilot: The Project

Q. Where can I get help / support?

The developers hang out in the #ubuntu-autopilot IRC channel on irc.freenode.net.

Q. Which version of autopilot should I install?

Ideally you should adopt and utilize the latest version of autopilot. If your testcase requires you to utilize an older version of autopilot for reasons other than *Porting Autopilot Tests*, please [file a bug](#) and let the development team know about your issue.

Q. Should I write my tests in python2 or python3?

As Autopilot fully supports python3 (see *Python 3*), you should seek to use python3 for new tests. Before making a decision, you should also ensure any 3rd party modules your test may depend on also support python3.

Q: Should I convert my existing tests to python3?

See above. In a word, yes. Converting python2 to python3 (see *Python 3*) is generally straightforward and converting a testcase is likely much easier than a full python application. You can also consider retaining python2 compatibility upon conversion.

Q. Where can I report a bug?

Autopilot is hosted on launchpad - bugs can be reported on the [launchpad bug page for autopilot](#) (this requires a launchpad account).

Q. What type of applications can autopilot test?

Autopilot works with severall different types of applications, including:

- The Unity desktop shell.
- Gtk 2 & 3 applications.
- Qt4, Qt5, and Qml applications.

Autopilot is designed to work across all the form factors Ubuntu runs on, including the phone and tablet.

Autopilot Tests

Q. Autopilot tests often include multiple assertions. Isn't this bad practise?

Maybe. But probably not.

Unit tests should test a single unit of code, and ideally be written such that they can fail in exactly a single way. Therefore, unit tests should have a single assertion that determines whether the test passes or fails.

However, autopilot tests are not unit tests, they are functional tests. Functional test suites tests features, not units of code, so there's several very good reasons to have more than assertion in a single test:

- Some features require several assertions to prove that the feature is working correctly. For example, you may wish to verify that the 'Save' dialog box opens correctly, using the following code:

```
self.assertThat(save_win.title, Eventually(Equals("Save Document")))
self.assertThat(save_win.visible, Equals(True))
self.assertThat(save_win.has_focus, Equals(True))
```

- Some tests need to wait for the application to respond to user input before the test continues. The easiest way to do this is to use the `Eventually` matcher in the middle of your interaction with the application. For example, if testing the `Firefox` browsers ability to print a certain web comic, we might produce a test that looks similar to this:

```
def test_firefox_can_print_xkcd(self):
    """Firefox must be able to print xkcd.com."""
    # Put keyboard focus in URL bar:
    self.keyboard.press_and_release('Ctrl+l')
    self.keyboard.type('http://xkcd.com')
    self.keyboard.press_and_release('Enter')
    # wait for page to load:
    self.assertThat(self.app.loading, Eventually(Equals(False)))
    # open print dialog:
    self.keyboard.press_and_release('Ctrl+p')
    # wait for dialog to open:
    self.assertThat(self.app.print_dialog.open, Eventually(Equals(True)))
    self.keyboard.press_and_release('Enter')
    # ensure something was sent to our faked printer:
    self.assertThat(self.fake_printer.documents_printed, Equals(1))
```

In general, autopilot tests are more relaxed about the 'one assertion per test' rule. However, care should still be taken to produce tests that are as small and understandable as possible.

Q. How do I write a test that uses either a Mouse or a Touch device interchangeably?

The `autopilot.input.Pointer` class is a simple wrapper that unifies some of the differences between the `Touch` and `Mouse` classes. To use it, pass in the device you want to use under the hood, like so:

```
pointer1 = Pointer(Touch.create())
pointer2 = Pointer(Mouse.create())
# pointer1 and pointer2 now have identical APIs
```

Combined with test scenarios, this can be used to write tests that are run twice - once with a mouse device and once with a touch device:

```
from autopilot.input import Mouse, Touch, Pointer
from autopilot.testcase import AutopilotTestCase

class TestCase(AutopilotTestCase):

    scenarios = [
        ('with mouse', dict(pointer=Pointer(Mouse.create()))),
        ('with touch', dict(pointer=Pointer(Touch.create()))),
    ]

    def test_something(self):
```

```
"""Click the pointer at 100,100."""
self.pointer.move(100,100)
self.pointer.click()
```

If you only want to use the mouse on certain platforms, use the `autopilot.platform` module to determine the current platform at runtime.

Q. How do I use the Onscreen Keyboard (OSK) to input text in my test?

The OSK is an backend option for the `autopilot.input.Keyboard.create` method (see this [Advanced Autopilot](#) section for details regarding backend selection.)

Unlike the other backends (X11, UInput) the OSK has a GUI presence and thus can be displayed on the screen.

The `autopilot.input.Keyboard` class provides a context manager that handles any cleanup required when dealing with the input backends.

For example in the instance when the backend is the OSK, when leaving the scope of the context manager the OSK will be dismissed with a swipe:

```
from autopilot.input import Keyboard

text_area = self._launch_test_input_area()
keyboard = Keyboard.create('OSK')
with keyboard.focused_type(text_area) as kb:
    kb.type("Hello World.")
    self.assertThat(text_area.text, Equals("Hello World"))
# At this point now the OSK has been swiped away.
self.assertThat()
```

Autopilot Tests and Launching Applications

Q. How do I launch a Click application from within a test so I can introspect it?

Launching a Click application is similar to launching a traditional application and is as easy as using `launch_click_package`:

```
app_proxy = self.launch_click_package(
    "com.ubuntu.dropping-letters"
)
```

Q. How do I access an already running application so that I can test/introspect it?

In instances where it's impossible to launch the application-under-test from within the testsuite use `get_proxy_object_for_existing_process` to get a proxy object for the running application. In all other cases the recommended way to launch and retrieve a proxy object for an application is by calling either `launch_test_application` or `launch_click_package`

For example, to access a long running process that is running before your test starts:

```
application_pid = get_long_running_processes_pid()
app_proxy = get_proxy_object_for_existing_process(pid=application_pid)
```

Autopilot Qt & Gtk Support

Q. How do I launch my application so that I can explore it with the vis tool?

Autopilot can launch applications with Autopilot support enabled allowing you to explore and introspect the application using the *vis tool*

For instance launching gedit is as easy as:

```
$ autopilot3 launch gedit
```

Autopilot launch attempts to detect if you are launching either a Gtk or Qt application so that it can enable the correct libraries. If it is unable to determine this you will need to specify the type of application it is by using the `-i` argument.

For example, in our previous example Autopilot was able to automatically determine that gedit is a Gtk application and thus no further arguments were required.

If we want to use the vis tool to introspect something like the *testapp.py script* from an earlier tutorial we will need to inform autopilot that it is a Qt application so that it can enable the correct support:

```
$ autopilot3 launch -i Qt testapp.py
```

Now that it has been launched with Autopilot support we can introspect and explore our application using the *vis tool*.

Q. What is the impact on memory of adding objectNames to QML items?

The `objectName` is a `QString` property of `QObject` which defaults to an empty `QString`. `QString` is UTF-16 representation and because it uses some general purpose optimisations it usually allocates twice the space it needs to be able to grow fast. It also uses implicit sharing with copy-on-write and other similar tricks to increase performance again. These properties make the used memory not straightforward to predict. For example, copying an object with an `objectName`, shares the memory between both as long as they are not changed.

When measuring memory consumption, things like memory alignment come into play. Due to the fact that QML is interpreted by a JavaScript engine, we are working in levels where lots of abstraction layers are in between the code and the hardware and we have no chance to exactly measure consumption of a single `objectName` property. Therefore the taken approach is to measure lots of items and calculate the average consumption.

Table 10.1: Measurement of memory consumption of 10000 Items

Without <code>objectName</code>	With unique <code>objectName</code>	With same <code>objectName</code>
65292 kB	66628 kB	66480 kB

=> With 10000 different `objectNames` 1336 kB of memory are consumed which is around 127 Bytes per Item.

Indeed, this is more than only the string. Some of the memory is certainly lost due to memory alignment where certain areas are just not perfectly filled in but left empty. However, certainly not all of the overhead can be blamed on that. Additional memory is used by the `QObject` meta object information that is needed to do signal/slot connections. Also, QML does some optimisations: It does not connect signals/slots when not needed. So the fact that the object name is set could trigger some more connections.

Even if more than the actual string size is used and `QString` uses a large representation, this is very little compared to the rest. A `qmlscene` with just the item is 27MB. One full screen image in the Nexus 10 tablet can easily consume around 30MB of memory. So `objectNames` are definitely not the first places where to search for optimisations.

Writing the test code snippets, one interesting thing came up frequently: Just modifying the code around to set the `objectName` often influences the results more than the actual string. For example, having a javascript function that assigns the `objectName` definitely uses much more memory than the `objectName` itself. Unless it makes sense from

a performance point of view (frequently changing bindings can be slow), `objectNames` should be added by directly binding the value to the property instead using helper code to assign it.

Conclusion: If an `objectName` is needed for testing, this is definitely worth it. `objectName`'s should obviously not be added when not needed. When adding them, the [general QML guidelines for performance](#) should be followed.

Porting Autopilot Tests

This document contains hints as to what is required to port a test suite from any version of autopilot to any newer version.

Contents

- *Porting Autopilot Tests*
 - *A note on Versions*
 - *Porting to Autopilot v1.4.x*
 - * *Gtk Tests and Boolean Parameters*
 - * *select_single Changes*
 - * *DBus backends and DBusIntrospectionObject changes*
 - * *Python 3*
 - *Porting to Autopilot v1.3.x*
 - * *QtIntrospectionTestMixin and GtkIntrospectionTestMixin no longer exist*
 - * *autopilot.emulators namespace has been deprecated*

A note on Versions

Autopilot releases are reasonably tightly coupled with Ubuntu releases. However, the autopilot authors maintain separate version numbers, with the aim of separating the autopilot release cadence from the Ubuntu platform release cadence.

Autopilot versions earlier than 1.2 were not publicly announced, and were only used within Canonical. For that reason, this document assumes that version 1.2 is the lowest version of autopilot present “*in the wild*”.

Porting to Autopilot v1.4.x

The 1.4 release contains several changes that required a break in the DBus wire protocol between autopilot and the applications under test. Most of these changes require no change to test code.

Gtk Tests and Boolean Parameters

Version 1.3 of the autopilot-gtk backend contained a [bug](#) that caused all Boolean properties to be exported as integers instead of boolean values. This in turn meant that test code would fail to return the correct objects when using selection criteria such as:

```
visible_buttons = app.select_many("GtkPushButton", visible=True)
```

and instead had to write something like this:

```
visible_buttons = app.select_many("GtkPushButton", visible=1)
```

This bug has now been fixed, and using the integer selection will fail.

select_single Changes

The `select_single` method used to return `None` in the case where no object was found that matched the search criteria. This led to rather awkward code in places where the object you are searching for is being created dynamically:

```
for i in range(10):
    my_obj = self.app.select_single("MyObject")
    if my_obj is not None:
        break
    time.sleep(1)
else:
    self.fail("Object 'MyObject' was not found within 10 seconds.")
```

This makes the authors intent harder to discern. To improve this situation, two changes have been made:

1. `select_single` raises a `StateNotFoundError` exception if the search terms returned no values, rather than returning `None`.
2. If the object being searched for is likely to not exist, there is a new method: `wait_select_single` will try to retrieve an object for 10 seconds. If the object does not exist after that timeout, a `StateNotFoundError` exception is raised. This means that the above code example should now be written as:

```
my_obj = self.app.wait_select_single("MyObject")
```

DBus backends and DBusIntrospectionObject changes

Due to a change in how `DBusIntrospectionObject` objects store their DBus backend a couple of classmethods have now become instance methods.

These affected methods are:

- `get_all_instances`
- `get_root_instance`
- `get_state_by_path`

For example, if your old code is something along the lines of:

```
all_keys = KeyCustomProxy.get_all_instances()
```

You will instead need to have something like this instead:

```
all_keys = app_proxy.select_many(KeyCustomProxy)
```

Python 3

Starting from version 1.4, autopilot supports python 3 as well as python 2. Test authors can choose to target either version of python.

Porting to Autopilot v1.3.x

The 1.3 release included many API breaking changes. Earlier versions of autopilot made several assumptions about where tests would be run, that turned out not to be correct. Autopilot 1.3 brought several much-needed features, including:

- A system for building pluggable implementations for several core components. This system is used in several areas:
- The input stack can now generate events using either the X11 client libraries, or the UInput kernel driver. This is necessary for devices that do not use X11.
- The display stack can now report display information for systems that use both X11 and the mir display server.
- The process stack can now report details regarding running processes & their windows on both Desktop, tablet, and phone platforms.
- A large code cleanup and reorganisation. In particular, lots of code that came from the Unity 3D codebase has been removed if it was deemed to not be useful to the majority of test authors. This code cleanup includes a flattening of the autopilot namespace. Previously, many useful classes lived under the `autopilot.emulators` namespace. These have now been moved into the `autopilot` namespace.

Note: There is an API breakage in autopilot 1.3. The changes outlined under the heading “*DBus backends and DBusIntrospectionObject changes*” apply to version 1.3.1+13.10.20131003.1-0ubuntu1 and onwards .

QtIntrospectionTestMixin and GtkIntrospectionTestMixin no longer exist

In autopilot 1.2, tests enabled application introspection services by inheriting from one of two mixin classes: `QtIntrospectionTestMixin` to enable testing Qt4, Qt5, and Qml applications, and `GtkIntrospectionTestMixin` to enable testing Gtk 2 and Gtk3 applications. For example, a test case class in autopilot 1.2 might look like this:

```
from autopilot.introspection.qt import QtIntrospectionTestMixin
from autopilot.testcase import AutopilotTestCase

class MyAppTestCase(AutopilotTestCase, QtIntrospectionTestMixin):

    def setUp(self):
```

```
super(MyAppTestCase, self).setUp()
self.app = self.launch_test_application("../my-app")
```

In Autopilot 1.3, the `AutopilotTestCase` class contains this functionality directly, so the `QtIntrospectionTestMixin` and `GtkIntrospectionTestMixin` classes no longer exist. The above example becomes simpler:

```
from autopilot.testcase import AutopilotTestCase

class MyAppTestCase(AutopilotTestCase):

    def setUp(self):
        super(MyAppTestCase, self).setUp()
        self.app = self.launch_test_application("../my-app")
```

Autopilot will try and determine the introspection type automatically. If this process fails, you can specify the application type manually:

```
from autopilot.testcase import AutopilotTestCase

class MyAppTestCase(AutopilotTestCase):

    def setUp(self):
        super(MyAppTestCase, self).setUp()
        self.app = self.launch_test_application("../my-app", app_type='qt')
```

See also:

Method `autopilot.testcase.AutopilotTestCase.launch_test_application` Launch test applications.

autopilot.emulators namespace has been deprecated

In autopilot 1.2 and earlier, the `autopilot.emulators` package held several modules and classes that were used frequently in tests. This package has been removed, and its contents merged into the `autopilot` package. Below is a table showing the basic translations that need to be made:

Old module	New Module
<code>autopilot.emulators.input</code>	<code>autopilot.input</code>
<code>autopilot.emulators.X11</code>	Deprecated - use <code>autopilot.input</code> for input and <code>autopilot.display</code> for getting display information.
<code>autopilot.emulators.bamf</code>	Deprecated - use <code>autopilot.process</code> instead.

These technical documents describe features of the autopilot ecosystem that test authors probably don't need to know about, but may be useful to developers of autopilot itself.

XPathSelect Query Protocol

This document defines the protocol used between autopilot and the application under test. In almost all cases, the application under test needs no knowledge of this protocol, since it is handled by one of the UI toolkit specific autopilot drivers (we support both Gtk and Qt). If you are a test author, you should be aware that this document likely contains no useful information for you!

Contents

- *XPathSelect Query Protocol*
 - *DBus Interface*
 - *Object Trees*
 - *Selecting Objects*
 - * *Selecting the Root Object*
 - * *Absolute Queries*
 - * *Relative Queries*
 - * *Mixed Queries*
 - * *Attribute Queries*
 - *Wildcard Nodes*
 - * *Selecting All Children*
 - * *Selecting Nodes based on Attributes*

- * *Invalid Wildcard Queries*
- *Returning State Data*
- * *Valid IDs*
- * *Special Attributes*
- * *Example GetState Return Values*

Who should read this document:

- People wanting to hack on autopilot itself.
- People wanting to hack on xpathselect.
- People wanting to use autopilot to test an application not supported by the current autopilot UI toolkit drivers.
- People wanting to write a new UI toolkit driver.

DBus Interface

Every application under test must register itself on a DBus bus. Traditionally this is the DBus session bus, but autopilot supports any DBus bus that the user running autopilot has access to. Applications may choose to use a well-known connection name, but it is not required.

The only requirement for the DBus connection is that the `com.canonical.Autopilot.Introspection` interface is presented on exactly one exported object. The interface has two methods:

- `GetVersion()`. The `GetVersion` method takes no parameters, and returns a string describing the DBus wire protocol version being used by the application under test. Autopilot will refuse to connect to DBus wire protocol versions that are different than the expected version. The current version string is described later in this document. The version string should be in the format “X.Y”, where X, Y are the major and minor version numbers, respectively.
- `GetState(...)`. The `GetState` method takes a single string parameter, which is the “XPath Query”. The format of that string parameter, and the return value, are the subject of the rest of this document.

Object Trees

Autopilot assumes that the application under test is constructed of a tree of objects. This is true for both Gtk and Qt applications, where the tree usually starts with an “application” object, and roughly follows widget stacking order.

The purpose of the XPathSelect protocol is to allow autopilot to select the parts of the application that it is interested in.

Autopilot passes a query string to the `GetState` method, and the application replies with the objects selected (if any), and their internal state details.

The object tree is a tree of objects. Objects have several properties that are worth mentioning:

- Each object *must* have a name that can be used as a python identifier. This is usually the class name or widget type (for example, `QPushButton`, in a Qt GUI application).
- Each object *must* have an attribute named “id”, whose value is an integer that is guaranteed to be unique for that object (note however that the same object may appear multiple times in the introspection tree in different places. In this case, the object id must remain consistent between each appearance).
- Objects *may* have more attributes. Each attribute name must be a string that can be represented as a python variable name. Attribute values can be any type that is representable over dbus.

- Objects *may* have zero or more child objects.

This tree of objects is known as an “introspection tree” throughout the autopilot documentation.

Selecting Objects

Objects in a tree are selected in a very similar fashion to files in a filesystem. The `/` character is used to separate between levels of the tree.

Selecting the Root Object

A shortcut exists for selecting the root object in the introspection tree. A query of `/` will always return the root object. This is used when autopilot does not yet know the name of, the root object.

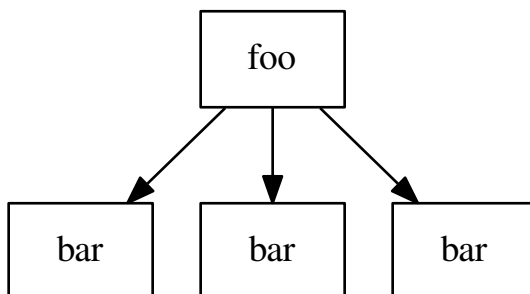
Absolute Queries

Absolute queries specify the entire path to the object, or objects autopilot is interested in. They must start with `/` and specify object names, separated by `/` characters. For example:

Table 12.1: XPathSelect Absolute Queries

Query:	Selects:
<code>/</code>	The root object (see above).
<code>/foo</code>	The root object, which must be named ‘foo’.
<code>/foo/bar</code>	Object ‘bar’, which is a direct child of the root object ‘foo’.

Using absolute queries, it is possible to select nodes in a tree of objects. However, there is no requirement for an absolute query path to match to exactly one object in the tree. For example, given a tree that looks like this:



a query of `/foo/bar` will select two objects. This is allowed, but not always what we want. There are several ways to avoid this, they will be covered later in this document.

Relative Queries

Absolute queries are very fast for the application under test to process, and are used whenever autopilot knows where the object it wants to look at exists within the introspection tree. However, sometimes we need to be able to retrieve

all the objects of a certain type within the tree. XPathSelect understands relative queries, which will select objects of a specified type anywhere in the tree. For example:

Table 12.2: XPathSelect Relative Queries

Query:	Selects:
<code>//foo</code>	All objects named 'foo', anywhere in the tree.

Relative queries are much slower for the application under test to process, since the entire introspection tree must be searched for the objects that match the search criteria. Additionally, relative queries can generate a large amount of data that has to be sent across DBus, which can slow things down further.

Mixed Queries

Absolute and relative queries can be mixed. All the relative queries in the above table will search the entire object tree. However, sometimes you only want to search part of the object tree, in which case you can use a mixed query:

Table 12.3: XPathSelect Mixed Queries

Query:	Selects:
<code>/foo/bar//baz</code>	Select all objects named 'baz' which are in the tree beneath '/foo/bar'
<code>/foo/far//bar/baz</code>	Select all 'baz' objects which are immediate children of a 'bar' object, which itself is in the subtree beneath '/foo/far'.

As you can see, mixed queries can get reasonably complicated.

Attribute Queries

Sometimes we want to select an object whose attributes match certain values. For example, if the application under test is a Qt application, we may want to retrieve a list of 'QPushButton' objects whose 'active' attribute is set to `True`.

The XPathSelect query protocol supports three value types for attributes:

- Boolean attribute values are represented as `True` or `False`.
- String attribute values are represented as strings inside double quote characters. The XPathSelect library understands the common character escape codes, as well as the `\x__` hexadecimal escape sequence (For example: `"\x41"` would evaluate to a string with a single character 'A').
- Integer attribute values are supported. Integers may use a sign (either '+' or '-'). The sign may be omitted for positive numbers. The range for integer values is from -2^{32} to $2^{31} - 1$.

Attribute queries are done inside square brackets (`[. . .]`) next to the object they apply to. The following table lists a number of attribute queries, as examples of what can be achieved.

Table 12.4: XPathSelect Attribute Queries

Query:	Selects:
<code>//QPushButton[active=True]</code>	Select all <code>QPushButton</code> objects whose “active” attribute is set to <code>True</code> .
<code>//QPushButton[label="Deploy Robots!"]</code>	Select all <code>QPushButton</code> objects whose labels are set to the string “Deploy Robots”.
<code>//QPushButton[label="Deploy Robots!", active=True]</code>	Select all <code>QPushButton</code> objects whose labels are set to the string “Deploy Robots”, <i>and</i> whose “active” attribute is set to <code>True</code> .
<code>//QSpinBox[value=-10]</code>	Select all <code>QSpinBox</code> objects whose value attribute is set to -10.

Note: While the XPathSelect protocol has a fairly limited list of supported types for attribute matching queries, it is important to note that autopilot transparently supports matching object attributes of any type. Autopilot will send attribute filters to the application under test using the XPathSelect protocol only if the attribute filters are supported by XPathSelect. In all other cases, the filtering will be done within autopilot. At worst, the test author may notice that some queries take longer than others.

Wildcard Nodes

As well as selecting a node in the introspection tree by node name, one can also use `*` to select any node. However, there are a few restrictions on this feature, to stop the inadvertent selection of the entire tree.

Selecting All Children

Wildcard nodes are often used to select all the children of a particular object. For example, if the query `/path/to/object[id=123]` returns the parent object you are interested in, then the query `/path/to/object[id=123]/*` will return all the immediate children of that object.

Selecting Nodes based on Attributes

The second use of wildcard nodes is to select nodes based purely on their attributes, rather than their type. For example, to select every object with a ‘visible’ property set to ‘True’, the following query will work: `//*[visible=True]`. However, care must be taken - this query will traverse the entire introspection tree, and may take a long time. Additionally, a large amount of data may be returned over DBus.

Invalid Wildcard Queries

The wildcard specifier may only be used after a search separator if you have also specified attribute filters. For example, all the following queries are invalid:

Invalid Queries

- `//*`
- `/path/to/some/node/*`
- `//node/*`

However, the following queries are all valid:

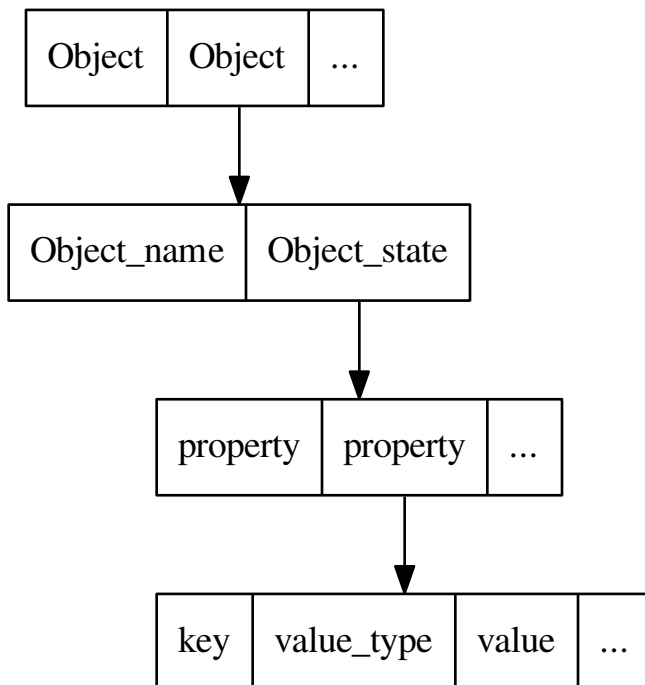
Valid Queries

- `//node/*`
- `/node//*[key="value"]`
- `//node//*[key=True]`

Returning State Data

Once the application under test has parsed the XPathSleect query, and has a list (possibly empty!) of objects that match the given query, it must serialize those objects back across Dbus as the return value from the `GetState` method. The data structure used is reasonably complex, and is described below:

- At the top level, the return type must be an array of objects. Each item in the array represents an object that matched the supplied query. If no objects matched the supplied query, an empty array must be returned.
- Each object is a Dbus structure that has two parts: a string, and a map. The string specifies the full tree path to the object being returned (for example `"/path/to/object"`).
 - The map represents the object state, and is a map of strings to arrays. The keys in this map are property names (for example `"visible"`).
 - The arrays represents the property value. It contains at least two parts, a value type id (see below for a list of these ids and what they mean), and one or more values. The values can be any type representable over dbus. Some values may actually be arrays of values, for example.



Valid IDs

The following table lists the various type IDs, and their meaning.

Table 12.5: Autopilot Type IDs and their meaning

Type ID:	Meaning:
0	Simple Type. The value is a DBus integer, boolean, or string, and that is exactly how it should be represented to the user.
1	Rectangle. The next four values are all integers, and represent a rectangle in cartesian space. The four numbers must represent the x, y, width and height of the rectangle, respectively. Autopilot will likely present the four values as 'x', 'y', 'w' and 'h' to test authors. Autopilot makes no assumptions about the coordinate space used.
2	Point. The next two values are integers, and represent an X, Y, point in catesian space.
3	Size. The next two value are integers, and represent a width,height pair, describing a size in catesian space.
4	Color. The next four values are all integers, and represent the red, green, blue, and alpha components of the color, respectively. Each component is bounded between 0 and 255.
5	Date or Date/Time. The next value is an integer representing the number of seconds since the unix epoch (1970-01-01 00:00:00), UTC time.
6	Time. The next values are all integers, and represent hours, minutes, seconds, milliseconds.
7	3D Point. The next values are all integers, and represent the X, Y and Z coordinates of a point in 3D cartesian space.

Special Attributes

Most attributes that are returned will be attributes of the UI toolkit class itself. However, there are two special attributes:

- The `id` attribute *must* be present, and must contain an integer number. This number must be unique for this instance of the object. This number must also be within the range suitable for integer parameter matching.
- The `Children` attribute *may* be present if the object being serialized has any children in the introspection tree. If it is present, it must be an array of strings, where each string is the class name of the immediate child object.
- The `globalRect` property should be present for any components that have an on-screen presence. It should be a 4-element array containing the x,y,w,h values of the items on-screen coordinates. Note that these coordinates should be in screen-space, rather than local application space.

Example GetState Return Values

All the examples in this section have had whitespace added to make them more readable.

Example 1: Unity Shell

Query: /

Return Value:

```
[
  (
    '/Unity',
    {
      'id': [0, 0],
      'Children': [0,
        [
```

```
        'DashController',
        'HudController',
        'LauncherController',
        'PanelController',
        'Screen',
        'SessionController',
        'ShortcutController',
        'SwitcherController',
        'WindowManager'
    ]
}
)
```

Example 2: Qt Creator Menu

This is a much larger object, and shows the `globalRect` attribute.

Query: `/QtCreator/QMenu[objectName="Project.Menu.Session"]`

Return Value:

```
[
  (
    '/QtCreator/QMenu',
    {
      '_autopilot_id': [0, 3],
      'acceptDrops': [0, False],
      'accessibleDescription': [0, ''],
      'accessibleName': [0, ''],
      'autoFillBackground': [0, False],
      'baseSize': [3, 0, 0],
      'Children': [0, ['QAction', 'DBusMenu']],
      'childrenRect': [1, 0, 0, 0, 0],
      'contextMenuPolicy': [0, 1],
      'enabled': [0, True],
      'focus': [0, False],
      'focusPolicy': [0, 0],
      'frameGeometry': [1, 0, 0, 100, 30],
      'frameSize': [3, 100, 30],
      'fullScreen': [0, False],
      'geometry': [1, 0, 0, 100, 30],
      'globalRect': [1, 0, 0, 100, 30],
      'height': [0, 30],
      'id': [0, 3],
      'inputMethodHints': [0, 0],
      'isActiveWindow': [0, False],
      'layoutDirection': [0, 0],
      'maximized': [0, False],
      'maximumHeight': [0, 16777215],
      'maximumSize': [3, 16777215, 16777215],
      'maximumWidth': [0, 16777215],
      'minimized': [0, False],
      'minimumHeight': [0, 0],
      'minimumSize': [3, 0, 0],
      'minimumSizeHint': [3, -1, -1],
      'minimumWidth': [0, 0],
      'modal': [0, False],
```



```

        'mouseTracking': [0, True],
        'normalGeometry': [1, 0, 0, 0, 0],
        'objectName': [0, 'ProjectExplorer.Menu.Debug'],
        'pos': [2, 0, 0],
        'rect': [1, 0, 0, 100, 30],
        'separatorsCollapsible': [0, True],
        'size': [3, 100, 30],
        'sizeHint': [3, 293, 350],
        'sizeIncrement': [3, 0, 0],
        'statusTip': [0, ''],
        'stylesheet': [0, ''],
        'tearOffEnabled': [0, False],
        'title': [0, '&Debug'],
        'toolTip': [0, ''],
        'updatesEnabled': [0, True],
        'visible': [0, False],
        'whatsThis': [0, ''],
        'width': [0, 100],
        'windowFilePath': [0, ''],
        'windowIconText': [0, ''],
        'windowModality': [0, 0],
        'windowModified': [0, False],
        'windowOpacity': [0, 1.0],
        'windowTitle': [0, ''],
        'x': [0, 0],
        'y': [0, 0]
    }
)
]

```

Note that most attributes are given the “plain” type id of 0, but some (such as ‘pos’, ‘globalRect’, and ‘size’ in the above example) are given more specific type ids.

SYNOPSIS

DESCRIPTION

autopilot is a tool for writing functional test suites for graphical applications for Ubuntu.

OPTIONS

General Options

-h, --help Get help from autopilot. This command can also be present after a sub-command (such as run or list) to get help on the specific command. Further options are restricted to particular autopilot commands.

-v, --version Display autopilot version and exit.

—enable-profile Enable collection of profile data for autopilot itself. If enabled, profile data will be stored in ‘autopilot_<pid>.profile’ in the current working directory.

list [options] suite [suite...] List the autopilot tests found in the given test suite.

suite See *SPECIFYING SUITES*

-ro, --run-order List tests in the order they will be run in, rather than alphabetically (which is the default).

--suites Lists only available suites, not tests contained within the suite.

run [options] suite [suite...] Run one or more test suites.

suite See *SPECIFYING SUITES*

- o FILE, --output FILE** Specify where the test log should be written. Defaults to stdout. If a directory is specified the file will be created with a file name of `<hostname>_<dd.mm.yyy_HHMMSS>.log`
- f FORMAT, --format FORMAT** Specify the format for the log. Valid options are 'xml' and 'text' 'subunit' for JUnit XML, plain text, and subunit, respectively.
- ff, --failfast** Stop the test run on the first error or failure.
- r, --record** Record failed tests. Using this option requires the 'recordmy-desk top' application be installed. By default, videos are stored in `/tmp/autopilot`
- record-options** Comma separated list of options to pass to recordmydesktop
- rd DIR, --record-directory DIR** Directory where videos should be stored (overrides the default set by the -r option).
- ro, --random-order** Run the tests in random order
- v, --verbose** Causes autopilot to print the test log to stdout while the test is running.
- debug-profile** Select a profile for what additional debugging information should be attached to failed test results.
- timeout-profile** Alter the timeout values Autopilot uses. Selecting 'long' will make autopilot use longer timeouts for various polling loops. This can be useful if autopilot is running on very slow hardware
- launch [options] application** Launch an application with introspection enabled.
 - v, --verbose** Show autopilot log messages. Set twice to also log data useful for debugging autopilot itself.
 - i INTERFACE, --interface INTERFACE** Specify which introspection interace to load. The default ('Auto') uses ldd to try and detect which interface to load. Options are Gtk and Qt.
- vis [options]** Open the autopilot visualizer tool.
 - v, --verbose** Show autopilot log messages. Set twice to also log data useful for debugging autopilot itself.
 - testability** Start the vis tool in testability mode. Used for self-tests only.

SPECIFYING SUITES

Suites are listed as a python dotted package name. Autopilot will do a recursive import in order to find all tests within a python package.

E

environment variable
 PATH, [10](#), [12](#)

P

PATH, [10](#), [12](#)
Python Enhancement Proposals
 PEP 257, [30](#)