

# Framework Scripting Guidelines

---

*Accenture StormTest Development Center Test Framework*

Document ID: ST-15010

Revision Date: October 2016

Version: 1.11

Contact us at <http://www.accenturestormtest.com> or [stormtest-support@accenture.com](mailto:stormtest-support@accenture.com)

The contents of this document are owned or controlled by Accenture and are protected under applicable copyright and/or trademark laws. The contents of this document may only be used or copied in accordance with a written contract with Accenture or with the express written permission of Accenture.

## Contents

<b>1</b>	<b>Setup .....</b>	<b>4</b>
1.1	File Layout .....	4
1.2	Configuration of the device-specific areas of the framework .....	5
1.2.1	Multi-device testing – how it works.....	6
1.2.2	Storing .stscreen files .....	<b>Error! Bookmark not defined.</b>
1.3	Config file setup .....	8
1.4	Navigator Setup .....	9
<b>2</b>	<b>Navigation using the Framework.....</b>	<b>10</b>
2.1	When to use the Navigator for navigating.....	12
2.2	Currently Saved Screen .....	12
2.3	Navigating: Source Screen is Verified .....	12
2.4	Navigating from Screens which Time Out.....	13
2.5	Using Navigator after a Language Change. ....	14
2.5.1	Implementing the Language Change .....	14
2.6	Validating All Navigable Navigator Screens .....	15
2.6.1	How it Works:.....	15
<b>3</b>	<b>Navigator Verifications .....</b>	<b>16</b>
3.1	Compare Image.....	16
3.2	OCR Compare String.....	16
3.3	Detect Motion.....	16
3.4	Compare Colour .....	16
3.5	Navigator Screen Names.....	17
3.6	Use of Resources.....	18
3.7	Use of SDOs (Screen Definition Objects) .....	19
<b>4</b>	<b>Reading Dynamic Text .....</b>	<b>19</b>
4.1	Tip: Dynamic nature.....	20
<b>5</b>	<b>Test Verifications.....</b>	<b>22</b>
5.1	Exception: Verification of Unknown-Location Icons.....	22
5.2	Exception: Verification of Change.....	22
<b>6</b>	<b>Zap Measurement .....</b>	<b>22</b>
<b>7</b>	<b>Tests .....</b>	<b>23</b>
7.1	Path to Utilities .....	23
7.2	Test Layout - Steps.....	23

7.3	Step Content .....	24
<b>8</b>	<b>STB Methods .....</b>	<b>25</b>
8.1.1	Test Sections .....	26
<b>9</b>	<b>Basic Utility methods .....</b>	<b>27</b>
9.1	stbGeneric .....	27
9.2	Other common utilities .....	30
<b>10</b>	<b>Error Handling .....</b>	<b>31</b>
<b>11</b>	<b>Warning Center in the Framework.....</b>	<b>32</b>
11.1	Warning Center utilities .....	32
11.2	Automatic Warning Center handling .....	33
11.2.1	Event information in Warning Center handler .....	33
11.2.2	Using Steps to set up Warning Center Events .....	33
11.2.3	Using Steps to send Warning Center Events.....	35
11.3	Developer-driven Warning Center handling .....	36
11.3.1	Updating the Warning Center result during testing .....	36
11.3.2	Non-Critical Processing within a Warning Center step .....	37
11.3.3	Secondary Warning Center Events .....	39
<b>12</b>	<b>Warning Center Tests.....</b>	<b>41</b>
12.1	Setup of the sample Warning Center Tests .....	41
12.2	Writing your own Warning Center Tests .....	43
12.2.1	Test Scripts .....	43
12.2.2	checkXXX Methods.....	44
12.3	Running Warning Center Tests .....	47

## Setup

### File Layout

The framework is broken into 2 separate areas:

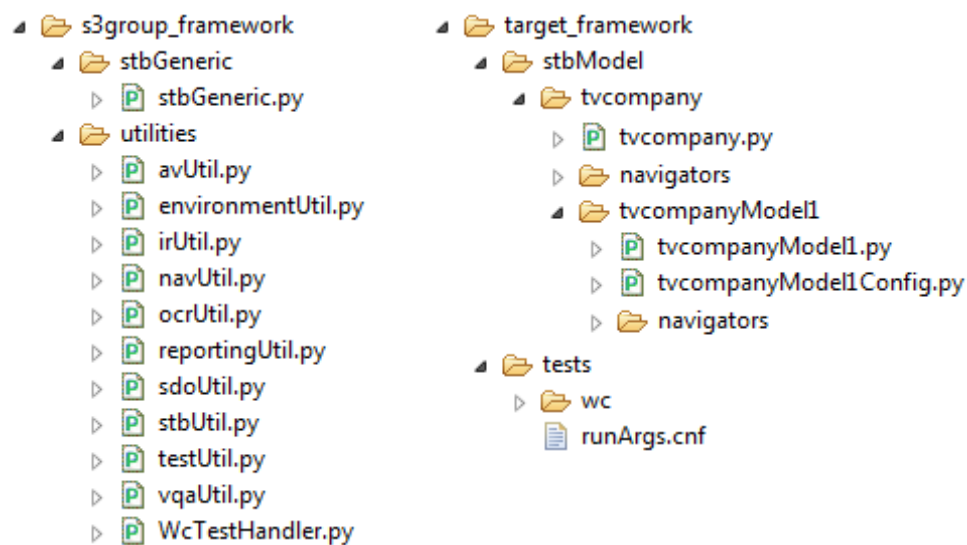
#### *s3group\_framework*

- Files containing generic code (i.e. code that is common to all projects)

#### *target\_framework*

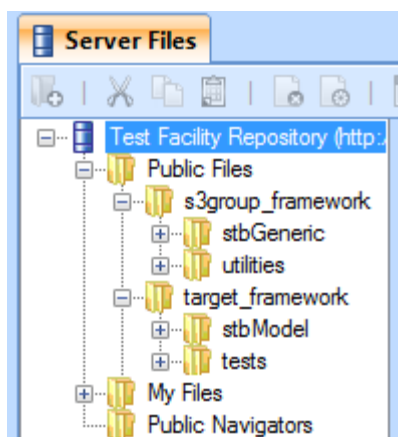
- Project-specific files need to be created

The following is the required structure of each area:



The contents of the *s3group\_framework* area should be downloaded directly. Only the *stbGeneric* and *utilities* folders should be retained in this area. The two other folders (*stbModel* and *test.examples*) should be used as templates to populate the project-specific *target\_framework* area as shown above.

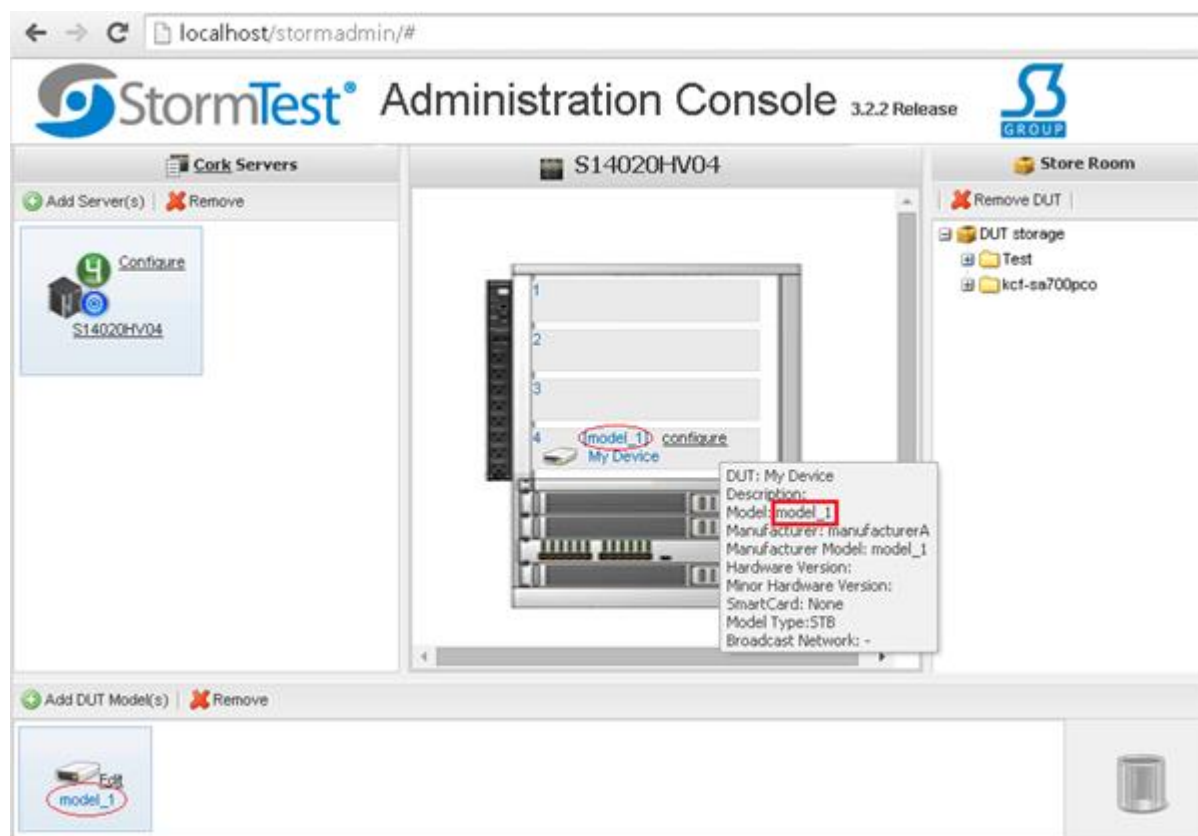
If running tests from the StormTest server, these two folders should be copied to the “Public Files” area.



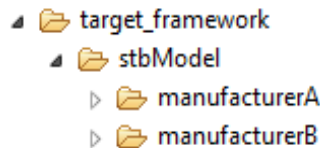
## Configuration of the device-specific areas of the framework

Once the basic files are in place, the DUT needs to be considered. Add the device to the StormTest server as usual, setting up the appropriate model and DUT instance in the Administration Console. Take particular note of the model name; this will be used in the framework to distinguish the various different types of devices.

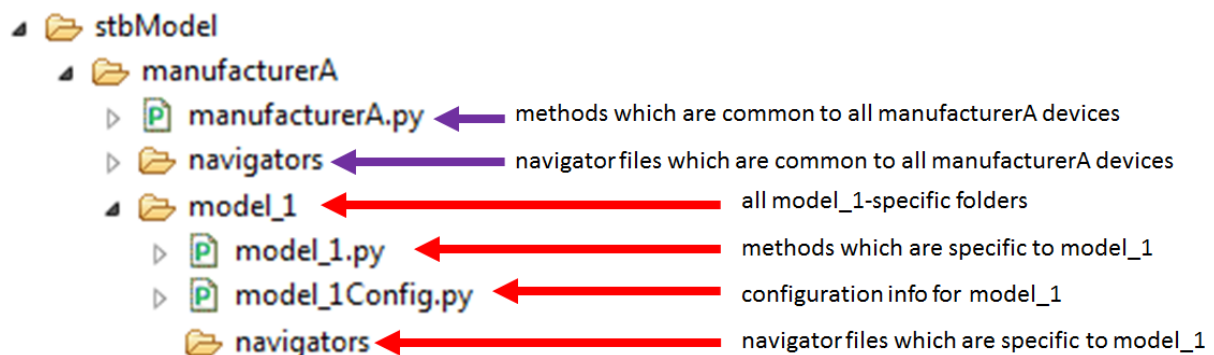
In the example below, an STB is added to the rack and is given the model name *model\_1* (highlighted in red below). No spaces or periods should be used in this name.



The template files and folders provided in the *stbModel* area now need to be renamed to correspond to the device(s) which will be tested. Rename the *tvcompany* folder to match the name of the device manufacturer (e.g. *manufacturerA*). This can be a name of your choosing.



The contents of each company/manufacturer folder should be renamed according to the device manufacturer and model name as follows (taking the device above as an example, where the manufacturer name is *manufacturerA* and the model name is *model\_1* - the model folder is always within the manufacturer folder):



It is important that the model “Name” which was given in the StormTest Administration Console (*model\_1*) is used exactly for all of the model-specific folders above (although differences in case between the admin console and the framework name are tolerated, the name used throughout the framework should be consistent).

*Note:* the navigator folder which is at the manufacturer level above is optional; often devices are so different that only device-specific navigator files are relevant.

Once the template files provided have been renamed according to the DUT then the framework is ready to be used.

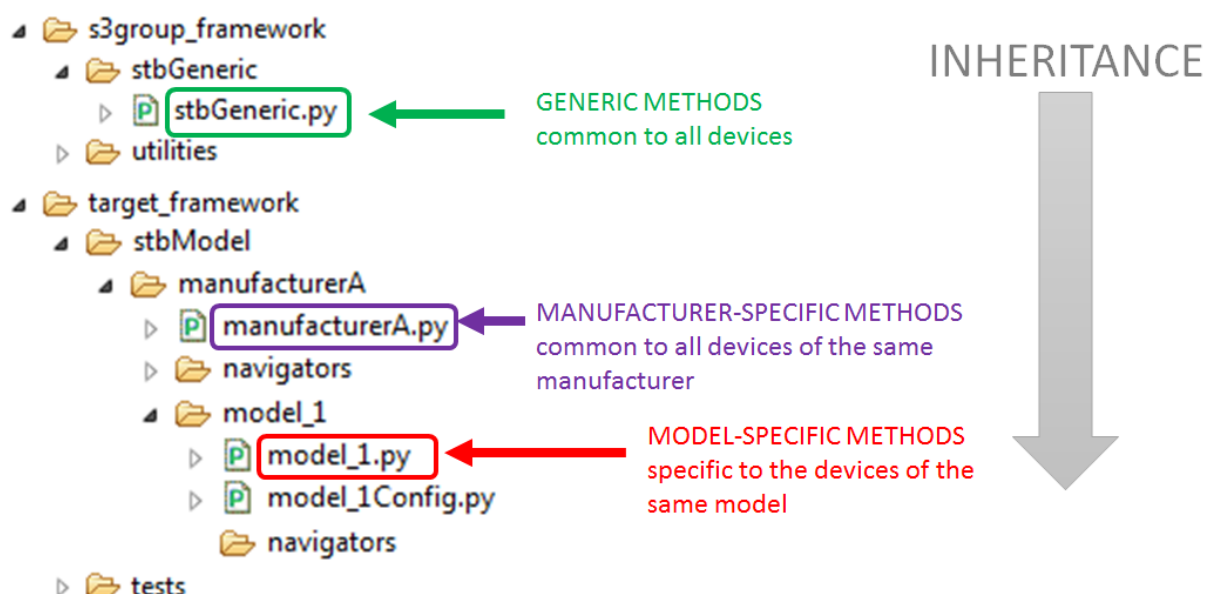
### Multi-device testing – how it works

The idea with this framework is that the same test script can be run on a number of different devices, depending on which type of device is in the slot that the test is being run on.

The core building blocks which are used to write a test script are methods from the *stb* class. These methods can be defined in a 3 different files, depending on whether they are generic, manufacturer-specific or model-specific.

- i. **Generic** methods can be used by all devices
- ii. **Manufacturer-specific** methods can be used by all devices of the same manufacturer
- iii. **Model-specific** methods can be used only by devices of the same model

The names and locations of the files containing each of these types of methods are shown below.

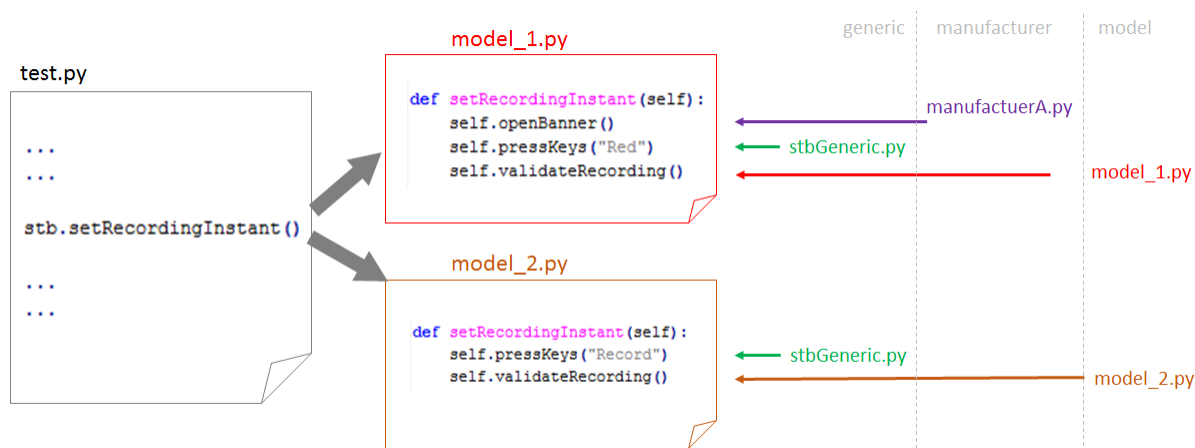


The object-oriented principal of inheritance applies to these stb files. Generic methods are available to all manufacturer-specific methods and similarly all model-specific methods have access to both generic methods and their manufacturer-specific methods. This reduces code duplication; if a method is the same for all but one device, it can be defined in the manufacturer file and simply redefined in the model-specific file of the device for which it is different.

When a test is run on a particular slot, the framework needs to create an instance of the stb class for the specific device on which the test is being run. Thus, before its instantiation, the appropriate stb model file and manufacturer file need to be loaded (the generic stb file is always loaded). The appropriate files are selected by acquiring the model name (e.g. *model\_1*) from StormTest and using it to find the corresponding model folder within the *stbModel* area (e.g. the *model\_1* folder). Therefore the correct model files and manufacturer files for any device are loaded (e.g. *model\_1.py* and *manufacturerA.py* respectively) and thus the stb instance which is created contains the correct methods for the device type under test.

Generic methods are provided with the framework (in the *s3group\_framework* area) and should not be modified in situ (if modification of a method is required for a project, the method should be overwritten in either the manufacturer or model file). The manufacturer and device methods (in the *target\_framework* area) need to be created for each project and device.

All of this means that, as long as all of the methods called directly in a test script are defined somewhere within the 3 stb files which are loaded for any given device (*generic.py*, *manufacturer.py* and *model.py*), this test can be run across all devices without changing the actual test code. The bodies of the methods will obviously change with each device, but the test itself does not need to.



In the example above a single test (*test.py*) can be run on two devices (*model\_1* and *model\_2*) because the method which is used (*setRecordingInstant*) is defined for both devices (i.e. a definition exists in both *model\_1.py* and *model\_2.py*). In this particular case, each model has its own definition of the method, but it is also conceivable that there would a manufacturer-specific method (defined in *manufacturerA.py*) which applies to *model\_1* but not to *model\_2*; in this case, *model\_2.py* would not contain any definition of *setRecordingInstant* – it would simply inherit the definition from *manufacturerA.py*.

The right hand side of the diagram shows the origin of the sub-methods (they span all hierarchy levels).

It is important to note that the *stb* object which is created at test startup (called *stb*) contains the appropriate methods for the device type under test. Thus, all *stb* methods must be called from this object. Thus, the following calls show how *methodA* is called from the test script and from within other *stb* methods respectively:

From within a test script: ***stb.methodA()***

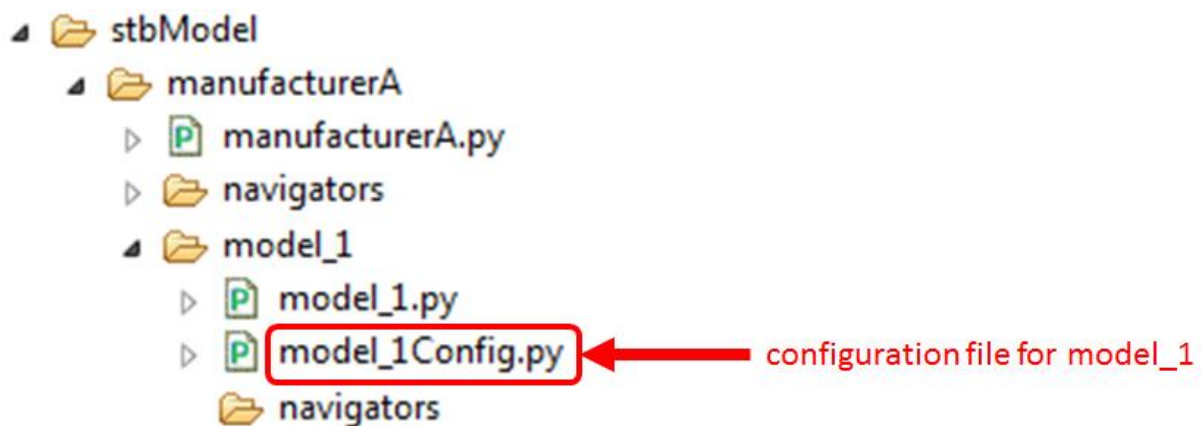
From within an *stb* method: ***self.methodA()***

## Config file setup

A configuration file exists for each device model and is named accordingly (e.g. *model\_1Config.py*). This file is used to store data which is specific to either the environment or device. Test parameters can also be stored here.

The data is stored in nested dictionary form. Hierarchical configuration files are not currently supported. A sample configuration file is provided in the framework and is a good starting point when creating your own model config file.





Configuration items can be accessed by calling one of the following generic stb methods with the required list of dictionary names/keys.

- getConfigItem
- getConfigItemIfAvailable

Examples of use:

```
audio_threshold = stb.getConfigItem("test_options", "AUDIO_THRESHOLD")

suffix_spanish = stb.getConfigItem("general", "LANGUAGE_SUFFIX_DICT", "SPANISH")
```

## Navigator Setup

All navigator files which are to be used should be listed in the config file as shown below (the relative path from the model area must be included, but not the file extension). The initial screen and the navigator file it is contained in are also specified (see explanation in section 0). The navigators must be used from files, not from the public navigator area of the Stormtest server – they should be stored in a “navigators” folder within the manufacturer and/or model area of the stbModel folder.

```
config_data = dict (
    general=dict(
        INITIAL_SCREEN="LIVE",
        INITIAL_NAVIGATOR="navigators\\top",
        NAVIGATORS=[
            "navigators\\top",
            "navigators\\tvguide",
            "navigators\\banner",
            "navigators\\misc",
        ],
    ),
)
```

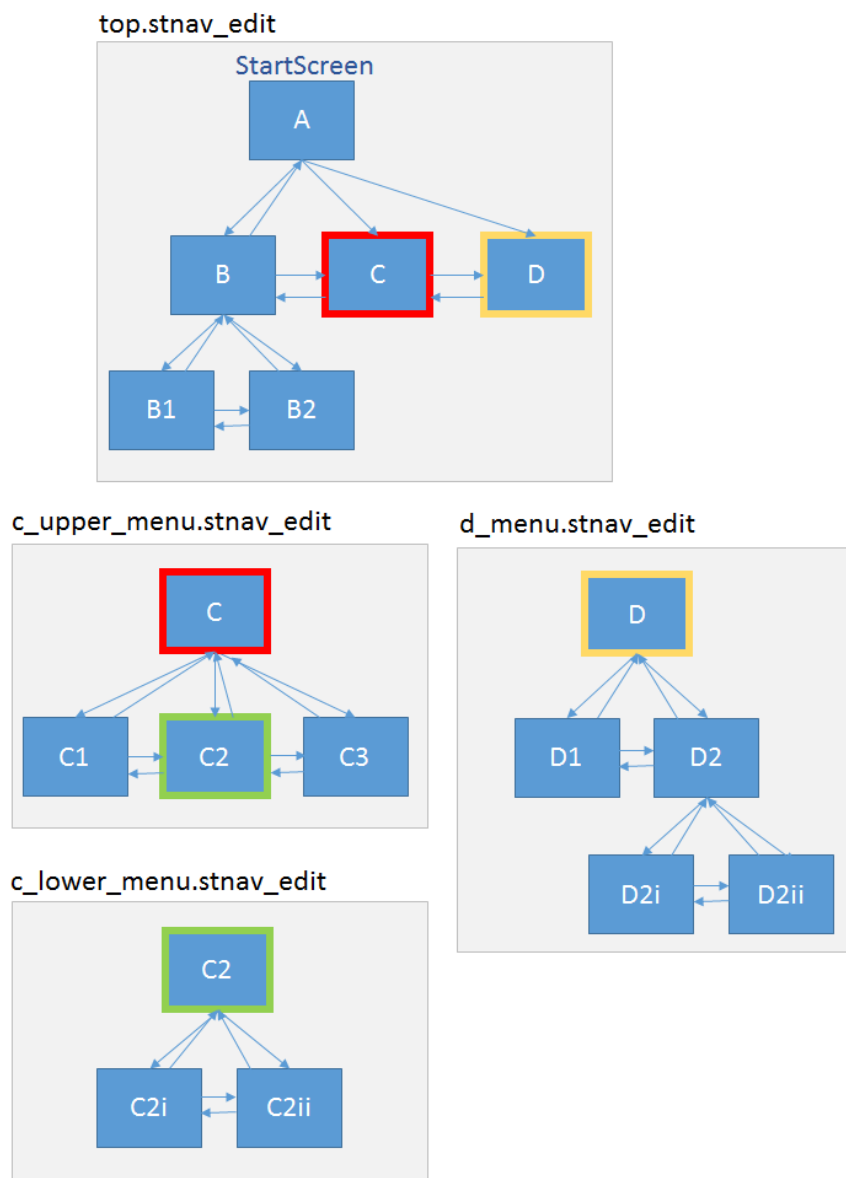
With this in place, navigation can be done between any of the screens shown using the generic stb method *navigateTo*.

## Navigation using the Framework

The framework provides extra functionality which allows for navigation using multiple navigator files. This is beneficial in two ways:

1. a large menu tree can to be split into more manageable groupings
2. multiple people can work on different navigator files simultaneously

In order to illustrate how this works, consider the following example whereby a number of different menu levels are represented across 4 navigator files.



One of these navigator files must contain a StartScreen (this should be a screen like LIVE which can be reached from everywhere using a defined button or buttons). It is important that only one start screen is defined. This screen is referred to as the "initial screen".

Each navigator file must share at least one screen with at least one other navigator file. These screens are duplicates of each other and are referred to as "common screens". These screens are used as a means of navigating between navigator files. The common screen pairs are shown in corresponding coloured outlines in the diagram above; C, C2, D

Navigation can now be done between any of the screens shown using the generic stb method *navigateTo*.

```
stb.navigateTo ( screen_to = "C3", screen_from = "" )
```

- no source screen is defined so the start screen will be used
- Path: A -> C -> C3 ... (two navigator files are used)

```
stb.navigateTo ( screen_to = "C3", screen_from = "B" )
```

- a source screen is defined so the start screen does not need to be used
- Path: B -> C -> C3 ... (two navigator files are used)

```
stb.navigateTo ( screen_to = "D2", screen_from = "C2" )
```

- Path: C2 -> C -> D -> D2 ... (three navigator files are used)

```
stb.navigateTo ( screen_to = "D2ii", screen_from = "C2ii" )
```

- Path: C2ii -> C2 -> C -> D -> D2 -> D2ii ... (four navigator files are used)

Note: it is usually assumed that valid paths exist between all screens in a navigator. However, if a path between the source and destination screen does not exist, then an attempt will be made to navigate to the destination screen using the "initial screen" (i.e. it will first navigate to the start/initial screen and then navigate to the desired destination screen).

The inner workings of this can be found in navUtil.py – the idea is that, given the intended source and destination screens, an attempt is made to find a path between them using the minimum number of navigator files (this is not necessarily the minimum path length). Paths can either be simple paths (within the same navigator file) or multi-part paths (involving a number of navigator files and using the common screens to link between them). For each part of the path the relevant navigator file will be opened and the StormTest navigateTo API is used to perform that section of the full path. There is a depth limit of about 15 navigator files in all.

## When to use the Navigator for navigating

The Navigator should be used for navigation between screens whenever possible. Explicit key presses should be avoided in code except when necessary. If it is functional (e.g. press Record button) or if it's navigating within a screen (e.g. between events in the tv guide) then it's ok. Examples of when key presses would be considered recommended and not recommended are as follows:

Example	Key Press Recommended / Not Recommended
Press "Right" to navigate to the next menu item	Not Recommended (use Navigator)
Press "Live" to go back to the live TV	Not Recommended (use Navigator)
Press "Right" to navigate to the next event in the TV Guide	Recommended (you are navigating within a screen, not to new one)
Press "OK" to navigate to the banner	Recommended (the banner can have an extremely short timeout so we avoid using the Navigator to deal with it)
Press "Rec" to record an event	Recommended (this is not a navigation, it is test of functionality. Also the steps required to set a recording can change depending on the event type and/or existing recordings)

## Currently Saved Screen

When using the navigator methods it is important to be aware of the fact that the current screen is stored locally (in the stb object). This can be set in a number of ways, both explicitly and implicitly:

- Explicitly set by the stb method *updateNavLocation* (only do this if you are certain of the screen you are on because you just made some sort of check).
- Explicitly set by specifying the *update\_nav* parameter as True in the following stb methods: *checkIfAt*, *validateAt*.
- Implicitly set when a successful navigation (using *stb.navigateTo*) takes place.

## Navigating: Source Screen is Verified

It is important to remember that when a call to *stb.navigateTo* is made, the source screen (whether provided explicitly or implicitly) will actually be verified before any navigation is done. Thus, if you request a navigation from *screenA* to *screenB*, the call will fail if you are not currently on *screenA*.

## Navigating from Screens which Time Out

Screens which time out can be handled in two ways:

**Option 1:** when there are only a small number of isolated screens which time out

Avoid using the *stb.navigateTo* call to navigate to screens which time out and do not update the currently-saved screen to such screens. Instead, only navigate to non-time-out screens and then manually open the desired time-out screen from there. For example, if you need to open the banner, the following is what advised and not advised:

**Not Advised**

```
stb.navigateTo("BANNER")
```

**Advised**

```
stb.navigateTo("LIVE")
stb.pressKeys("Ok")
stb.validateAt("BANNER")
```

**Option 2:** when there are a number of timeout screens which need to be handled

The framework provides a way of dealing with navigation using screens which time out. All such screens must be specified in the config file using one of the following entries. These entries are provided in the sample config file.

TIMEOUT_SCREEN_REGEX_LIST	List of regular expression values which determine the screens which time out
TIMEOUT_SCREEN_ADDITIONAL	Screen names which should be added to those specified by the regular expression list
TIMEOUT_SCREEN_EXCEPTIONS	Screen names which would match the specified regular expressions but which should be not be included in the timeout list

The handling works as follows: At the start of a test, a list of all screens which time out is compiled. Because these screens may disappear very soon after they are navigated to, they should never be navigated away from.

If a call to *navigateTo* is made whereby the source screen is one of the timeout screens, then the initial screen (as defined in the config file) will be navigated to before moving onward.

## Using Navigator after a Language Change.

The framework provides a means of handling changes in the UI language as follows:

The config file should include entries for the following:

- LANGUAGE\_SUFFIX\_DICT (a dictionary in the "general" config area)
- LANGUAGE (the current language in the "settings" config area)

```
config_data = dict(
    general=dict(
        ...
        LANGUAGE_SUFFIX_DICT=dict(
            ARABIC="_AR",
            SPANISH="_ES",
            FRENCH="_FR"
        ),
    ),
    settings=dict(
        LANGUAGE="English", # "English", "French", "Arabic", "Spanish"
    ),
)
```

Navigator screens in the default language (e.g. English) should be created as normal. All screens which are required to be available in an alternative language (i.e. the minimum required for the test, including the screens needed to reset the language) should be created with a language suffix; e.g. the French version of "MENU" should be named "MENU\_FR". Navigator files should contain screens of a single language only.

After the language has been changed in a test, all references to navigator screens (e.g. calls to `checkIfAt`, `validateAt`, `navigateTo`, `updateNavLocation`, `navigateToInitialScreen`, etc) can be made as normal using the default-language screen names - thus the code doesn't need to change!

If the current language that is set (as per the LANGUAGE setting in the config file) is listed as a key in the LANGUAGE\_SUFFIX\_DICT in the config file, then a check will be done to determine if a language-specific version of the screen exists (using the specified suffix list). If such a screen exists it will be used instead of the specified screen.

E.g. if the language has been set to French (thus setting language to "french" in config file) then the following call

```
stb.checkIfAt ("MENU")
```

will actually be carried out as if the following call was made:

```
stb.checkIfAt ("MENU_FR")
```

(provided that the screen "MENU\_FR" exists of course).

## Implementing the Language Change

A template method `changeLanguage` is provided in `stbGeneric` which describes how a language change should be implemented in order to allow the functionality described above to work effectively.

## Validating All Navigable Navigator Screens

The stormtest API *ValidateNavigator* can only be run on a single navigator file which is required to contain a start screen. In scripting projects, we not only use multiple navigator files, but only one navigator file should contain a start screen. Therefore only a very small subset of all navigable screens may be validated in this manner.

When software updates are being doing during test development, it is crucial to be able to quickly detect changes to the UI. A method called *validateAllNavigators* is provided in *stbGeneric.py* in order to achieve this purpose.

### How it Works:

Non-navigable navigator files are specified in the config file using a list of regular expressions. Individual non-navigable screens from otherwise-navigable files should also be specified in a list.

```
config_data = dict(
    general=dict(
        ...
        NON_NAVIGATABLE_NAVIGATORS_REGEX_LIST=[ "_ocr\\.stnav",
                                                  "_verifications\\.stnav",
                                                  "errors\\.stnav",
                                                  ],
        NON_NAVIGATABLE_SCREEN_LIST=[ "NO_VIDEO" ],
    ),
)
```

A list of all navigable screens is thus generated.

The method itself functions as follows:

1. Navigate to the initial screen (LIVE)
2. For each screen which has not already been validated, navigate to that particular screen.
  - Each intermediate screen which is passed on the way is added to a list of validated screens.
  - If the navigation fails at any point, the screen on which failure occurred is added to a list of failing screens (the name of the failing screen can be found as the last value of *nextScreen* from *navigatorCallback* method). Process starts again at Step (1).

At completion, a list of validated screens and a list of error screens has been generated.

## Navigator Verifications

When setting verifications on navigator screens, the Trainer should be used to verify if the tolerances you are setting are adequate. It is important that screen verifications pass when the expected screen is showing, but also it is equally important that at least one verification will fail when the expected screen is not showing.

The following are some general guidelines that may be useful to get started.

### Compare Image

On HD systems tolerance of over 90% should be used: 98% is recommended.

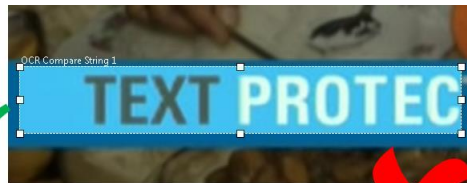
Don't use image compare to verify colours: it does not perform well for single-colour areas. Colour compare could be used instead.

### OCR Compare String

The selected area should be vertically tight around the text which needs to be read:



All text in an area should be of the same colour



### Detect Motion

Motion detect should be avoided on all screens except where absolutely necessary: e.g. verification screens which are used to detect motion explicitly like FULLSCREEN\_MOTION. Motion detection should not be used on navigator screens which are used for navigation purposes.

When using this verification, ensure that the timeout is sufficient to be able to detect motion even on screens which are mostly static (e.g. news show). At least 30 seconds is recommended.

### Compare Colour

Recommended thresholds:



Flatness	<input type="checkbox"/>	50
MaxWait Time	<input type="checkbox"/>	1
PassOnNoMatch		False
PeakError	<input type="checkbox"/>	50
ReferenceColor	<input checked="" type="checkbox"/>	132, 128, 105
ToleranceBlue	<input type="checkbox"/>	30
ToleranceGreen	<input type="checkbox"/>	30
ToleranceRed	<input type="checkbox"/>	30

## Navigator Screen Names

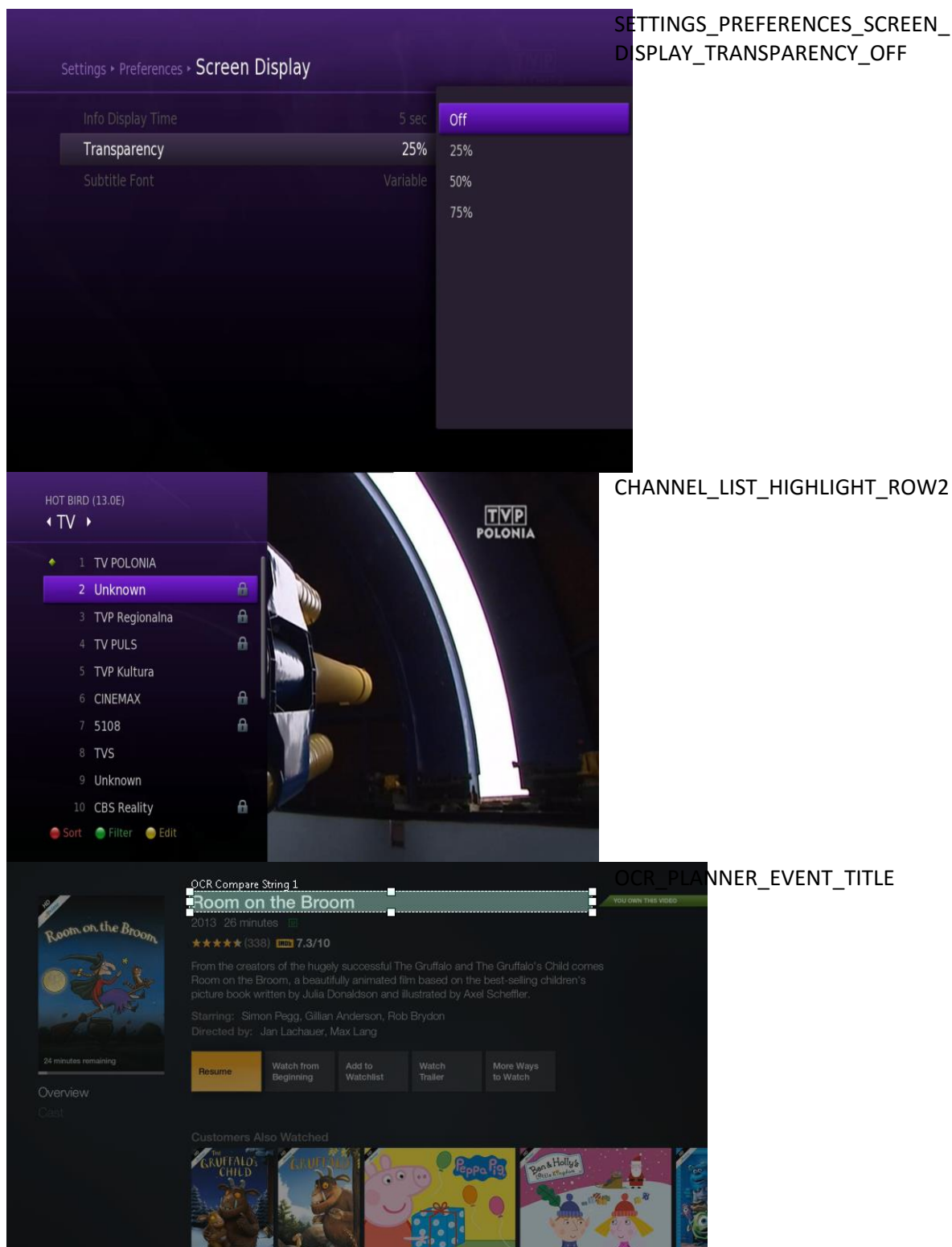
Navigator screens should be named in such a way that their context and/or purpose is obvious. The hierarchical context of the screen is usually the best way of doing this.

For example



MENU\_ALL\_CHANNELS

MENU\_ENTERTAINMENT



## Use of Resources

Since resources can only be stored centrally on the server they are not practical for scripting purposes where multiple developers are working in parallel. Therefore it is not recommended to use them on any scripting projects.

## Use of SDOs (Screen Definition Objects)

SDOs are essentially individually-saved navigator screens. They provide more sophisticated logging information and an ability to dynamically modify constituent verifications.

Although Stormtest allows SDOs to be defined explicitly and saved as individual files, this approach is not recommended when using the framework. The recommended approach is to use Navigator screens to define all verifications and OCR operations (as well as screens for navigating).

At the start of each test, all navigator screens which are in scope (i.e. those which are defined in the config-listed navigator files) are automatically converted to navigator screens and stored in a dictionary. Thus, the option exists to use any navigator screen in either its original “screen” form or its “sdo” form. This is the only type of SDO that should be used when using the framework.

Aside: Some of the reasons for not using explicitly-defined SDOs (whether defined initially as an SDO or converted from a navigator screen using the GUI) are as follows:

- Isolated SDOs are not part of a navigator screen context – they are just files floating around the place, so it is not easy to relate the verification to the box state or menu layout (in contrast to a cluster of navigation screens)
- Editing of SDOs is not an intuitive process - full information is often not present and thus it can be very awkward.

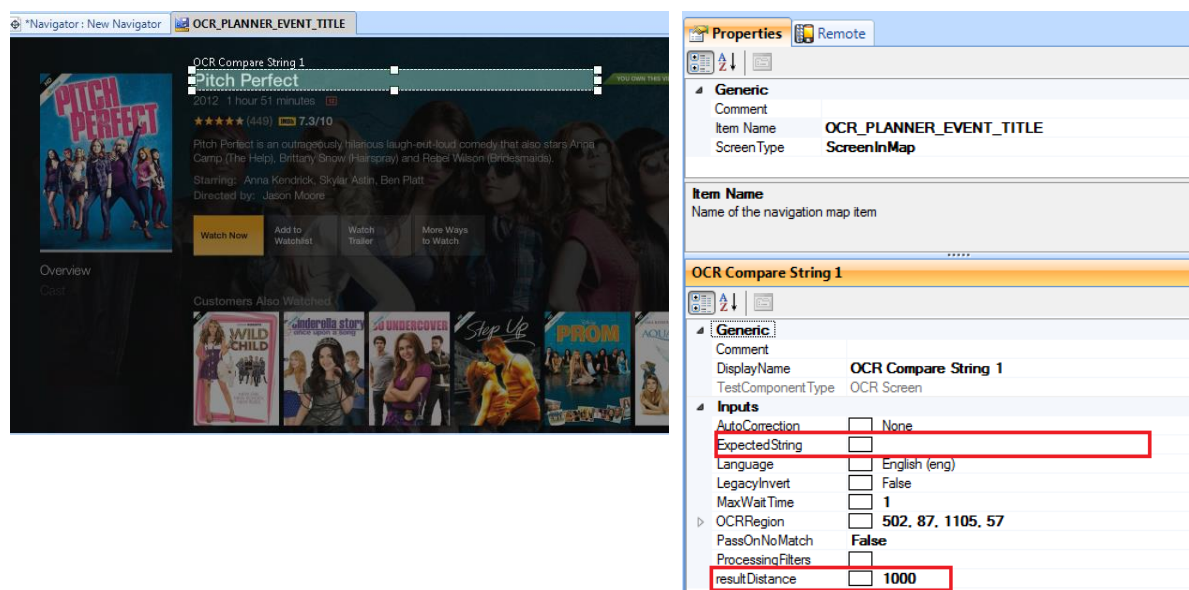
## Reading Dynamic Text

All reading of dynamic text (text which changes) should be done by using navigator screens with the following stb methods (explained in Section 0):

- |               |                |
|---------------|----------------|
| • readText    | • readTime     |
| • readNumbers | • readContains |
| • readDate    |                |

The screens which are required for use by these methods should be single un-mapped (i.e. non-navigable) screens which contain at least one OCR Compare String verification. The verifications themselves should have a blank “ExpectedString” value and a setting of 1000 for “resultDistance”. (note: If the screen has been converted to an SDO then the Verify parameter must be set to False while the expected string and resultDistance parameters are ignored.)

Such screens should be named with the “OCR\_” prefix (e.g. OCR\_PLANNER\_EVENT\_TITLE, OCR\_TVGUIDE\_CURRENT\_TIME)



Multiple strings can be read from a screen if more than one OCR Compare String verification is set. This will cause a list of results to be output by the methods above (in the order that they are defined on the screen); at the moment there is no way of differentiating the results – this will be added at a later date, whereby the results will be output as a dictionary with verification names acting as keys.

Another interesting application of these screens is where additional verifications are defined before the OCR verifications in order to allow the dynamic OCR to be triggered by some condition. In this way, the OCR read operations will be delayed until such a point as the initial verifications pass. For example, if you want to read the text on a popup but you don't know when it will appear you can set an image compare verification to detect the popup; if the timeout of this verification is 60 seconds then Stormtest will wait for up to 60 seconds until the popup appears before it attempts to read text.

It is important to note that the “triggering” verifications must be defined before the OCR Compare String verifications (note: If the screen has been converted to an SDO however, this is not required).

## Tip: Dynamic nature

Always ensure that the area you select will completely contain all of the required text, even when the dynamic nature of the screen is taken into account.

OCR Compare String 1

**Pitch Perfect**

2012 1 hour 51 minutes TV

★★★★★ (449) INDU 7.3/10

Pitch Perfect is an outrageously hilarious laugh-out-loud comedy that also stars Anna Camp (The Help), Brittany Snow (Hairspray) and Rebel Wilson (Bridesmaids).

Starring: Anna Kendrick, Skylar Astin, Ben Platt

Directed by: Jason Moore

YOU OWN THIS VIDEO



OCR Compare String 1

**Pitch Perfect**

2012 1 hour 51 minutes TV

★★★★★ (449) INDU 7.3/10

Pitch Perfect is an outrageously hilarious laugh-out-loud comedy that also stars Anna Camp (The Help), Brittany Snow (Hairspray) and Rebel Wilson (Bridesmaids).

Starring: Anna Kendrick, Skylar Astin, Ben Platt

Directed by: Jason Moore

YOU OWN THIS VIDEO



OCR Compare String 1

**Room on the Broom**

2013 26 minutes TV


★★★★★ (338) INDU 7.3/10

From the creators of the hugely successful The Gruffalo and The Gruffalo's Child comes Room on the Broom, a beautifully animated film based on the best-selling children's picture book written by Julia Donaldson and illustrated by Axel Scheffler.

Starring: Simon Pegg, Gillian Anderson, Rob Brydon

Directed by: Lee Leakey, Mark Leno

YOU OWN THIS VIDEO



## Test Verifications

**All verifications which are performed in the course of a test** (whether called from the test script itself or from an stb method) **should be done using Navigator screens (or SDOs)**. The following methods (from stbGeneric) which are described in Section 0 should therefore perform all such checks.

- checkIfAt
- validateAt
- validateNotAt
- detectScreen

For example, if you need to check for the presence of a popup screen:

```
stb.checkIfAt ("POPUP_NO_SATELLITE_SYSTEM")
```

There are only two exceptions to this navigator-verification-only rule where Navigator screens cannot be used. Rectangular coordinates are thus needed for these scenarios (they should be stored in the config file)

### Exception: Verification of Unknown-Location Icons

The method *verifyIcon* is provided in *avUtil* to allow tester to check if a reference image section matches to a certain point on the live screen.

This is useful to verify the presence of icons whose locations can change (usually in a discrete horizontal or vertical way)

### Exception: Verification of Change

The method *verifyKeyChange* is provided in *stbGeneric* to allow a check whereby an area of the screen is checked for change before and after a key press.

This is useful in instances where determining if a screen has changed by reading text is problematic (maybe the screen times out very quickly) or impossible (you want to determine if an event poster has changed). For example to verify navigation within the banner.

## Zap Measurement

A method called *getZapMeasurement* is provided in *stbGeneric*. This measures the response time to a particular button press or series of digits. A black screen must show on the STB at least briefly during the transition period in order for a transition to be detected and timed.

## Tests

### Path to Utilities

At the top of each test the path to the utilities folder must be specified. All of the sample tests included in the framework already provide this (as per the file structure described in Section 0).

```
testdir = os.path.dirname(os.path.abspath(__file__))
sys.path.append(testdir + "\\..\\..\\..\\s3group_framework\\utilities")
```

### Test Layout - Steps

All tests should be partitioned into test steps in a logical manner.

```
#####
# STEP - Video Present
#####

step_name='videoPresent'

step_description=" Video Presence detection"
expected_result="Video is Present"
defineStep(step_name, step_description, expected_result)

stb.checkVideoPresent()

#####
# STEP - Video Motion
#####

step_name='videoMotion'

step_description=" Video Motion detection"
expected_result="Video Motion"
defineStep(step_name, step_description, expected_result)

stb.checkVideoMotion()
```

Steps do not need to be closed explicitly – the current step will be automatically closed in the following 3 situations:

1. A new step is defined
2. The test finishes
3. An exception is thrown and not caught (thus ending the test)



## Step Content

All code in a test script should be quite general. Tests should be able to run on any STB. Therefore, things like explicit button presses or screen names should be kept in the stb files. Therefore, unless it makes absolutely no sense for a particular step, try to contain all / most of the code in an stb method.

```
#=====
# STEP
#=====

step_name="Tune to Next Channel"
step_description="Tune to the next channel"
expected_result="The Next Channel is tuned to"
defineStep(step_name,step_description,expected_result)

stb.pressKeys("Ch+")

new_channel=stb.readChannelNumber()

if new_channel!=old_channel:
    raise TestVerificationError("Channel Change Failed")
```



```
#=====
# STEP
#=====

step_name="Tune to Next Channel"
step_description="Tune to the next channel"
expected_result="The Next Channel is tuned to"
defineStep(step_name,step_description,expected_result)

stb.tuneToNextChannel()
```



Also, try to ensure that the method itself is named in a non-specific way, where possible. E.g. instead of calling a method *changePreferencesScreenDisplayTransparency()* – a name which specifies where the setting in question is found in the menu hierarchy – a better name would be *changeTransparency()*.



## STB Methods

Ideally STB methods should be written to either take all possible considerations into account or in a way that allows the method to be extended easily.

For example, instead of writing a method `changeBannerTimeoutToFive()` which changes the banner timeout to 5sec, a method `changeBannerTimeout(setting)` should be written. This could (depending on time constraints) either support all settings or only the situation where `setting=5` (but in a way that makes it easy to add support for additional settings).

```
def changeBannerTimeoutToFive(self):
    self.navigateTo("SETTINGS_BANNER_TIMEOUT_5")
    self.pressKeys("Green")
    self.validateAt("SETTING_SAVED")
```



```
def changeBannerTimeout(self, setting=5):
    if setting==5:
        self.navigateTo("SETTINGS_BANNER_TIMEOUT_5")
        self.pressKeys("Green")
        self.validateAt("SETTING_SAVED")
```



```
def changeBannerTimeout(self, setting):
    if setting!=5:
        raise TestExecutionError("Only the 5 second setting is supported")
    destination_screen="SETTINGS_BANNER_TIMEOUT_"+str(setting)
    self.navigateTo(destination_screen)
    self.pressKeys("Green")
    self.validateAt("SETTING_SAVED")
```



```
def changeBannerTimeout(self, setting):
    destination_screen="SETTINGS_BANNER_TIMEOUT_"+str(setting)
    self.navigateTo(destination_screen)
    self.pressKeys("Green")
    self.validateAt("SETTING_SAVED")
```



## Test Sections

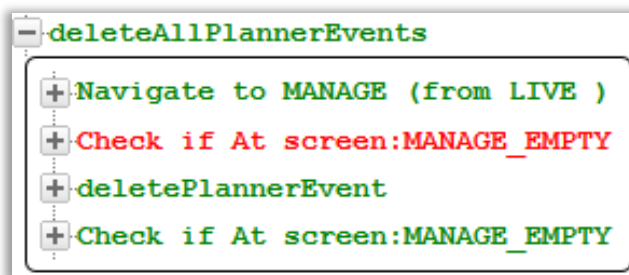
Test sections are useful to segment different parts of the code within a step. It is good practise to enclose each stb method within its own section. Unlike test steps, test sections do need to be explicitly closed (unless an exception is thrown).

```
def deleteAllPlannerEvents(self):
    test_section_name="deleteAllPlannerEvents"
    beginTestSection(test_section_name)

    self.navigateTo("MANAGE")

    while self.checkIfAt("MANAGE_EMPTY")==False:
        self.deletePlannerEvent()

    endTestSection(test_section_name,True,"")
```



## Basic Utility methods

**Almost all** utilities that you will need to call from a test or from another stb method are wrapped in a generic stb method.

The **only other methods** which should be used are listed in Section 0 below.

### stbGeneric

The following are the stbGeneric methods which are used frequently (broken into various categories) – the ones in red are the most commonly used:

Miscellaneous	
pressDigits	Press a series of numerical buttons <code>pressDigits(101)</code>
pressKeys	Press a button or list of buttons <code>pressKeys("Ok")</code>
waitFor	General wait statement: <code>waitFor (milliseconds=50)</code> <code>waitFor (seconds=30)</code> <code>waitFor (minutes=2)</code> <code>waitFor (hours=3)</code>
powerOff	Powers off the DUT (power is unplugged)
powerOn	Powers on the DUT (power is plugged in)
detectMotion	Detects motion over the specified amount of time (in seconds) using the specified navigator screen (which must contain a single motion detect verification) or rectangle coordinates. <code>detectMotion("FULL_SCREEN_MOTION", seconds_to_wait=120, expected_result=True)</code>
getZapMeasurement	Measures the transition time after the given key (or series of numerical keys) is pressed. Note: a black "transition" screen must be present briefly onscreen in order for the transition to be detected. <code>getZapMeasurement (key="ok")</code> <code>getZapMeasurement (key="101")</code> <code>getZapMeasurement (key=101)</code>
verifyKeyChange	Verifies that the specified area of the screen changes after the specified button is pressed. Useful in instances where determining if a screen has changed by reading text is problematic (maybe the screen times out very quickly) or impossible (you want to determine if an event poster has changed) <code>verifyKeychange ( key="Right", rect=(20,40,200,150) )</code>
Navigator	
navigateToInitialScreen	Navigates to the initial screen as specified in the config files. Option to force navigation is True by default so that even if you think you

	are on the LIVE screen (i.e. if that is the current screen that is saved), the start keys will still be used to ensure that you are indeed tuned to LIVE.
updateNavLocation	Updates the saved location to the specified screen. Should only be used if you are certain of the screen you are on.
navigateTo	<p>Navigates between navigator screens. If no source screen is provided, the currently saved screen will be used. Both source and destination screens can be from any of the navigator files specified in the config file.</p> <pre>navigateTo ("SCREEN_B") navigateTo ("SCREEN_B", screen_from="SCREEN_A")</pre>
validateAt	<p>Validate that the specified screen is showing. Throws an exception if it is not showing.</p> <pre>validateAt ("SCREEN_A")</pre>
validateNotAt	<p>Validate that the specified screen is not showing. Throws an exception if it is showing.</p> <pre>validateNotAt ("SCREEN_A")</pre>
checkIfAt	<p>Checks if the specified screen is showing. Returns True if it is showing, False if it is not showing.</p> <pre>checkIfAt ("SCREEN_A")</pre>
detectScreen	<p>Attempts to detect the specified screen for a given amount of time (seconds). The only reason for using this method instead of the more common checkIfAt is that you want to allow a greater time period over which to detect the screen. Returns True as soon as the screen is correctly detected. If the screens is not detected within the specified time, either a False is returned or an exception is thrown (depends on the status of the <i>allow_failure</i> parameter)</p> <pre>detectScreen ("SCREEN_A", 120)</pre>
OCR	
readText	<p>Reads text from the specified screen (the screen must contain at least one OCR verification). Multiple parameter options are available: see method description.</p> <pre>readText ("SCREEN_A")</pre>
readContains	<p>Reads the text from the specified screen (the screen must contain at least one OCR verification) and attempts to find the best match with the specified string / specified string list. If an adequate match is not found, an exception is thrown. If an adequate match is found, the matched item, the matched section of the onscreen string and the percentage match is returned. Multiple parameter options are available: see method description.</p> <pre>readContains ( "SCREEN_A", "SHORT" ) - onscreen text: "this event will start shortly so " - return value: ["SHORT", "short", 100]</pre>

	<pre>readContains ( "SCREEN_A", ["SOON", "SHORT"] )</pre> <ul style="list-style-type: none"> <li>- onscreen text: "this event will start shortly so"</li> <li>- return value: ["SHORT", "short", 100]</li> </ul>
<b>readNumbers</b>	<p>Reads a sequence of numbers from the specified screen (the screen must contain at least one OCR verification).</p> <p>Multiple parameter options are available: see method description.</p> <pre>readNumbers ("SCREEN_A")</pre>
<b>readTime</b>	<p>Reads the time from the specified screen (the screen must contain at least one OCR verification) and is returned as a datetime object (although only the time part is valid).</p> <p>Most time formats are supported (e.g. "05:34", "5.34am", "05;34pm").</p> <p>Multiple parameter options are available: see method description.</p> <pre>readTime ("SCREEN_A")</pre>
<b>readDate</b>	<p>Reads the date from the specified screen (the screen must contain at least one OCR verification) and returns the day (str), date (int) and month (int).</p> <p>Many formats are supported but the day and month must be in text format (e.g. "Thu 04 July", "Thu July 04", "4 Thu July").</p> <p>The config file must contain the list of valid days and months (see sample config file)</p> <p>Multiple parameter options are available: see method description.</p> <pre>readDate ("SCREEN_A")</pre>
<b>Configuration</b>	
<b>getConfigItem</b>	<p>Returns the config item specified by the list of key arguments. Throws exception if the item is not available.</p> <pre>getConfigItem ("general", "POWER_ON_TIME")</pre>
<b>getConfigItemIfAvailable</b>	<p>Returns the config item specified by the list of key arguments. Returns None if the item is not available.</p> <pre>getConfigItemIfAvailable ("general", "POWER_ON_TIME")</pre>
<b>changeConfigItem</b>	<p>Changes the config value specified by the key arguments. This is useful if you need to keep track of a setting which is being changed.</p> <pre>changeConfigItem (240, "general", "POWER_ON_TIME")</pre>
<b>Warning Center</b>	
<b>disableWCResult</b>	Already described in WC section of document
<b>enableWCResult</b>	Already described in WC section of document
<b>setWcResultStatus</b>	Already described in WC section of document
<b>setWcResultScreenshot</b>	Already described in WC section of document
<b>getWcServiceNumber</b>	Already described in WC section of document
<b>getWcSubtestParam</b>	Already described in WC section of document
<b>addWcExtraEvent</b>	Already described in WC section of document

checkServiceAvailableLinear	Checks that a linear service can be tuned to. <a href="#">CAN ALSO BE RUN FROM NON-WARNING-CENTER TESTS!</a>
checkVideoQuality	Checks that the video quality meets requirements. <a href="#">CAN ALSO BE RUN FROM NON-WARNING-CENTER TESTS!</a>
checkAudioPresent	Checks that audio meets acceptable threshold. <a href="#">CAN ALSO BE RUN FROM NON-WARNING-CENTER TESTS!</a>
checkVideoPresent	Checks that a black screen is not present onscreen. <a href="#">CAN ALSO BE RUN FROM NON-WARNING-CENTER TESTS!</a>
checkVideoMotion	Checks that motion exists onscreen. <a href="#">CAN ALSO BE RUN FROM NON-WARNING-CENTER TESTS!</a>

## Other common utilities

Other than the methods defined in `stbGeneric`, the following methods (as well as the methods defined in `miscUtil.py`) may be used directly when writing tests / stb methods.

avUtil	
captureScreen	Captures an image to file (and log) and returns [True, image] <code>ret, captured_image = avUtil.captureScreen()</code>
verifyIcon	Checks if a reference image section matches to a certain point on the live screen. Useful to verify the presence of icons whose locations can change (usually in a discrete horizontal or vertical way) <code>verifyIcon( ref_icon_image, (10,20) )</code>
measureAudio	Checks that audio is present <code>measureAudio (threshold=-50, timeout=5)</code>
setStepStatusFail	Allows the step status to be set to Fail. By default a step that did not throw an exception passes. This method can be useful if you catch the exception from a step and want to continue testing. <code>setStepStatusFail ()</code>
testUtil	
beginTestSection	Starts a test section <code>beginTestSection ("Section A")</code>
endTestSection	Ends a test section <code>endTestSection ("Section A", True, "optional comment")</code>

## Error Handling

The framework primarily depends on exceptions to indicate failures, removing the need for excessive result checking. The following exceptions should be thrown in the case of a failure.

TestExecutionError	Used to indicate a testing error (e.g. missing file, lost connection, etc) <code>raise TestExecutionError ("File not found")</code>
TestVerificationError	Used to indicate a functional verification error (e.g. incorrect behaviour seen, etc) <code>raise TestVerificationError ("Unable to open the Banner")</code> <code>raise TestVerificationError ("TV Guide is empty")</code>

Please adhere to the error-handling style shown in the test templates.

## Warning Center in the Framework

### Warning Center utilities

All Warning Centre (WC) methods that are required in the framework are contained in a handler class called *WcTestHandler* which is defined in the utility file of the same name.

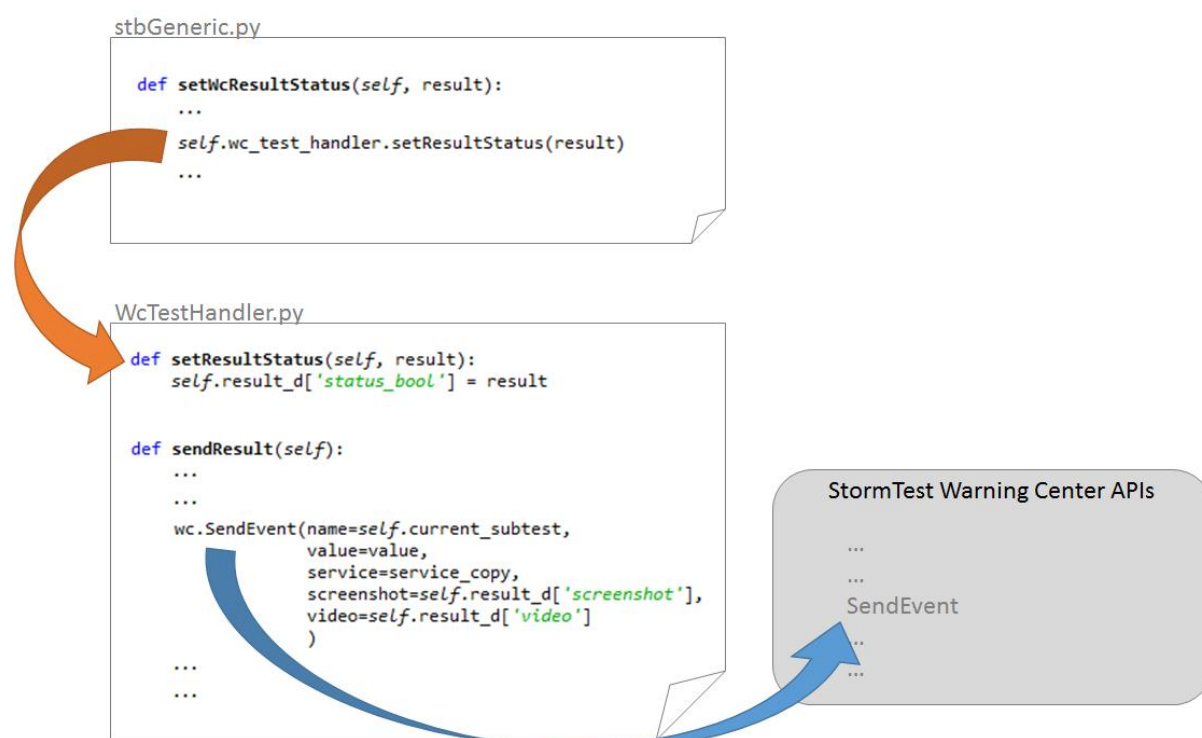
When a test is run, a WC handler object is initialised and saved in the stb object (*self.wc\_test\_handler*). During initialisation the handler attempts to connect to a valid WC portal in order to request the next task (which will be stored in the handler object). Note: If a connection to a valid WC portal does not exist, the placeholder *self.wc\_test\_handler* remains set to *None*.

All WC-related methods are called from the handler object. Here, the WC-related StormTest API calls are wrapped; they should not be called directly. A selection of these handler methods are themselves wrapped in stb methods (in *stbGeneric.py*) and it is only these methods (below) which should be called from tests and/or other stb methods.

setWcResultStatus	getWcSubtestParam
setWcResultValue	getWcServiceNumber
setWcScreenshot	disableWCResult
addWcExtraEvent	enableWCResult

The diagram below illustrates the following examples:

- The stb generic method *setWcResultStatus* wraps the handler method *setResultStatus*
- The handler method *sendResult* wraps the StormTest Warning Center API *SendEvent*





## Automatic Warning Center handling

Most of the Warning Center functionality is built into the framework in such a way as to minimise the amount of WC-specific code that needs to be explicitly called by the developer.

### Event information in Warning Center handler

The WC handler (*stb.wc\_test\_handler*) stores the following information for the current WC event / subtest:

Item	Description
service_d	Service dictionary
current_subtest	Event / Subtest name
result_d	Result dictionary <ul style="list-style-type: none"> <li>• status</li> <li>• value</li> <li>• screenshot</li> <li>• video</li> </ul>
video	The currently-recording video
extra_events_list	optional

The service dictionary *service\_d* is initialised during test initialisation (after communication with the WC portal) while the others are initialised each time a WC event is set up (this will be explained in the next section).

### Using Steps to set up Warning Center Events

In conventional non-WC tests, test steps are used to partition the test into useful sections. This functionality has been extended so that the definition of a step is now also used to define the WC subtest / event which will be performed.

The definition of a test step which does not represent a Warning Center subtest / event (such as the start-up step) is defined as usual with a step name, step description and expected result:

```
step name = "Box init"
step description="Box init - Navigate to Live"
expected_result = "Current screen is Live TV"
defineStep(step_name, step description, expected_result)
stb.testStartup()
```

Steps which represents Warning Center subtests / events should be defined according to the following rules:

1. The **step name** must be set as the name of the WC subtest / event (e.g. “serviceAvailable”, “videoMotion”, “audioPresent”)
2. The **WC handler object** (*stb.wc\_test\_handler*) must be passed into the step definition using the optional 4<sup>th</sup> parameter
3. If the step is **required to pass** in order for the test to continue, this must be specified by setting the optional 5<sup>th</sup> parameter (*require\_pass*) to True. By default, this is set to False, allowing a test to continue if the WC step fails.

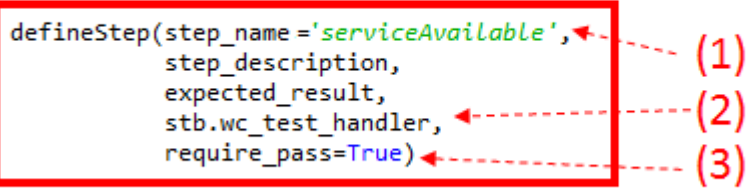
```

step_description=" Channel tune verification, firing event to WC"
expected_result="Channel is available"

defineStep(step_name='serviceAvailable',
           step_description,
           expected_result,
           stb.wc_test_handler,
           require_pass=True)

stb.checkServiceAvailableLinear( )

```



Each time a WC step is defined in this way, the framework automatically resets the handler’s event information as follows:

- **Sets the WC event name** (*current\_subtest*) in the handler object
- **Resets the WC result dictionary** (*result\_d*) in the handler object to have fail values as follows:
  - status = False
  - value = None
  - screenshot = None
  - video = None
- **Starts video capture** (saved to *video* variable)

Once a step has been defined, all WC setup has now been done and the actual testing can commence.

## Using Steps to send Warning Center Events

The steps used by the framework do not need to be explicitly closed – the current step will be automatically closed in the following 3 situations:

4. A new step is defined
5. The test finishes
6. An exception is thrown and not caught (thus ending the test)

Warning Center events are “closed” or “sent” in the same way!

At the point where a step is to be closed, the current WC event (if one has been initialised) will be sent to the WC portal. The following table shows how the event values that are stored in the handler are mapped to the parameters of Warning Center’s *SendEvent* API:

API parameter:	Handler Variable
name	<i>current_subtest</i>
value	<i>result_d</i> [“value”] if non-null: otherwise <i>result_d</i> [“status”]
service	<i>service_d</i>
screenshot	<i>result_d</i> [“screenshot”] if non-null: otherwise a screenshot is taken
video	<i>result_d</i> [“video”] .... The video recording which was started during step initialisation is now stopped and saved to this result dictionary item.

Now remember that when a WC step / event is initialised, the WC handler’s result dictionary (*result\_d*) is reset to contain failure values. Unless these values are updated during the course of the step, the event will thus be sent as a failure when the step is eventually closed. The next section explains the actions a developer needs to take to ensure that the correct result will be sent to the Warning Center portal.

## Developer-driven Warning Center handling

### Updating the Warning Center result during testing

As discussed previously, the WC handler's results dictionary provides the information for sending an event to the Warning Center portal. Since this dictionary is initialised to contain failure values, it must be updated to reflect the result of testing. The following stb generic wrapper methods are used for this purpose:

Method	Parameters	Description
<code>setWcResultStatus</code>	result (True or False)	
<code>setWcResultScreenshot</code>	screenshot (image object)	generated using the following code: <code>status, screenshot = avUtil.captureScreen()</code>
<code>setWcResultValue</code>	Value (depends on event)	Sets the "value" parameter of the result

A simple example of a WC test step is provided below in order to illustrate how and where these methods are used. All WC-related code is indicated using orange stars – notice how little WC "awareness" is needed by the developer!

`test.py`

```

=====
# STEP - Video Motion
=====
step_description=" Video Motion detection"
expected_result="Video Motion"
defineStep(step_name='videoMotion',
            step_description,
            expected_result,
            stb.wc_test_handler,★
            require_pass=False)★

stb.checkVideoMotion()

```

`stbGeneric.py`

```

def checkVideoMotion(self):
    result = self.detectMotion("FULL_SCREEN_MOTION",
                              seconds_to_wait=60,
                              expected_result=True,
                              allow_failure=True,
                              motion_threshold=0.1)

    ret, screenshot=avUtil.captureScreen()
    self.setWcResultScreenshot(screenshot)★
    self.setWcResultStatus(result)★

    return result

```

`test.py`

```

=====
# STEP - Video Present
=====

```

**(1) Set up the WC step / event**

- WC result set to Fail (default)

**(2) Perform testing**

**(3) Update handler's result values**

- WC result set to Pass

**(4) Next step defined => Previous step closed**

- WC event (Pass) sent to WC portal

In the example above a screenshot is explicitly taken and provided to the handler, but this is not absolutely necessary; If the handler has not been provided with a screenshot during the step, one will

automatically be taken before the WC event is sent (at the end of the step). While this is sufficient for simple tests (e.g. testing audio), being able to provide a particular screenshot is useful for more complicated tests where the timing of the screenshot is important. For example, you might want a screenshot to be taken immediately after the channel change, not after you have verified that the channel is correct.

It is also important to realise that since the handler's result is a Fail by default, it is not strictly necessary to update it to reflect a Fail. Likewise, if something causes a step to exit before the result is updated, the WC event **will automatically be reported as a Fail!** For example, in the step above, if an exception is thrown for any reason in the *detectMotion* method, the step will exit and a "*videoMotion*" failure event will be reported to the WC portal.

Therefore, if there is any section of a step whose failure should not be reported as a Warning Center event failure (i.e. a non-critical section from Warning Center's perspective), then this section requires special treatment. The next section explains what needs to be done.

### Non-Critical Processing within a Warning Center step

Any section of a step where a failure should not be reported as a Warning Center event failure needs to be marked as being non-critical by "enclosing" such code using the following methods (from *stbGeneric.py*). If a step exits from within this enclosed section of code, no failing WC event will be sent.

- *disableWCResult*
- *enableWCResult*

The following example illustrates how this should be used - it is ideal for a "*serviceAvailable*" subtest where there will be a certain amount of setup before the actual channel change can take place; In the example below, the EPG needs to be opened and the desired channel selected before an attempt is made at tuning to it. The stars below show the potential failure points – the ones in orange are within the "non-critical" section and thus would result in no WC event being reported (the step would simply close) – the ones in red are outside of the "non-critical" section and thus would result in a WC event failure being reported when the step is closed.

test.py

```
defineStep(step_name='serviceAvailable',
            step_description,
            expected_result,
            stb.wc_test_handler,
            require_pass=True)

stb.checkServiceAvailableLinear()
```

stbGeneric.py

```
def checkServiceAvailableLinear(self):
```

```
    result, screenshot = self.tuneToLinearService(str(service_number))
    model_1.py
```

```
def tuneToLinearService(self):
```

```
    # -----
    self.disableWCResult()
```

```
    # Navigate to the EPG
    result, screenshot = self.openEPG()
```

```
    if result == False:
        return False, None
```

WC Event Disabled

```
    # Navigate to the correct channel
    self.pressKeys(entry_key)
    result = self.checkIfAt(channel_logo_screen)
```

```
    if result == False:
        return False, None
```

```
    self.enableWCResult()
```

```
    # -----
    self.pressKeys("watch")
    ret, screenshot = avUtil.captureScreen()
    result = self.checkIfAt("LIVE", update_nav=True)
```

```
    if result == False:
        return False, None
```

WC Event FAIL

```
    return True, zap_time, final_screenshot
```

```
    if result == False:
        return False
```

```
    self.setWcResultScreenshot(screenshot)
    self.setWcResultStatus(True)
```

```
    return result
```

```
=====
# STEP - Video Present
=====
```

## Secondary Warning Center Events

While it makes sense to represent most WC events as individual steps, there are some events for which this is not conducive. The actions required to generate results for different events may sometimes be intrinsically linked.

For example, events which measure and report the performance of certain functions are inherently tied to the events which report the availability of those same functions. A performance-related event can thus be viewed as being “secondary” to the corresponding availability event.

Example:

Primary “Availability” Event	Secondary “Performance” Event
serviceAvailable	channelSwitchingTime
audioPresent	audioLevel
searchSuccess	searchTime

The measurements required for such secondary events should be performed within the step associated with the primary event. After this is done, the following method (stored in *stbGeneric.py*) should be used to add the “extra” / “secondary” event to the primary event’s step.

Method	Parameters
<i>addExtraEvent</i>	name value screenshot

Multiple additional events may be added. They will be reported immediately after the primary event is reported (i.e. when the step is closed). The following example shows how this is done for the events “*serviceAvailable*” and “*channelSwitchingTime*”:



test.py

```
defineStep(step_name='serviceAvailable',
           step_description,
           expected_result,
           stb.wc_test_handler,
           require_pass=True)
```

```
stb.checkServiceAvailableLinear()
```

stbGeneric.py

```
def checkServiceAvailableLinear(self):
```

```
    result, zap_time, screenshot = self.tuneToLinearService()
```

```
    if result == False:
        return False
```

```
    # Set the main serviceAvailable result and screenshot
```

```
    self.setWcResultScreenshot(screenshot)
```

```
    self.setWcResultStatus(True)
```

```
    # Set the additional accessTime result and snapshot
```

```
    if zap_time>0:
```

```
        self.addWcExtraEvent(name="channelSwitchingTime",
                             value=zap_time,
                             screenshot=screenshot)
```

```
    return result
```

Define the step for the primary event

Perform verification for primary event **AND** measurement for the secondary event

Update handler's result values for the primary event

Add the secondary event



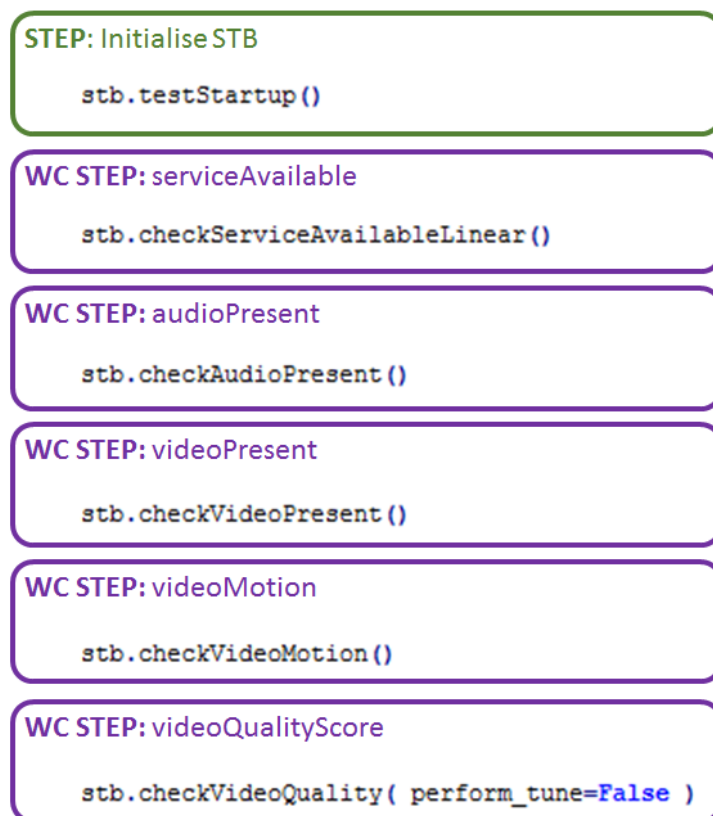
## Warning Center Tests

### Setup of the sample Warning Center Tests

There are two Warning Center tests provided in the framework in the folder *tests.examples/wc*: *linearServiceMonitor.py* and *linearServiceMonitor\_MOS.py*. Both provide basic linear service monitoring by reporting the following WC events:

Event:	linearServiceMonitor.py	linearServiceMonitor_MOS.py
<i>serviceAvailable</i>	✓	✓
<i>channelSwitchingTime</i>	✓ (optional)	✓ (optional)
<i>audioPresent</i>	✓	✓
<i>videoPresent</i>	✓	✓
<i>videoMotion</i>	✓	✓
<i>videoQualityScore</i>	✗	✓

The following diagram shows the outline of the tests:



The initial setup step will depend on each device and thus the method *testStartup* is not provided in the framework and needs to be implemented.

All of the methods which are called in the WC steps are fully implemented and are provided in the *stbGeneric.py* file. Nonetheless, these methods do have a number of dependencies which must be put in place before the tests are run.

All dependencies are outlined as follows:

STB Methods	
testStartup	<p>This method should be implemented in either <i>manufacturer.py</i> or <i>model.py</i>. It needs to put the box into a stable starting state. The following are some examples of the tasks which may be required:</p> <ul style="list-style-type: none"> <li>• Turn STB on (if off)</li> <li>• Remove popups</li> <li>• Navigate to home screen (LIVE)</li> </ul>
tuneToLinearService	<p>A <b><i>tuneToLinearService</i></b> method must be created (in either <i>manufacturer.py</i> or <i>model.py</i>) according to the template provided in <i>stbGeneric.py</i>. This is required for the serviceAvailable step, and also for the videoQualityScore step (when perform_tune=True).</p>
Navigator Screens	
BLACK_SCREEN	<p>A navigator screen called “<b>BLACK_SCREEN</b>” must be created (and its file referenced in the config file); it should return True if a black screen is present.</p>
FULL_SCREEN_MOTION	<p>A navigator screen called “<b>FULL_SCREEN_MOTION</b>” must be created (and its file referenced in the config file); it should return True if motion is detected (verification timeout should not be less than 30 seconds).</p>
Configuration Items	
	<p>These items must be present in the “<i>test_options</i>” dictionary of the configuration file (<i>modelConfig.py</i>); they are provided in the sample configuration file.</p>
PERFORM_ZAP_MEASUREMENT	<p>If True, a zap measurement will be made and reported as a “channelSwitchingTime” event.</p>
AUDIO_TIMEOUT AUDIO_THRESHOLD	<p>Max time to detect audio. Threshold value, below which is considered a Fail.</p>
VIDEO_PRESENT_TIMEOUT	<p>Max time to detect a non-black screen.</p>
MOTION_TIMEOUT	<p>Max time to detect motion.</p>
MOS_THRESHOLD	<p>Threshold value, below which is considered a Fail.</p>

Once all of these dependency items have been created, the sample tests are ready to be run. Remember that the test code must be transferred to Public Files area of the StormTest server.

## Writing your own Warning Center Tests

### Test Scripts

The sample tests which are provided with the framework should be used as templates when creating your own Warning Center test. The structure of each test should be as follows:

```
# === STEP - bring box up =====

defineStep( "Box initialisation",
            "Navigate To Live",
            "Current screen is Live TV")

stb.testStartup()

# === WC STEP - eventA =====

defineStep('eventA',
            "Event Description",
            "Expected Result",
            stb.wc_test_handler,
            require_pass=True)

stb.checkEventA()

# === WC STEP - eventB =====

defineStep('eventB',
            "Event Description",
            "Expected Result",
            stb.wc_test_handler,
            require_pass=True)

stb.checkEventB()
```

## checkXXX Methods

A *checkTemplate* template method is provided in *stbGeneric.py* and this (together with the other check methods) can be used to define any additional *checkXXX* methods which are required. These should be defined in either the manufacturer or model stb files. The outline of the template is shown below, along with the stb method it uses to perform the actual testing. All Warning Center code is shown with a yellow star.

```
def checkXXX(self, param=None):

    if param==None:
        param=self.getWcSubtestParam("myParam")
    if param==None:
        param=self.getConfigItem("test_options","PARAM")
    if param==None:
        raise TestExecutionError("No param provided!")

    result, access_time, screenshot = self.openXXX(param)

    def openXXX(self, measure_access):

        ★ self.disableWCResult()

        self.pressKeys("Backup")
        if self.checkIfAt("BANNER"):
            return False

        ★ self.enableWCResult()

        access_time = -1
        if measure_access:
            rect = self.getConfigItem("regions","XXX_AREA")
            access_time = avUtil.measureTransition("Ok",rect)
        else:
            self.pressKeys("Ok")

        screenshot = avUtil.captureScreen()[1]

        result = self.checkIfAt("INFO")

        return False, access_time, screenshot

    if result == False:
        return False

    ★ self.setWcResultScreenshot(screenshot)
    ★ self.setWcResultStatus(True)

    if access_time > 0:
        ★ self.addWcExtraEvent(name="accessTime",
                               value=access_time,
                               screenshot=screenshot)

    return result
```

(1) Get Parameters

(2) Perform Testing Action

(3) Exit on Failure

(4) Update handler's Result values

(5) Add Extra Event (optional)

The **checkXXX** methods should be written in such a way as to allow them to be used by both WC and non-WC tests. All of the *checkXXX* methods which are provided in *stbGeneric.py* (including the template) allow this. Such dual-usage is achieved in two ways:

### (1) Multiple Parameter Sources

These methods allow parameters to be specified in any of the following ways (in order of preference):

1. Provided as method parameters
2. Provided in some way by the Warning Center test handler (*stb.wc\_test\_handler*)
3. Provided by the config file

Method parameters should be provided for, although they should be set to *None* by default. Where a parameter is not explicitly provided in the method call, an attempt will then be made to get it from the WC handler (using the *stb* wrapper methods *getWcServiceNumber* or *getWcSubtestParam*). If this does not succeed, then the config file will be accessed.

While it is not necessary to provide for all of these options, the methods should not rely entirely on Warning Center information and should at least allow parameters to be passed in with calls to the method. An exception can also be thrown if a value for a parameter cannot be found.

The following examples shows how parameters can be specified:

```
def checkVideoPresent(self, timeout=None):
    if timeout==None:
        timeout=self.getWcSubtestParam("timeout")
    if timeout==None:
        timeout=self.getConfigItem("test_options","VIDEO_PRESENT_TIMEOUT")
    print "checkVideoPresent: timeout =",timeout
    ...
    ...
```

Method parameter (1<sup>st</sup> preference)

Warning Center parameter (2<sup>nd</sup> preference)

Config parameter (3<sup>rd</sup> preference)

```
def checkServiceAvailableLinear(self, service_number=None, measure_zap_time=None):
    if service_number==None:
        service_number=self.getWcServiceNumber()
    if service_number==None:
        raise TestExecutionError("No service number provided")
    print "checkServiceAvailableLinear: service_number =",service_number
    ...
    ...
    ...
```

Method parameter (1<sup>st</sup> preference)

Warning Center parameter (2<sup>nd</sup> preference)

Exception if no value for service\_number exists

Note: the wrapper method *getWcSubtestParam* allows access to any subtest parameters which may be listed in the *task* section of the TestRun which is provided to the test by the WC portal. This is currently empty as standard and so a value of *None* will be returned, thus requiring the next parameter preference to be used.

## (2) Flexible stb wrappers for WC handler methods

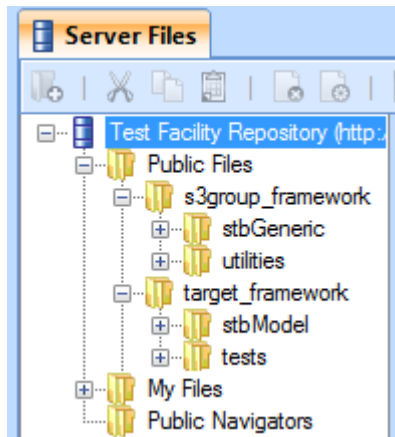
As already mentioned, all Warning Center handling which needs to be done by the developer is done using wrapper methods which are stored in stbGeneric.py. **All of these methods (listed below) can be called regardless of whether a Warning Center test is being run or not** – i.e. if the handler object does not exist, they simply exit.

Thus, a checkXXX method can be written with the intention of being used in a Warning Center test (complete with result updates, disabling of the WC result, etc) but it can then be called in any test, regardless of whether Warning Center is required or not.

setWcResultStatus	getWcSubtestParam
setWcResultValue	getWcServiceNumber
setWcScreenshot	disableWCResult
addWcExtraEvent	enableWCResult

## Running Warning Center Tests

As already mentioned, Warning Center tests required the test code to reside in the Public Files area of the StormTest server.



Once this is in place, the tests can be kicked off from the Warning Center portal as follows:

1. In the Admin/Services tab add as many services as you require
2. In the Admin/Schedules tab add a schedule, selecting the desired services and devices from the options given. To select the test which should be run by, check the "Local" option in the Tests section, select your server and navigate to the required test. The Recurrence section allows you to select how often the test will run. Make sure that the schedule has been enabled (in the General section) and save.

The test should now run periodically as required. You can force an immediate run by clicking on the "Run Schedule Now" button in the schedule list.

Test runs can be view on dashboard as normal. Warning Center events can be viewed on the portal.