Mustella Overview

Mustella

Mustella is a test framework that automates manual testing of the Flex SDK. Whenever changes are made to the SDK, Mustella should be used to validate the changes. This may result in modifications to existing tests or the addition of new tests.

Mustella's goals are:

- Allow a test author to automate the manual test in a way that allows explicit control over mouse positions and frame intervals.
- Create consistency among the tests. Using MXML and a small set of test steps, authors can better understand each others' test scripts since the environment is familiar (versus a test framework that is non-Flex based).

Documentation for Mustella's ActionScript classes can be found at http://people.apache.org/~pent/mustella/asdoc.

Installation

Repository Location
Mustella is currently located at https://git-wip-us.apache.org/repos/asf/flex-sdk.git. Mustella is located in the mustella folder. The latest changes are in the develop branch.

Windows and Cygwin

Mustella may be installed on Windows (using a Unix terminal emulator such as a Cygwin) or Mac OS X. Note that **Cygwin** users should set bash to ignore carriage returns (`\r`) by either creating or editing the bash profile (`~/.bash_profile`) so it contains:

```
export SHELLOPTS
set -o igncr
```

On Windows, this file is located in the following directory:
`<cygwinhome>\home\<your_user>\.bash_profile`

When running the AIR tests on Windows, make sure to set the Windows theme to "Classic." The Mustella AIR test baselines rely on this theme to determine the min sizes and some chrome related values.

Flex SDK and Property Values

Before you can use Mustella you must have a complete Apache Flex SDK ready for use. Chances are you already have the source since that is where Mustella is located. You can check by looking at the Mustella parent directory and finding the `framework` directory as a sibling to `mustella`. If you haven't already done so, build the Apache Flex SDK first (see the **`README`** in the `flex` directory for details).

Important
When building the Apache Flex SDK, you will prompted to accept the license agreement for the Flash Player embedded font support. Without this module, tests that rely on the fonts Mustella uses (see `Assets/Fonts` in the `mustella` directory) will be substituted with system fonts and the tests will fail any bitmap comparisons.

These instructions assume you are running Mustella from within the Flex source tree (its default location). However, you might have checked out Mustella independently from the Flex source, so there are settings you should be aware of.

Several properties Mustella uses are set either in the `env.properties` file of the Flex source directory (see `env-template.properties` for details in that same directory) or set as environment variables. The Mustella directory contains a file called, `local-template.properties`, that contains a number of properties that can be set to determine how Mustella runs the tests. You can copy `local-template.properties` to `local.properties` and Mustella will also look at that file to find property settings.

Out of the box, Mustella does not need `local.properties` as long as Mustella is within the Flex source tree. While you can use `local.properties` to override `env.properties` and environment variables, it is recommended you use `env.properties` or environment variables and use `local.properties` for Mustella-specific values.

- To run any tests that use the AIR SDK (available at http://www.adobe.com/devnet/air/air-sdk-download.html), either set `env.AIR_HOME` in `env.properties` of the Flex SDK, or set the environment variable, `AIR_HOME`, to point to the location of the AIR SDK you are using.

Be sure to use the correct AIR SDK that corresponds to the version of the Flex SDK being tested; at the time of this writing, AIR SDK 3.1 is most appropriate.

- If you are running tests in the debug Flash Player, either set `env.FLASHPLAYER_DEBUGGER` in `env.properties` to the absolute path to the Flash Player content debugger for your platform or set the environment variable, `FLASHPLAYER_DEBUGGER`.
- Mustella also requires `playerglobal.swc`. The location of this platform-specific file is set either with `env.PLAYERGLOBAL_HOME` in `env.properties` or as an environment variable, `PLAYERGLOBAL_HOME`.
- Mustella also requires ANT which can be found at `http://ant.apache.org/bindownload.cgi`. Mustella has been tested with ANT 1.7.0. Once downloaded, set the environment variable, `ANT_HOME`, to point to the ANT installation directory.

Some of Mustella's specific properties you might use (set in Mustella's `local.properties` file) are:

- `use_browser` which, when true, instructs Mustella to use the browser instead of the debug Flash Player. You should also set the corresponding `browser` property to point to the browser application you want Mustella to use.
- `use_apollo` which, when true, instructs Mustella to run tests in the AIR player. When this option is set, the location of the AIR SDK must be specified (see above).

> *At this time the mobile tests have not been vetted but you are welcome to try them out by setting the various `mobile` properties in the `local.properties` file.*

Trust Files

Due to Flash Player security concerns, you should create or modify the Flash Player trust file to include the Mustella tests directory.

The trust file is called, `dev.trust.cfg`, and its contents are the full paths of directory structures containing SWFs that the Flash Player should trust; each directory is on a separate line. The format of the file confirms to the operating system conventions. The location of `dev.trust.cfg` is also different for each operating system:

On Mac OS X, `dev.trust.cfg` is in the `/Library/Application Support/Macromedia/FlashPlayerTrust` directory. Add a line with the full path to the Mustella tests directory; for example: `/Users/`*`your account`*`/apache/flex/mustella/tests`.

On Windows XP, `dev.trust.cfg` is in the `C:\Documents and Settings\YourUser\Application Data\Macromedia\Flash Player#Security\FlashPlayerTrust` directory. Add a line with the full path to the Mustella tests directory; for example: `c:\apache\flex\mustella\tests`.

On Windows 7, `dev.trust.cfg` is in the `C:\Users\YourUser\AppData\Roaming\Macromedia\Flash Player#Security\FlashPlayerTrust` directory. Add a line with the full path to the Mustella tests directory; for example, `c:\apache\flex\mustella\tests`.

You can now run Mustella (see below for more details).

WebServer

Some of the MarshallPlan Mustella tests require that http://localhost point to the Mustella `tests` directory.

Here is how you can set this up on a Mac :

1. Go to settings and turn off Web Sharing if it is on.
2. Edit `/etc/apache2/httpd.conf` (you will need super-user privileges).
3. Find the setting for `DocumentRoot` and change it:

```
<IfDefine WEBSHARING_ON>
#
# DocumentRoot: The directory out of which you will serve your
# documents. By default, all requests are taken from this directory, but
# symbolic links and aliases may be used to point to other locations.
#
#DocumentRoot "/Library/WebServer/Documents"
DocumentRoot "[your-path-to-the-Apache-Flex-SDK]/mustella/tests"
```

4. There is another place to change it, a dozen lines or so below:

```
#
# This should be changed to whatever you set DocumentRoot to.
#
#<Directory "/Library/WebServer/Documents">
<Directory "[your-path-to-the-Apache-Flex-SDK]/mustella/tests">
```

5. Save the file.
6. Go to settings and turn on Web Sharing to restart the server.
7. Enter `http://localhost` into the browser. It should display the contents of the `mustella/tests` directory.

Here is how you can set this up on Windows with Apache:

1. Edit C:\Program Files (x86)\Apache Group\Apache2\conf\httpd.conf
2. Find the setting for DocumentRoot and change it:

```
<IfDefine WEBSHARING_ON>
#
# DocumentRoot: The directory out of which you will serve your
# documents. By default, all requests are taken from this directory, but
# symbolic links and aliases may be used to point to other locations.
#

DocumentRoot "[your-path-to-the-Apache-Flex-SDK]\mustella\tests"
```

3. There is another place to change it, a dozen lines or so below:

```
#
# This should be changed to whatever you set DocumentRoot to.
#

<Directory "[your-path-to-the-Apache-Flex-SDK]\mustella\tests">
```

4. Save the file.
5. Open the Apache Service Monitor.
   If it is not running, start it from here: C:\Program Files (x86)\Apache Group\Apache2\bin\ApacheMonitor.exe
   If it is running, you can find it in the notification area in the Taskbar.
6. In the Apache Service Monitor, restart the Apache2 service.
7. Enter http://localhost into the browser. It should display the contents of the mustella/tests directory.

Here is how you can set this up on Windows 7 with IIS:

1. Type **InetMgr** in the Start menu to Open IIS Manager
2. Select **Sites>Default Web Site** on the left panel
3. Click on **Base Settings** in the right Action Panel
4. Type in or browse to the path to mustella\tests in the **Physical access path** box, then OK.
5. Double-click **Directory Browsing** icon in the IIS section of the main panel to open the browsing options
6. Click on **Enable** option in the right panel
7. Restart the server to make the changes effective
8. Enter http://localhost into the browser. It should display the contents of the mustella/tests directory

The instructions above assumes your server uses default port 80.
If this is not the case, you can pass the port to use to mini_run.sh using the following syntax (eg. for port 8080):
`./mini_run.sh -addArg=-includes=Localhost8080 ...`

Mustella Anatomy

Basics

Mustella is 98% Flex (there is some Java code to manage launching of the tests and such). A "test" is actually two files. One file is the "test SWF" which is a main program MXML or AS file that contains the items to be tested, such as a DataGrid, along with any other data required by the test. The second file is the "test Script" which is an MXML file with `<TestCase>` elements. The items within the `<TestCase>` elements set up the objects defined by the test SWF and then manipulate them by changing their properties or interacting with them. Many tests contain a bitmap comparison with a baseline image to detect changes in the visual aspect. Tests can also contain assertions against the property values to make sure a values remain true to the original values.

To run a test, both the main test file and the test script files are compiled into a SWF and executed. The Mustella mini_run.sh script has a `-rerun` argument that will skip the compilation process, but most of the time the files are compiled with each run.

Directory Layout

The Mustella directory consists of numerous sub-directories, some of which are:

- `tests` - All of the test scripts (MXML files and test-specific assets) are located here. New tests should be placed in the appropriate sub-folders.
- `Assets` - Many tests share the same images, fonts, and data. The `Assets` directory provides a common place for these shared resources. Before creating new assets, you should see if the existing set will meet the needs of your tests. These assets are deliberately generic and should suffice for most needs.

In the `tests` directory you will find a bit of a hodgepodge of test folders loosely grouped by function or feature. Some highlights are:

`apollo` - Apollo was the codename for AIR when it was being developed. You will find tests related to AIR functionality such as using the file system.
`components` and `containers` - the standard Flex MX components.
`mx` - additional tests for the MX components.
`gumbo` - the Adobe codename for the Spark project, this directory contains tests for the Spark components.
`spark` - additional tests for the Spark components.

Mustella tests are comprised of:

1. A test SWF. This should be a simple MXML application, such as an application tag and a single component. Sometimes a set of data and assets are also needed. Other components can be used to prove that the component uder tests can integrate properly.
2. One or more test scripts; a test script is another MXML file.

A Mustella Test Script:

1. Generally references the test SWF it belongs to. Tests are run by specifying the script to run which then determines which test SWF to use.
2. Has the top-level tag of "UnitTester".
3. Has an ActionScript block that allows the test script to register itself with the test engine.
4. Has a set of TestCase tags.

An example of a test script can found in the `tests/components/Button/properties/Button_DataBinding_tester.mxml` file. Examination of this file's root tag shows:

```
<UnitTester testDir="components/Button/properties/"
xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*" testSWF="Button_DataBinding.mxml">
```

The `testSWF` attribute identifies the main MXML file that will run this test script. Main MXML files are always found in the `swfs` folder (such as `tests/components/Button/swfs`).

The test cases are enclosed by the `<testCases>` element.

A TestCase tag:

1. Has a description of what the test does.
2. Has a unique name for the test.
3. Usually has a sub-tag called "setup" where instructions to set set up the test are placed.
4. Should have a sub-tag called "body" where the test sets are placed.

An example of a test case can be found in the `tests/components/Button/properties/Button_Properties_tester.mxml` file; for instance, the first test case tag is:

```
<TestCase testID="Checking_emphasized_true_property_of_Button" description="Test that
emphasized draws a thick border around the button" keywords="[Button, emphasized,
properties]">
```

The "setup" and "body" sub-tags:

1. Have one ore more "test steps".
2. The setup should be a body of a previous test.
3. The setup portion should never generate failures, although it can.

Using the previous example test case, `Checking_emphasized_true_property_of_Button`, the `<setup>` of the test case reads:

```
<setup>
    <ResetComponent target="myButton" className="mx.controls.Button"
waitEvent="updateComplete" waitTarget="myButton"/>
    <SetProperty target="myButton" propertyName="emphasized" value="true"
waitEvent="updateComplete" waitTarget="myButton"/>
</setup>
```

The `ResetComponent` tag replaces the current component with a new instance so as to start the test in a fresh, clean, state.

The `<SetProperty>` tag then sets a specific property on the target component using the `propertyName` and `value` attributes. The name and value must be valid for the component being tested, of course. The remaining attributes wait for specific events to happen on the component before proceeding to the next step.

The body of this test case is contained with the `<body>` tag:

```
<body>
    <AssertPropertyValue target="myButton" propertyName="emphasized" value="true"/>
    <CompareBitmap
url="../properties/baselines/Checking_emphasized_true_property_of_Button.png"
target="myButton"/>
</body>
```

The body elements of the test case methodically test whatever aspect of the component is required. Here, `<AssertPropertyValue>` makes sure that the property, `emphasized`, will have the value, `true`. This tag simply gets the named property and compares the result returned - as a string - to the contents of the `value` attribute. If there is match, the test proceeds to the next step.

`<CompareBitmap>` takes a snapshot of the running test and compares it to a known baseline image. Before running any test for the first that that uses `<CompareBitmap>`, the `mini_run` script should be run with the `-createImages` parameter (see below), to generate the first baseline images; otherwise this test will fail.

Test Steps

There are a set of test steps. The goal is to keep the set small, but new ones can be added if they represent a common test pattern. The set of steps are:

- `ResetComponent` – This test step is often the first step in the setup. It attempts to destroy the old instance of the component under test and replace it with a new one. It tries to clean up things a previous test might have left behind like popups, and will try to wait for a running effect to end before resetting.
- `SetProperty`, `SetStyle` and `SetURL` – These allow the test author to manipulate public APIs of the component under test.
- `ChangeState` – Changes the state of a component or application.
- `DispatchMouseEvent`, `DispatchKeyEvent`, `DispatchEvent`, `DispatchMouseClickEvent` – These allow the test author to test the component's event handling.
- A set of `Assert` subclasses. These are used to verify the component did what was expected. There is:

    - `AssertEvent`, `AssertNoEvent`, `AssertEventPropertyValue`-- These make sure a component did or did not dispatch an event with specific properties
    - `AssertError` – Makes sure an error gets thrown.
    - `AssertPropertyValue`, `AssertStyleValue`, `AssertURL`, `AssertTitle`, `AssertType` – These are used to test the values in public APIs of the component under test.
    - `AssertMethodValue` – Makes sure a method returns the expected value.
    - `CompareBitmap`, `AssertPixelValue` --These are used to validate that the display list is correct. It is probably over-used in the current set of test media, but it is more convenient that testing tons of properties to validate that a popup isn't still overlaying the component.
- `WaitForEffectsToEnd`, `WaitForEvent`, `WaitForSandboxApp`, `WaitForWindow` – These cause execution of the script to pause until it is ready to continue.
- `Pause` – This causes the test to pause for the indicated amount of time. Use of Pause is discouraged since it creates timing dependencies. There really should be an event or sequences of events to wait for. But you will see it over-used as well in some tests.
- `RunCode` – This causes some code to run. It should only be used if there is no combination of test steps to perform the same task, but it is also over-used in the current test media. It runs the code in the context of the script instead of the application. Ideally, every once in a while we'd look at all the RunCode usage and determine from there if a new test step needs to be added; for example, one that adds and removes children from a parent.

Every test step has a "waitEvent" property. If there is no waitEvent, the test engine immediately executes the next test step. If there is a waitEvent, the test engine sets up a listener for that event and pauses which allows the frame to end and the player to render. Essentially, waitEvents are used to control when the frame updates are allowed in the test.

Using Mustella

Help

```
./mini_run.sh
```

With no arguments, `mini_run.sh`, produces a list of arguments and usage information.

Running All Tests

```
./mini_run.sh -all
```

Use this shell command to execute all of the tests in the `tests` directory. This will take a number of hours to complete. Failures are noted but only severe errors will stop the tests from completing.

The positive results are recorded in the `results.txt` file and failed tests are recorded in the `failures.txt` file. Should any tests fail during `-all` run, use

```
./mini_run.sh -failures
```

to run just the failed tests over. See the section, "Re-running Failed Tests", below, for more tips.

Running Test Groups

```
./mini_run.sh tests/components/DataGrid
```

This command runs all of the tests in the `components/DataGrid` directory. You will find the results in `results.txt` and failures in `failures.txt`. The console output contains more details however.

Running Specific Tests

```
./mini_run.sh -caseName=datagrid_properties_columns_0
tests/components/DataGrid/DataGrid_SparkSkin/Properties/datagrid_properties_columns.mx
ml
```

This commands runs a single test case as given by the `caseName` option. You must also specify the file in which the test is defined in order to run a single test case.

Re-running Failed Tests

If you find some tests are failing, they are reported in the `failures.txt` file. Once you fix some or all of them, you can quickly re-run just those tests that failed using the `-failures` option:

```
./mini_run.sh -failures
```

To save some time, you can include the switch, `-rerun`, which will not re-compile the tests. However, `-rerun` has some caveats:

- If only one test case fails in test file, `-failures` by itself will re-compile the test file and run only that one test. Adding `-rerun` avoids the re-compile, but all of the tests, including the failed test, will run. That is, if there are 10 test cases in a file and 1 case fails, adding `-rerun` will run all 10 test cases.
- If the test case that fails has failed with "Failed timed out", no other test cases in that test file were ran. Using `-failures -rerun` will run all of the test cases.
- If the error has the name of the SWF (not the test case), `-failures -rerun` will definitely run all of the test cases in SWF.

Baseline Images

Many tests require a comparison of the screen to a baseline image in order to see if the test is successful. If your tests use the `CompareBitmap` or `AssertPixelValue` tests, you will need to generate an initial baseline image for subsequent runs of the test to use. Generating baseline images is done with the `-createImages` parameter to `mini_run`.

```
./mini_run.sh -createImages -caseName=Checking_emphasized_true_property_of_Button
tests/components/Button/properties/Button_Properties_tester.mxml
```

The above command runs a specific test case and generates a new baseline image (or images) for the test, storing the image in the `tests/components/Button/properties/baselines` directory.

```
./mini_run.sh -createImages tests/components/Button
```

The above command generates baseline images for **all** of the test cases within the `tests/components/Button` directory structure, which could be hundreds of images. Note that test cases may still fail, just not for bitmap comparisons.

Once you generate images, be sure to re-run the tests without the `-createImages` switch to verify the image comparisons work and no errors are found as a result of bitmap-compare problems.

AIR and Browser Testing

Tests are, by default, run in the standalone Flash Player, but you can also run tests in AIR and in the browser.

```
./mini_run.sh -apollo tests/components/Button
```

The command above will compile the test with the AIR SDK and run the test using AIR (AIR was codenamed, "Apollo"). Optionally, you can set `use_apollo` to true in the `local.properties` file as described above.

```
./mini_run.sh -browser tests/components/Button
```

Use the `-browser` option to run the tests in the browser. Optionally, you can set `use_browser` to true in the `local.properties` file as described above.

The `mini_run.sh` has a number of other options; see the `mini_run.sh` for more details on those options.

Test Time-Out Results
If you find that tests are failing due to a time-out, it could be due to a stuck `LocalConnection`. Mustella uses `LocalConnection` to force garbage collection and occasionally `LocalConnection` hangs and you have to reboot or shutdown all instances of the player (including the browser, instant-message programs and chats, and anything else that might have the Flash Player embedded). Once you done that you will see better results.

Excluding Tests

Sometimes a test will work on one operating system but not on another. If you find such case, the test can be excluded for that particular operating system. The exclusion list is operating system dependent (due to the syntax of file path names) and is found in the `mustella/tests` directory. The `ExcludeListMac.txt` and `ExcludeListWin.txt` files contain lists of tests that will be skipped. Each file has the same format:

`path-to-test-script$test-case,`

For example, in the `ExcludeListMac.txt` file you might find:

`SkinStates/Styles/SkinStates_Styles$Test_DateChooser_Skin_normal_gif_embeded,`

A dollar-sign ($) separates the test script from the test case and the entry ends with a comma (,). Note that the test script path begins in the `tests` directory and does not include the file extension.

Repairing Tests

> *This section is temporary until all current tests pass. Mustella was recently donated and not all tests are passing. This section provides information on getting existing tests to pass.*

Examples

In order to donate Mustella, images and fonts were changed. All fonts were replaced with OFL fonts and images were replaced with generic versions. When these changes were made, a large number of tests failed. The tests failed either because the original bitmap snapshot of the test no longer matched or the size of the components changed (making width/height or x/y tests invalid).

While effort was made to correct as many tests as possible to account for the changed (including new bitmap images), you may see a failure that says something like, "returned 316 expected 250" or "returned 0xFF0000 expected 0x098C2F". In both cases the remedy is to change the test statement to reflect the new values.

Here is an example, most likely the result of replacing the original fonts with the OFL fonts.

```
gumbo/components/RichEditableText/Integration/RichEditableText_sizing_test1
RichEditableText_no_text_fixed_width Failed AssertPropertyValue(body:step 1)
wOnly.height 11 != 12
```

To change the test, edit the file, `gumbo/components/RichEditableText/Integration/RichEditableText_sizing_test1.mxml` and locate the test case, `RichEditableText_no_text_fixed_width`. Look in the <body> section of the test. Since the error is reporting `body:step 1`, look at the first body test which is the `AssertPropertyValue`. The error reports `wOnly.height` and in the `AssertPropertyValue` the target is `wOnly` with property `height`. The value expected is `11`, but with the changes to the fonts and other test data, the new value is `12`.

```
<AssertPropertyValue target="wOnly" propertyName="height" value="11" />
```

becomes

```
<AssertPropertyValue target="wOnly" propertyName="height" value="12" />
```

The errors reported give the file, the test case, the test step, the expected value and the returned value. By reconciling the test case with the new data, the tests will pass. Once you change a test case, re-run the test (see above about running specific tests or use the `-failures` option to `mini_run.sh`) since Mustella will stop a test on the first step that fails and other steps may still fail; you have to repeat running the tests until they all pass.

Tests that rely on specific mouse locations may also need correction. The change in fonts can increase or decrease the size of a component and move the location being tested against; even a single pixel change can cause a test to fail. Another possibility is that the mouse location is no longer visible due to a size change. For example, many of the DataGrid tests are located in containers that have a specific size. If the DataGrid is larger than it was originally, the DataGrid (or part of it) may no longer be visible because it has gone outside of the container's view port (the container will now have scrollbars). To correct this, increase the size of the container to completely expose the component being tested.

If a test is failing a bitmap comparison, be sure to use the `ImageDiffAIR` program that comes with Mustella. You will find it in `mustella/as3/src/mustella/ImageDiffAIR.air`. Just open the `ImageDiffAIR.air` file and it will install automatically. This program can help you identify why the bitmap comparison is failing. Follow these steps to use the program once it launches:

1. Select the directory of baseline images, such as `tests/components/TabBar/Styles/baselines` and pick the `FindImages` button at the bottom.
2. The next screen will display all of the `*.bad.png` files generated from the test run - these are the images that were just made and fail to compare with the original baseline images.
3. Select one or more of them and pick the `Compare Selected Images` button.
4. On the next screen you will see both the bad and baseline image.

You will have to use your judgement to determine if a new baseline needs to be made (using the `-createImages` option to `mini_run.sh`) or the test needs to be altered in some way.

If you see a bad bitmap that has the "broken image" in the test bitmap, it means the image in the test could not be found. Images used to live in a directory within the test suite but were standardized and moved to the `mustella/Assets/Images` directory. Consequently, relative paths to the images were changed and may not be completely correct. Here is an example in `tests/components/TabBar/SWFs/comps/button_jpg_runtime.mxml`. In this skin class you will see `<mx:Image>` elements pointing to the images:

```
<mx:Image source="../../../../Assets/Images/ButtonImages/buttonOver.jpg"
maintainAspectRatio="false" width="100%" height="100%"/>
```

Note that the path to the image is relative to the `SWFs` directory and not to the component skin directory; this is different than the embedded images since that path is relative to the component. Just be sure to check to make sure the image exists and the path is correct.

If you cannot repair a test or feel the test is in error in some way, add it to the exclusion lists (see above, Excluding Tests).

Q & A

1. What will I see on the error log if there is a time out failure?

It depends on what kind of timeout. Here is a step timeout from the console output:

```
[java] gumbo/components/RadioButton/properties/FxRadioButton_properties
FxRadioButton_select_icon Failed DispatchMouseClickEvent(body:step 1)
            Timeout waiting for updateComplete from srg.s1
```

It tells you what step number failed and what event it was waiting for.

There are a couple of reasons for SWF timeouts:

- An uncaught exception is thrown. In this case the stack trace is in the `.log` file for the SWF.
- A really slow SWF. Sometimes a SWF is creating tons of widgets up front and can't get output to the test engine in time. Other times, the SWF doesn't have the correct fonts embedded and lots of warnings go into the log and that delays getting output to the test engine.

2. Is there a way to check that the Embedded Fonts are working in Mustella?

There is a `CheckEmbeddedFonts` mixin, but it currently only works for MX components. You can look at the log to see if warnings are going in there. You can tell generally because the fonts are Times New Roman in Spark if the font isn't embedded or it will show up in the debug console when the SWF is run.

Excluding Tests

The "Excluding Tests" section above described how to exclude tests based on operating system. For repairing tests, you may need to set aside a test for other reasons, such as when an error keeps happening and the fix may take a while. This is the recommended way to skip tests rather than using the Exclusion list files.

1. Locate the `<TestCase>` you want to exclude.
2. Use XML comments to remove it from being handled: `<!`~~`- <TestCase>`~~ to `</TestCase> ->`.
3. Include in the comment the string, FIXME, along with a short note as to why the case is being excluded.

```
<!-- FIXME: This case relies on a remote server
<TestCase ...>
...
</TestCase> -->
```

Later, all of the excluded cases can be easily found by searching for the "FIXME" string.

Debugging

Debugging with FDB

Tests can be debugged using `fdb` but it is tricky. Here are the basic steps to use `fdb` to set a break point on a specific line in file (there are other ways to set break points, this is the easiest):

Start by building a SWF based on the test case (or cases) you want to debug. For example, if you want to debug test case "Spark_condenseWhite_true__property_Label" in tests/components/Label/properties/Label_properties_tester_Spark.mxml, generate the SWF with this specific test case using the `-caseName` option to mini-run:

```
$ ./mini-run.sh -caseName=Spark_condenseWhite_true__property_Label
tests/components/Label/properties/Label_properties_tester_Spark.mxml
```

This will generate tests/components/Label/SWFs/Label_main_Spark.swf because the `testSWF` option of the Label_properties_tester_Spark.mxml names this as the main application test SWF; the SWF contains only the code to run the specific test. Now you can debug it, putting the path to the SWF as the argument to `fdb`.

```
$ fdb <path to>.swf
Apache fdb (Flash Player Debugger) [build 0]
Copyright 2012 The Apache Software Foundation.
Attempting to launch and connect to Player using URL <path to>.swf
Player connected; session starting.
Set breakpoints and then type 'continue' to resume the session.
(fdb) b <file name>.<ext>:<line #>
Breakpoint 1 created, but not yet resolved.
The breakpoint will be resolved when the corresponding file or function is loaded.
(fdb) continue
[SWF] <path to>.swf - #,###,### bytes after decompression
Additional ActionScript code has been loaded from a SWF or a frame.
To see all currently loaded files, type 'info files'.
Resolved breakpoint 1 to <function name> at <file name>.<ext>:<line #>

Set additional breakpoints as desired, and then type 'continue'.
(fdb) continue
...
... do whatever you need to do to hit your breakpoints
...
```

At this point you interact with the SWF and fdb will stop when one of your break points is hit. You can then see the contents of variables, set additional break points, single-step through the code, etc.

Debugging with FlashBuilder

- **Generate the SWF for the test**
  - run the mini_run script like this from the commandline/cygwin:

    ```
    $ ./mini_run.sh -caseName=datagrid_properties_columns_0
    tests/components/DataGrid/DataGrid_SparkSkin/Properties/datagrid_properties
    _columns.mxml
    ```

    This will create the DataGridApp.swf file in the SWFs folder

- **Create a Web Flex Project** called MustellaDebug.
  - It does not matter which version of Flex compiler you chose.

-
  - Add the Mustella test sources to the project
    Right click project > Flex Build Path > Source path
    Add the path to the Mustella test directory you want to debug. In this example, I will use this path: <my path>\git\flex-sdk\mustella\tests\components\DataGrid\DataGrid_SparkSkin
    path>\Properties

For this example, I will chose the test:
tests/components/DataGrid/DataGrid_SparkSkin/Properties/datagrid_properties_columns.mxml

-
  - Disable "Generate HTML wrapper", so that you can point directly to the SWF

- **Create a run/debug configuration**
  - Under the debug menu, select 'Debug Configurations...' item
  - Create a new 'Web Application' run configuration
  - On the right side, select your MustellaDebug project and the default Applicaton mxml file based on the MustellaTest app.
  - For "URL or Path to launch", uncheck the Úse default option and browse to the SWF that has been generated in step 1.

- **Set breakpoints** in the source code
  - In Flash Builder, Open the file

```
[source
path]DataGrid_SparkSkin/Properties/datagrid_properties_columns.mxml
```

    and set a breakpoint on Line: 65
  - Open the file

```
[source path]DataGrid_SparkSkin/SWFs/DataGridApp.mxml
```

    and set a breakpoint on Line: 530

- **Debug**
  - Debug the RunConfiguration you have just created, by select Debug Configuration.. then Debug.

- If you make any changes to the tests code, you need to run mini_run.sh again
- If you make any changes to the SDK, you need to update the SWF, then start mini_run.sh again.

Debugging with IDEA

- Generate the test SWF:
  Start by building a SWF based on the test case (or cases) you want to debug. For example, if you want to debug test case "Spark_condenseWhite_true__property_Label" in tests/components/Label/properties/Label_properties_tester_Spark.mxml, generate the SWF with this specific test case using the `-caseName` option to mini-run:

```
$ ./mini-run.sh -caseName=Spark_condenseWhite_true__property_Label
tests/components/Label/properties/Label_properties_tester_Spark.mxml
```

- Setup a mustella Flash Module in your sdk project
  - Add the sources of the test to debug in the module source folders
  - Check the "skip compilation" option as the SWF is compiled separately
  - Uncheck Use HTML Wrapper
  - the other compiler options, including the SDK, can be set to anything you like

- Setup a run configuration for the mustella test
  - Select "URL or locale file" and browse to the sWF that was generated in the mini-run

- set breakpoints in the source code
- debug the run configuration above

Note: you can define more than one run configuration if you want to debug several mustella tests.

Running Mobile Tests

Running the mobile tests (see `tests/mobile` directory) is very similar to running normal tests, but are there are few settings that are needed. At this time, running the tests on the desktop emulator works, but running on an actual device is not yet working.

Just as with desktop and browser tests, mobile tests run a Flex application, examine values and compare bitmaps. Today, tests run in the desktop emulator, which is `adl`, but sized to correspond to popular mobile devices with various pixel densities. Keep in mind that the tests are not testing the capabilities of a device, or even of the operating system on that device, but rather the capabilities of the Apache Flex SDK. For example, even if a mobile operating system can do X, if that feature is not exposed through the Flex SDK, it cannot be tested. Likewise, if a feature is exposed through the Flex SDK but is not available on the device, that feature cannot be tested.

Put another way, the Apache Flex SDK has components that use features found in AIR. AIR provides the interface to the device and its operating system. The tests are designed to verify that the Flex SDK works and if it uses a feature of AIR that is not implemented by the AIR pointed to by AIR_HOME (or the equivalent environment or property values) you need to change the value of AIR_HOME.

`local.properties`

Make a copy of `local-template.properties` called `local.properties` and edit it to enable mobile testing as follows:

`target_os_name=android` Tests are only known to work for android. ios and qnx have not been worked on.
`android_sdk=location of your Android SDK`
`runtimeApk=location of the AIR Android runtime, typically AIR_HOME/runtimes/air/android/emulator/Runtime.apk`
`device_name=mac` or `device_name=win`

You can also set these two values if you don't want to keep adding the -mobile flag to mini_run.sh

```
use_apollo=true
run_mobile_tests=true
```

The `local.properties` file includes a list of `device_name` values. Set `device_name` to either `mac` or `win`, depending on the type of machine you are using. This sets Mustella to launch the desktop device emulator

You can also need to enable one of the alternate `adl_extras` but the default is (and tests are only known to work for):
`adl_extras=-screensize 640x960:640x960 -profile mobileDevice -XscreenDPI 240`

These parameters are passed to adl to size the emulator and set its screen DPI.

Baseline Images

The combination of `device_name` and `adl_extras` produces the file name for the baseline images. For example, @mac_240ppi, will be appended to the baseline image file name provided that there is a matching `<ConditionalValues>` such as:

```
<ConditionalValue deviceDensity="240" os="mac"/>
```

The baseline image filename will be @mac_240ppi because a) `mac` matches the `device_name` in `local.properties` and b) 240 matches the `screenDPI` from the `adl_extras` setting in `local.properties`.

Targeting a Different OS

When the emulator is used the device and operating system are based on the `device_name` setting in `local.properties`. You can however, target a different system when comparing (or generating) bitmaps. Add:

`target_os_name=android` or `target_os_name=ios`

to `local.properties` and the bitmaps used for comparison will be named using this OS, such as `@android_240ppi`, rather than the OS associated with the `device_name`. This enables you to run the mobile tests using the emulator but compare bitmaps for either an Android or iOS device. You must have matching `<ConditionalValue>` elements in the test cases as well as an appropriate set of `adl_extras` in `local.properties`. This table should help:

Bitmap Suffix	device_name	target_os_name	adl_extras	Conditional Value
@ios_320ppi	mac or win	ios	`adl_extras=-screensize 640x960:640x960 -profile mobileDevice -XscreenDPI 320`	`<ConditionalValue deviceDensity="320" os="ios" />`
@android_240ppi	mac or win	android	`adl_extras=-screensize 640x960:640x960 -profile mobileDevice -XscreenDPI 240`	`<ConditionalValue deviceDensity="240" os="android" />`
@mac_160ppi	mac	N/A	`adl_extras=-screensize 320x455:320x480 -profile mobileDevice -XscreenDPI 160`	`<ConditionalValue deviceDensity="160" os="mac"/>`
@win_320ppi	win	N/A	`adl_extras=-screensize 640x960:640x960 -profile mobileDevice -XscreenDPI 240`	`<ConditionalValue deviceDensity="320" os="win"/>`

Testing a release

To test a release candidate, or to fully test a branch or trunk, download the release candidate or source and build it. Make sure you opt in for all options, especially the embedded font handling.

If it is a release candidate, set the FLEX_HOME environment variable to point to the release candidate's frameworks directory. Set the sdk.dir in the mustella/local.properties to point to the release candidate. If you are testing a branch or trunk, you do not need these two settings.

Make sure you have followed the steps for running mobile tests.

Then:

1. run mini_run.sh -all This will take about 10 hours so best to do overnight.
2. run mini_run.sh -apollo tests/apollo This will run all of the AIR specific tests.
3. run mini_run.sh -mobile tests/mobile This will run all of the mobile tests.

The above three steps will run every mustella test we have. If everything passes, then the release is in good shape.

Tips

1. Try to use generic assets whenever possible. The top-level Mustella directory, `Assets`, has a number of fonts, images, and data that are generic enough to be used for most testing purposes. If your test does require specific data, place that within the test directory itself.
2. Keeps tests specific. Overly complex tests are hard to maintain and can easily break down if the components being tested change too much. Have more tests that are specific to a component or, for complex components, specific to a component's features. For example, the `DataGrid` component has many, many tests to check its many, many features. Should a specific feature be changed, only those tests that are checking that feature need to be modified (but of course, you would run all of the tests to make the sure the changes didn't adversely affect other parts of the component).
3. When analyzing test failures, note that the failure text gives you the test name and the result of the comparison ("received X expecting Y"). The "Y" in the message will be on the test case tag ("`AssertProprtyValue`" for example); the step in the `<body>` failed is also noted. Open the test MXML file and look at the `unitTest` attribute of the root tag to determine the SWF that was run.
4. Tests that receive timeout failures are most often due to tests that make use of remote services; the timeout happens when the URL to the service is no longer valid or the service is simply not running. If you expect network latency to be the cause, use the `mini_run.sh` parameter, `-timeout=`*`milliseconds`* to increase the wait-time for all of the tests being run.

Mustella ASDoc Generation

As stated above, the current ASDoc has been generated and can be found here: http://people.apache.org/~pent/mustella/asdoc. Follow these instructions if you make changes to Mustella's ActionScript classes and need to generate new ASDoc files.

The Mustella ActionScript sources are in the as3 sub-folder and the ASDoc output is in the `asdoc-output` folder of the `mustella` directory. Here is how to generate that documentation:

```
cd as3/src
asdoc -doc-sources mustella -output ../../asdoc-output/ -load-config
../../../frameworks/air-config.xml -source-path mustella -main-title "Mustella
ActionScript Classes"
```

Note that the `asdoc` command uses the `air-config.xml` due to the AIR classes referenced in the source.

Utilities

There are a couple of utility programs to help you avoid having to run 35000+ tests over several hours in order to prove you didn't break anything with your changes. These utilities are in the utilities folder. You should build them yourself.

MustellaDependencyDB

This utility requires that you have previously run mini_run.sh -all. You don't really have to wait for all of the tests to run, you can just get through the compile phase (probably two or three hours of compilation). This utility reads the link-reports of all of the compiled SWFs and builds up a database of which test files potentially use which SDK source files. Naturally, this is going to include some test files that have nothing to do with a source file because of some dependency issues, but better safe than sorry.

MustellaTestChooser

This utility requires that you have run MustellaDependencyDB. After you have a set of changed files, run MustellaTestChooser and copy and paste the GIT status into the utility and it will write out a list of scripts to be run that use the modified files. GIT status can show lots of unversioned files so it might be easier to pipe it to a filter via:

```
git status -s | grep "^[ M]"
```

Copy and paste the results into MustellaTestChooser's UI, be sure to remove the eventual white space on the left of the M and hit the button. It will write out a file that mini_run.sh looks for with the list of test files to run.

Then run "mini_run.sh -changes" and it will run just those tests. If everything passes, then you know you haven't broken any mustella tests.

Mustella Journal

12 Nov 2012
Mobile tests now run on desktop emulator. Bitmaps can be generated and compared for desktop emulators and devices. Tests still need to be adjusted (due to use of generic images and fonts, or other changes) but they can be run. Launches tests on actual devices is not working yet.

23 Oct 2012
Directory paths with spaces are now allowed. This includes `FLEX_HOME` as well as the path to the Mustella directory. Paths with spaces below the `mustella` directory are not allowed.

18 Oct 2012
We have finished getting all tests in the default configuration (mini_run --all) to pass on Mac and Windows platforms! That is currently 35729 tests. You can start making changes to the SDK and ensure you didn’t break anything.

Still to do:

- Creating recommended test suites so you do not have run large numbers of tests just to check your work.
- Getting the mobile tests to run without errors and on devices.
- Removing (or reducing) pauses within the tests. When added up, the total length of time the entire test suite pauses is an hour!