ULC
Test Framework
Guide

Canoo RIA-Suite 2014 Update 2

canooria-suite

Canoo Engineering AG
Kirschgartenstrasse 5
CH-4051 Basel
Switzerland
Tel: +41 61 228 9444
Fax: +41 61 228 9449
ulc-info@canoo.com
http://riasuite.canoo.com/

DISCLAIMER OF WARRANTIES

This document is provided “as is”. This document could include technical inaccuracies or typographical errors. This document is provided for informational purposes only, and the information herein is subject to change without notice. Please report any errors herein to Canoo Engineering AG. Canoo Engineering AG does not provide any warranties covering and specifically disclaim any liability in connection with this document.

TRADEMARKS

Java and all Java-based trademarks are registered trademarks of Sun Microsystems, Inc.

All other trademarks referenced herein are trademarks or registered trademarks of their respective holders.

Contents

ULC Test Framework Guide

1 Overview

1.1 Introduction

The ULC test framework is a tool that enables functional testing of ULC applications. It shields the test developer from all ULC specifics. Using this framework, test developers can invest all of their time into writing tests and do not need to invest time in technical ULC test issues.

The ULC test framework integrates into the JUnit test framework. Therefore, ULC functional tests are written in Java. It uses the popular and mature Jemmy library on top of JUnit to interact with the user interface of an ULC application.

An ULC functional test is written as a repeated sequence of the following actions:

1 Identify a user interface element with help of some search criteria, e.g. the text on a button, the title of a window, the index of a user interface element in a container:
   - Input field inside the login dialog (title *Login Dialog*) that holds the login name of a user (*first* input field in dialog).
   - *Login* button inside the login dialog (*Login* button text).

2 Trigger a user interactions on the identified user interface element, e.g.
   - Type *Alice* into an input field.
   - Push button.

3 Identify another user interface element, e.g.
   - Label inside the login dialog that holds the status text (*second* label in dialog).

4 Check for a certain condition on the identified user interface element, e.g.
   - Assert that the status label displays *Wrong password*.

The ULC test framework provides an easy-to-learn API that covers all aspects of writing functional tests for ULC applications.

You can find examples in the sample/testframework directory of your ULC installation.

1.2 Requirements

- ULC
  The test framework is part of the ULC release from version 6.2 onwards.
  http://www.canoo.com/ulc/
- JUnit 3.8, a test framework
  http://www.junit.org/
- Jemmy, a user interface test framework
  http://jemmy.netbeans.org/
- Jetty, a Java web server
  http://www.mortbay.org/
- Java Development Kit
  The test framework needs at least JDK 1.4.2.
  http://java.sun.com/javase/

2 Installation and Creating a Sample Test

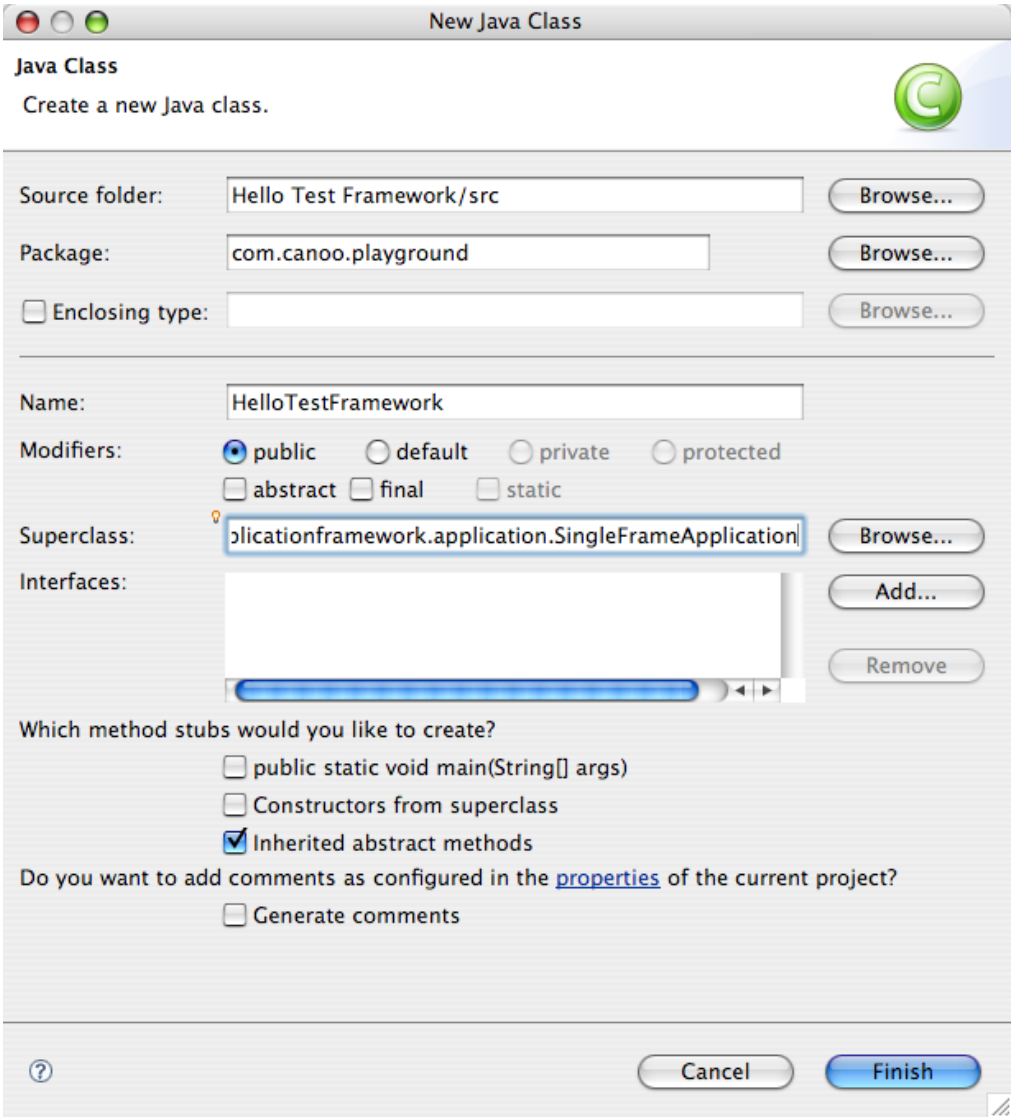
This chapter describes the installation of the ULC test framework with Eclipse and how to create a sample test for an ULC application. The same mechanism applies to other IDEs.

You can use an existing ULC project to install the ULC test framework or you can start with an empty project. The following section describes how to create an empty ULC project and add a simple ULC application to that project. If you want to start with an existing ULC project you can skip the following section.

2.1 Create a Sample ULC Project

- Use the **New Java Project** wizard to create a new Java project.
- Select **Properties** in the context menu of the newly created project to open the project properties dialog.
- Select **Java Build Path** in the list on the left side to display the Java build path configuration.
- Select the **Libraries** tab to configure the Java build path.
- Click **Add External JARs…** and from the *<ULC_HOME>/all/lib* directory add the *ulc-core-development.jar* ULC library and all the jar files that do not start with "*ulc-"* to the Java build path.

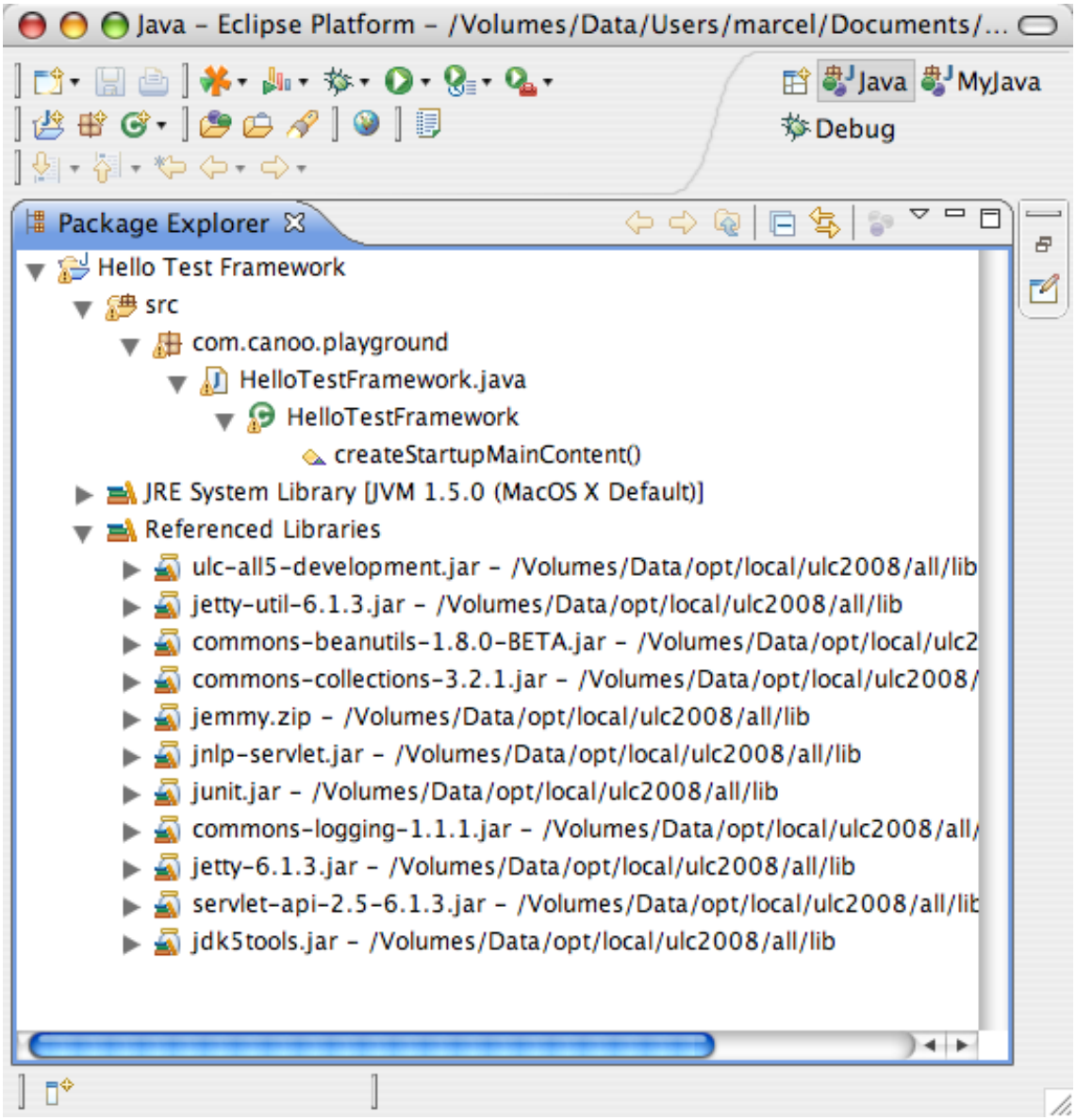
- Use the **New Java Class** wizard to create a new ULC application.
  Make sure that the new Java class extends `SingleFrameApplication`.



- Add application code for the `HelloTestFramework` application to the `start()` method.
- Your Java file should now look as follows:

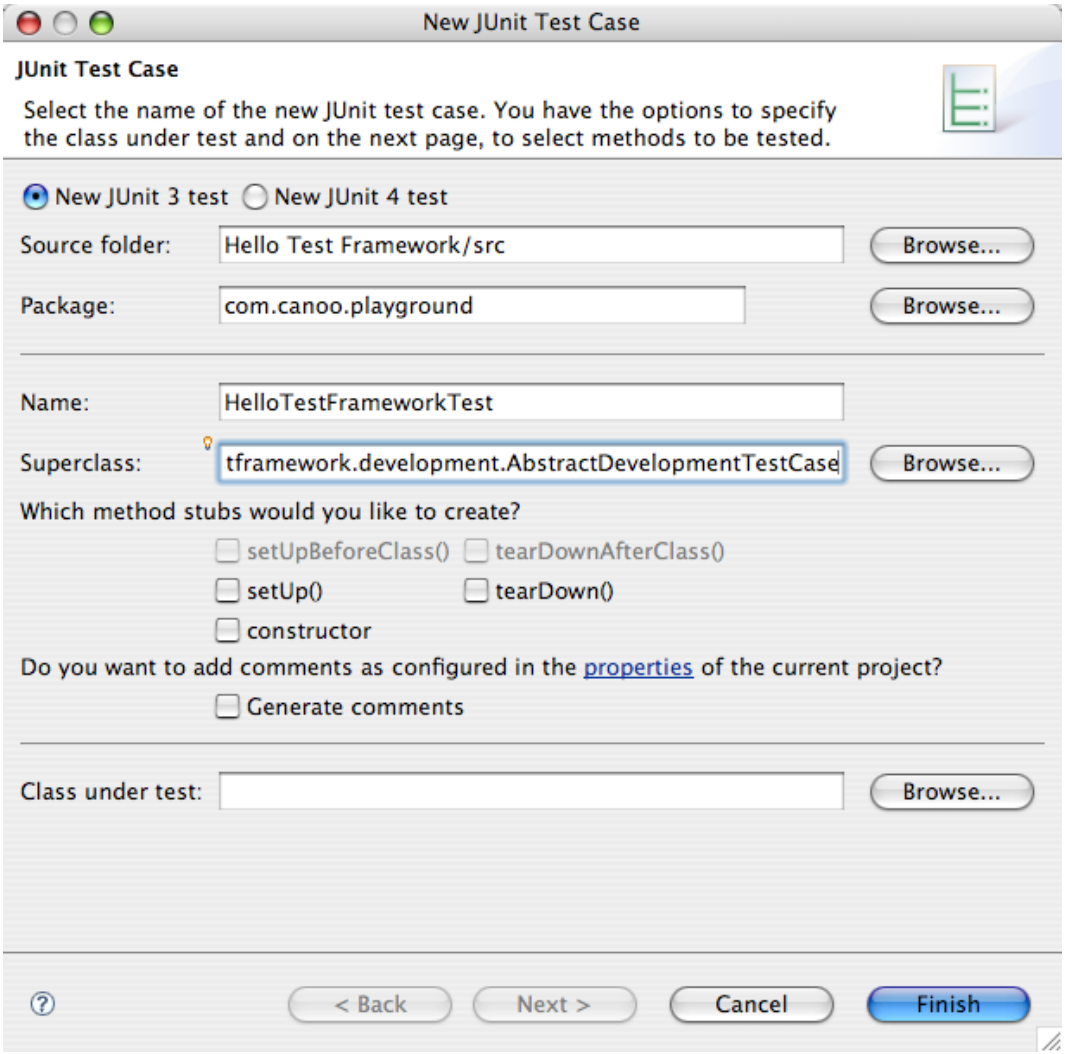
```
package com.canoo.playground;

import com.ulcjava.applicationframework.application.SingleFrameApplication;
import com.ulcjava.base.application.ULCComponent;
import com.ulcjava.base.application.ULCLabel;

public class HelloTestFramework extends SingleFrameApplication {
    @Override
    protected ULCComponent createStartupMainContent() {
        return new ULCLabel("Hello Test Framework");
    }
}
```

- Your project configuration should now look as follows:



2.2 Add a Test to the ULC Project

- Use the **New JUnit Test Case** wizard to create a new test class.
  Make sure that the new test class extends `AbstractDevelopmentTestCase`.



- Implement the abstract `getApplicationClass()` method.
- Add test code to the `HelloTestFrameworkTest` test class.
- Your Java file should now look as follows:

```
package com.canoo.playground;

import com.ulcjava.testframework.operator.ULCFrameOperator;
import com.ulcjava.testframework.operator.ULCLabelOperator;
import com.ulcjava.testframework.development.AbstractDevelopmentTestCase;

public class HelloTestFrameworkTest extends AbstractDevelopmentTestCase {
    public void testHelloTestFrameworkLabel() {
        ULCFrameOperator frame = new ULCFrameOperator();
        ULCLabelOperator label = new ULCLabelOperator(frame);

        assertEquals("Hello Test Framework", label.getText());
    }

    protected Class getApplicationClass() {
        return HelloTestFramework.class;
    }
}
```

2.3 Start the Tests

- Select **Run As > JUnit Test** in the context menu of your test class to start the tests.

3 Test How Tos

The following chapters describe common test tasks.

3.1 How To Decide Which Abstract Test Case Class to Use

Writing an ULC test starts by subclassing an existing abstract test case class. However, there are many abstract test case classes so it isn't easy to figure out which one to choose. You determine the best-suited abstract test case class by answering two questions:

1 What do I want to test?
(complete application, application parts, or deployed application)

2 What client environment do I want to test
(development, Applet, JNLP, or standalone client environment)

3.1.1 Abstract test case classes to test complete applications

In that case the ULC test framework determines the application class under test from your ULCApplicationConfiguration.xml file. If you don't configure your ULC application with help of a ULCApplicationConfiguration.xml file then you have to override `getApplicationClass()` and return the application class. The ULC test framework then uses this information to determine the application class under test.

Sublcass one of the following abstract test case classes depending on the client environment:

- Development → `AbstractDevelopmentTestCase`
- Applet → `AbstractAppletTestCase`
- JNLP → `AbstractJnlpTestCase`
- Standalone → `AbstractStandaloneTestCase`

3.1.2 Abstract test case classes to test application parts

In that case the test class and the application class are the same. This means you have to implement the abstract `start()` method in your concrete test case class.

Sublcass one of the following abstract test case classes depending on the client environment:

- Development → `AbstractSimpleDevelopmentTestCase`
- Development → `AbstractAnnotatedSimpleDevelopmentTestCase`
- Applet → `AbstractSimpleAppletTestCase`
- JNLP → `AbstractSimpleJnlpTestCase`
- Standalone → `AbstractSimpleStandaloneTestCase`

3.1.3 Abstract test case classes to test deployed applications

In this case you have to override the `getUrlString()` method and return the URL to the deployed ULC application under test. The application URL is the URL that the `ServletContainerAdapter` is mapped to. You can find this information in the web.xml of the deployed ULC application.

Sublcass one of the following abstract test case classes depending on the client environment:

- Applet → `AbstractAppletRemoteServerTestCase`
- JNLP → `AbstractJnlpRemoteServerTestCase`
- Standalone → `AbstractStandaloneRemoteServerTestCase`

3.2 How To Find User Interface Elements

In the ULC test framework, user interface elements are accessed via so-called operators. Use the operators to find other operators, to manipulate the underlying user interface element, and to read state from the underlying user interface element. There is an operator class for each user interface element, e.g.

- `ULCButtonOperator` for buttons
- `ULCFrameOperator` for windows
- `ULCLabelOperator` for text output
- `ULCTextFieldOperator` for input fields
- …

Therefore, finding a user interface element actually means creating a corresponding operator with help of some search criteria, , e.g. the text on a button, the title of a window, the index of a user interface element in a container. If two or more user interface elements match the search criteria then the one that is first found is returned.

You can facilitate the identification of user interface elements by adding metadata to the components in your ULC application. The ULC test framework uses the following metadata:

- **`name`** property of a **`ULCComponent`**
  → `com.ulcjava.testframework.operators.ComponentByNameChooser`
- **`toolTipText`** property of a **`ULCComponent`**
  → `com.ulcjava.testframework.operators.ComponentByToolTipTextChooser`

Additionally each ULCComponentOperator has a Constructor that takes the corresponding ULCComponent (e.g. ULCLabelOperator(ULCLabel label)) and returns the operator for that ULCComponent.

To create an operator you basically have two options:

- **Wait** for the user interface element as specified by the search criteria until it appears on the screen.

  This is the default option. After some time the ULC test framework throws a timeout exception if the user interface element does not appears.

  To use this option you can either use constructors (e.g. `new ULCButtonOperator(`*`searchCriteria`*`)`) or static methods that are prefixed with wait (e.g. `ULCButtonOperator.waitULCButton(`*`searchCriteria`*`)`).
- **Find** the user interface element among the currently visible user interface elements.

  The ULC test framework returns the found user interface element or null if there is no user interface element currently visible that matches the search criteria.

  To use this option you have to use static methods that are prefixed with find (e.g. `ULCButtonOperator.findULCButton(`*`searchCriteria`*`)`).

The recommended way to find user interface elements is to use constructors as this makes the code most readable.

All wait and find operations are within the context of the surrounding container. This means you do not search for a user interface element globally, you always search for it inside another user interface element. The only exceptions to this rule are windows as they are root elements that are not contained within other user interface elements.

Examples

Wait for the first window that contains *foo* in its title and wait for the *second* button that has a text of *bar* inside this window:

```
ULCFrameOperator frame = new ULCFrameOperator("foo");
ULCButtonOperator button = new ULCButtonOperator(frame, "bar", 2));
```

Now the same example with **static wait** methods instead of constructors:

```
ULCFrameOperator frame = new ULCFrameOperator(
  ULCFrameOperator.waitULCFrame("foo", false, false));
ULCButtonOperator button = new ULCButtonOperator(
  ULCButtonOperator.waitULCButton(frame, "bar", false, false, 1));
```

And finally the same example with **static find** instead of static wait:

```
ULCFrameOperator frame = new ULCFrameOperator(
  ULCFrameOperator.findULCFrame("foo", false, false));
ULCButtonOperator button = new ULCButtonOperator(
  ULCButtonOperator.findULCButton(frame, "bar", false, false, 2));
```

Find the first table that is named *bar* inside a window:

```
ULCTableOperator button = new ULCTableOperator(frame,
  new ComponentByNameChooser("bar")));
```

3.3 How To Trigger User Interactions

In addition to the find API, the operator classes provide API to trigger user interactions.

The `AWTComponentOperator` class is the base class of all operators. It provides only very generic user interaction API, e.g. click mouse button, type key. In most cases you should not use this generic user interaction API. It is a best practice to use the operator class specific user interaction API. Examples of operator specific user interaction API are:

- Push a button → Use `ULCButtonOperator.pushButton()` instead of `AWTComponentOperator.clickMouse()`.
- Type text into a text field → Use `ULCTextFieldOperator.typeText(String)` instead of repeated `AWTComponentOperator.typeKey()` calls.
- Select item in list → Use `ULCListOperator.selectItem(String item)` instead of a combination of `ULCListOperator.findItemIndex(String)`, `ULCListOperator.getClickPoint(int index)`, and `AWTComponentOperator.clickMouse(int x, int y, int clickCount)` calls.

Operator specific user interaction APIs can be found on each Operator class and the description can be found in the corresponding Javadoc.

Examples

Clear input field and enter *foo*:

```
ULCTextFieldOperator.clearText();
ULCTextFieldOperator.typeText("foo");
```

Push a button:

```
ULCButtonOperator.pushButton();
```

Close a window:

```
ULCFrameOperator.close();
```

3.4 How to Read State from User Interface Elements

The operator classes provide API to read state from user interface elements. You can use this state to check for certain conditions in your user interface.

Examples

Get the text that is displayed by an output field:

```
ULCLabelOperator.getText()
```

Get the a window title:

```
ULCFrameOperator.getTitle()
```

3.5 How to Find Cells in User Interface Elements

Some user interface elements are built out of a large number of sub-elements. These sub-elements are usually referred to as cells. The following user interface elements are cell-based:

- List
- Table
- Tree
- TableTree

The ULC test framework does not provide cell specific operators. Instead, the operator for the cell-based widgets itself provides API to find, access, read and, manipulate cells.

ULC uploads cells of the cell-based widgets to the client in lazy manner. The ULC test framework takes care of the specifics of the ULC's lazy loading feature. Therefore, you need not worry about whether a certain cell is visible or not before working with that cell.

Examples

Find the first index of a list item that contains *foo* as text:

```
int itemIndex = ULCListOperator.findItemIndex("foo");
```

Find the location of the *second* table cell that contains *foo* as text:

```
Point cellLocation = ULCTableOperator.findCell("foo", 2);
```

Find the first table row that contains *foo* in its cells:

```
int rowIndex = ULCTableOperator.findCellRow("foo");
```

Find the first table column that contains *foo* in its cells:

```
int columnIndex = ULCTableOperator.findCellColumn("foo");
```

Find the second tree row that contains *foo* as text:

```
int rowIndex = ULCTreeOperator.findRow("foo", 2);
```

Find the first tree path that matches the specified path description:

```
TreePath treePath = ULCTreeOperator.findPath("foo,bar", ",");
```

3.6 How to Test with Launchers

The ULC test framework enables you to use exactly the same classes that you will deploy in a production environment. There is no need to write test specific infrastructure classes. This enables real end-to-end tests that involve all parts of an ULC application.

For each launcher type there is a corresponding abstract test class you can use instead of the `AbstractDevelopmentTest` class:

- Applet based launchers (i.e. extending `AbstractAppletLauncher`)
  → use `AbstractAppletTestCase`
- JNLP based launchers (i.e. extending `AbstractJnlpLauncher`)
  → use `AbstractJnlpTestCase`
- Standalone launchers (i.e. extending `AbstractStandaloneLauncher`)
  → use `AbstractStandaloneTestCase`

For each test, these abstract test classes:

1 Start an "in process" Servlet container.

2 Deploy the ULC application into that container.

3 Start this application with help of a launcher.

This means these abstract test classes do exactly the same thing that an end user would do with your ULC application in production.

3.6.1 Setup for launcher based testing

To enable production environment testing feature, the ULC test framework needs some more libraries on the class path to start an "in process" Servlet container and deploy the ULC application

- ULC deployment license: *ulc-deployment-key.jar*
- Jetty Servlet container (v6.1.16): *jetty.jar*, *jetty-util.jar*, *servlet-api.jar (v2.5)*

3.6.2 Simulating client/server environments

With help of the abstract launcher test classes you can configure the ULC test framework to simulate an environment that comes very close to the production environment for your ULC application.

Using a configuration file

If you don't overwrite getApplicationClass() to return a application class, the test case is configured with the ULCApplictionConfiguration.xml file. The ULC application to run for the test case is read from the configuration file as well as the environments are configured as specified in the configuration file. See the ULC Application Development Guide for further information about the configuration file.

Specifiying the configuration file

The application configuration file is loaded as a resource with the name */ULCApplicationConfiguration.xml*. If you'd like to run a test that does not use the application's own configuration file, you can overwrite the `getConfigurationResourceName()` method of the test case to specifiy another configuration file (for example */org/myproject/test/TestConfiguration.xml*).

Configuring the environments without a configuration file

If getApplicationClass() returns not *null*, no configuration file is read. The client and server test environment, can be configured by overriding the two *configure* methods, one for the client side test environment and one for the server side test environment.

Examples

Override *AbstractAppletTestCase.configure(AppletTestEnvironmentAdapter)* to configure your custom Applet launcher:

```
protected void configure(AppletTestEnvironmentAdapter clientAdapter){
    clientAdapter.setLauncherClass(MyAppletLauncher.class);
  }
```

Override *AbstractAppletTestCase.configure(AppletTestEnvironmentAdapter)* to configure user parameters:

```
protected void configure(AppletTestEnvironmentAdapter clientAdapter){
  Properties userParameters = new Properties();
  userParameters.setProperty("user1", "value1");
  userParameters.setProperty("user2", "value2");

  clientAdapter.setUserParameters(userParameters);
}
```

Override *AbstractAppletTestCase.configure(ServletTestEnvironmentAdapter)* to configure your custom Servlet:

```
protected void configure(ServletTestEnvironmentAdapter servletAdapter) {
  servletAdapter.setServletClass(ServletContainerAdapter.class);
}
```

The default server port number is 45365. To make the Jetty server listen on another port override *AbstractServletTestCase.configure(ServletTestEnvironmentAdapter)* method and specify the new port in the *AbstractClientTestEnvironmentAdapter.setUrlString()* method :

```
public class TestWithCustomPort extends AbstractSimpleStandaloneTestCase {
    private static final int SERVER_PORT = 12345;
    protected void configure(ServletTestEnvironmentAdapter servletAdapter) {
        servletAdapter.setServerPort(SERVER_PORT);
    }
    protected void configure(StandaloneTestEnvironmentAdapter clientAdapter) {
        clientAdapter.setUrlString("http://localhost:" +  SERVER_PORT +
                                                        "/application.ulc");
    }
    …
```

3.7 How to Test Already Deployed Applications

The ULC test framework enables you to test already deployed ULC applications. Use this feature to check the availability and performance of your ULC applications in production.

For each launcher type, there is a corresponding abstract test class you can use instead of the `AbstractDevelopmentTest` class:

- Applet based launchers (i.e. extending `AbstractAppletLauncher`)
  → use `AbstractAppletRemoteServerTestCase`
- JNLP based launchers (i.e. extending `AbstractJnlpLauncher`)
  → use `AbstractJnlpRemoteServerTestCase`
- Standalone launchers (i.e. extending `AbstractStandaloneLauncher`)
  → use `AbstractStandaloneRemoteServerTestCase`

For each test, these abstract test classes start your remote ULC application with the help of a launcher. This means these launchers do exactly the same thing that an end-user would do with your ULC application in production.

3.7.1 Setup to test already deployed application

To set up testing for deployed applications, the ULC test framework needs further libraries on the class path to start your ULC application with a **JNLP based launcher**

- JNLP Services: *javaws.jar* (you can find this library in your Java home directory)

The following example tests the ulc set sample application.

```
public class RemoteTestCase extends AbstractJnlpRemoteServerTestCase {

    @Override
    protected String getUrlString() {
        return "http://localhost:45165/ulcset.ulc";
    }

    public void testTheUI() throws Exception {
        ULCFrameOperator frame = new ULCFrameOperator();
        ULCTextFieldOperator field = new ULCTextFieldOperator(frame, "A button");
        ULCButtonOperator button = new ULCButtonOperator(frame, "A button");

        field.enterText("button label");
        assertEquals("button label", button.getText());
    }

}
```

Before running the test you must start the sample applications by running the startServer script in the ULC_INSTALLATION_DIR/sample directory.

3.7.2 Configuring the launcher test classes

If your application uses the ULCApplicationConfiguration.xml file, all you have to do is to subclass the AbstractRemote_xxx_TestCase for the client environment you like to use and write the tests.

If your application does not use the configuration file you must use the abstract launcher test classes to configure the client environment for the ULC test framework to test your ULC application that is already deployed in a production environment. To do so, you need to override the configure method for the client side test environment.

Examples

To configure your custom AppletLauncher override the method *AbstractAppletRemoteServerTestCase.configure(AppletTestEnvironmentAdapter)*:

```
protected void configure(AppletTestEnvironmentAdapter clientAdapter {
  clientAdapter.setLauncherClass(MyAppletLauncher.class);
}
```

To configure user parameters override the method *AbstractAppletRemoteServerTestCase.configure(AppletTestEnvironmentAdapter)*:

```
protected void configure(AppletTestEnvironmentAdapter clientAdapter {
  Properties userParameters = new Properties();
  userParameters.setProperty("user1", "value1");
  userParameters.setProperty("user2", "value2");
  clientAdapter.setUserParameters(userParameters);
}
```

3.8 How to Take Care of Platform Asynchronicity

Some combinations of operating system platform and Java version are known to have an asynchronous behavior with respect to user interactions, e.g. Linux and early Java versions. By asynchronous we mean that when a Java call to the operating system platform returns it is not assured that the operating system platform has already completed the operation. For example, a window close operation returns before the window is actually closed by the underlying operating system. Such asynchronicity can result in strange effects in your tests:

- A test fails but when you restart it, it is suddenly successful.
- Your tests are successful 99% of the time but fail for 1%. The failing 1% are always different test cases.

To deal with such platform asynchronicity, use *MaxFailCountTest*. In the constructor, you specify the maximum number of failures a test could have. As soon as this number is exceeded, the test is considered as failed. Default value is 1 meaning that a given test is allowed to fail *once*.

Examples

```
public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new MaxFailCountTest<TestSuite>(
            new TestSuite(ULCGreenTest.class), 5));
    suite.addTest(new MaxFailCountTest<TestSuite>(
            new TestSuite(ULCRedTestWithAssertionFailedError.class), 5, 1000));
    return suite;
 }
```

3.9 How to Fine Tune Jemmy

The ULC test framework uses the Jemmy library to interact with the user interface of your ULC application. The test framework takes care to configure Jemmy in a meaningful way to test ULC applications:

- The test output is set to a `NullOutput`. Therefore Jemmy does not log the test steps to standard out.
- The timeouts are decreased. Jemmy uses timeouts stop an operation after some time if it is not successful. See *Chapter 3.2 How To Find User Interface Elements* for an example.
- The dispatching model is set to Jemmy's queue model.
- The drivers from the `TestFrameworkDriverInstaller` are installed.

The timeouts can be configured by providing a property file named *jemmyTimeout.properties*, which is located on the class path. You can reconfigure Jemmy anywhere in your test code. Just keep in mind that the ULC test framework

overwrites your changes for the next test run. The best way to change the default configuration for all tests in a test class is to put your configuration code after the default `AbstractTestCase.setUp()` method.

Examples

Configure the wait frame timeout to one minute inside one test method:

```
public void testHelloTestFrameworkLabel() {
  JemmyProperties.setCurrentTimeout("FrameWaiter.WaitFrameTimeout", 60000);

  ULCFrameOperator frame = new ULCFrameOperator("Hello Test Framework");
  ULCLabelOperator label = new ULCLabelOperator(frame);

  assertEquals("Hello Test Framework", label.getText());
}
```

Configure the wait frame timeout to one minute for the whole test case:

```
protected void setUp() throws Exception {
  super.setUp();
  JemmyProperties.setCurrentTimeout("FrameWaiter.WaitFrameTimeout", 60000);
}
```

3.10 How to Use Other (Jemmy) API

The ULC test framework provides an easy-to-understand API. In case this API is not sufficient for your tests, you can use any other API you wish. Just make sure that you invoke this API in the context of the ULC test framework. This ensures that the ULC test framework takes care of all ULC specifics.

The com.ulcjava.testframework.AbstractTestCase, the base class of all the environment specific test case classes (such as AbstractDevelopmentTestCase), provides methods that help you to invoke any API in the context of the ULC test framework.

Examples

Run the garbage collector:

```
runVoidCommand(new TestCommand() {
  protected void proceed() {
    Runtime.getRuntime().gc();
  }
});
```

Get the name of the current thread:

```
String name = (String)runObjectCommand(new TestCommand() {
  protected void proceed() {
    setResult(Thread.currentThread().getName());
  }
});
```

The following example shows how to handle exceptions:

```
try {
  runVoidCommand(new TestCommand() {
    protected void proceed() throws IOException {
      throw new IOException();
    }
  });
} catch (ProceedException e) {
  IOException cause = (IOException)e.getCause();
}
```

3.11 How to execute code on the server side

If you are using one of the integrated test environments that are running the client and the server side in the same VM (see 3.6 How to Test with Launchers), you can execute code on the server side. This is useful to test server side logic and server side event handling. This works exactly as executing code on the client but using a *ServerSideCommand* instead of the common TestCommand The method that gets executed is *proceedOnServer()*, instead of just *proceed()*.

Examples

Get the name of the thread on the server side:

```
String name = (String)runObjectCommand(new ServerSideCommand() {
  protected void proceedOnServer() {
    setResult(Thread.currentThread().getName());
  }
});
```

The following example shows how to handle exceptions:

```
try {
  runVoidCommand(new ServerSideCommand() {
    protected void proceedOnServer() throws IOException {
      throw new IOException();
    }
  });
} catch (ProceedException e) {
  IOException cause = (IOException)e.getCause();
}
```

You can get access to the server side ULCComponents through the ULCOperators, e.g ULCLabelOperator.getULCLabel(), ULCListOperator.getULCList() and so on.

The following example shows how to change the text of a label on the server side.

```
package com.ulcjava.sample.testframework;

import com.ulcjava.testframework.operator.ULCFrameOperator;
import com.ulcjava.testframework.operator.ULCLabelOperator;
import com.ulcjava.testframework.development.AbstractDevelopmentTestCase;

public class ServerSideCommandExampleTest extends AbstractDevelopmentTestCase {
    public void testChangeLabelTextOnServerSide() {
        ULCFrameOperator frame = new ULCFrameOperator("Hello Test Framework");
        final ULCLabelOperator label = new ULCLabelOperator(frame);

        assertEquals("Hello Test Framework", label.getText());

        runVoidCommand(new ServerSideCommand() {
            protected void proceedOnServer() throws Throwable {
               ULCLabel ulcLabel = label.getULCLabel();
               ulcLabel.setText("Goodbye");
            }
        });

        assertEquals("Goodbye", label.getText());
    }

    protected Class getApplicationClass() {
        return HelloTestFramework.class;
    }
}
```

3.12 How to write a Test Case that doesn't need an application

If you just want to write a Test that doesn't need a specific application, for example if you're unit-testing a single dialog, you can use the AbstractSimple-*SomeEnvironment*-TestCase for the environment in which you wish to run the test such as *AbstractSimpleDevelopmentTestCase*, *AbstractAnnotatedSimpleDevelopmentTestCase*,

AbstractSimpleJnlpTestCase, *AbstractSimpleAppletTestCase*, and *AbstractSimpleStandaloneTestCase*.

These classes implement the *com.ulcjava.base.application.IApplication* Interface leaving the start method to be overwritten in subclasses.

Example

The test class is also the ULC-Application.

```
package com.ulcjava.sample.testframework;

import com.ulcjava.base.application.ULCFrame;
import com.ulcjava.base.application.ULCLabel;
import com.ulcjava.testframework.ServerSideCommand;
import com.ulcjava.testframework.development.AbstractSimpleDevelopmentTestCase;
import com.ulcjava.testframework.operator.ULCLabelOperator;

public class SimpleDevelopmentTestCase extends AbstractSimpleDevelopmentTestCase {
    private ULCLabel fLabel;

    public void testChangeLabel() {
        runVoidCommand(new ServerSideCommand() {
            protected void proceedOnServer() throws Throwable {
                fLabel.setText("Goodbye");
            }
        });

        // ULCLabelOperator is found using the ULCLabel
        ULCLabelOperator labelOperator = new ULCLabelOperator(fLabel);
        assertEquals("Goodbye", labelOperator.getText());
    }

    // server side
    public void start() {
        ULCFrame rootPane = new ULCFrame("Hello World");
        fLabel = new ULCLabel("Hello");
        rootPane.add(fLabel);
        rootPane.setVisible(true);
    }
}
```

As an alternative to *AbstractSimpleDevelopmentTestCase*, *AbstractAnnotatedSimpleDevelopmentTestCase* allows writing tests with JUnit 4 *Annotations*:

Example

```
import org.junit.Test;

import com.ulcjava.base.application.ULCFrame;
import com.ulcjava.base.application.ULCLabel;
import com.ulcjava.testframework.ServerSideCommand;
import com.ulcjava.testframework.operator.ULCLabelOperator;

public class AnnotatedSimpleDevelopmentTestCase extends
AbstractAnnotatedSimpleDevelopmentTestCase {
    private ULCLabel fLabel;

    @Test
    public void checkLabelText() {
        ULCLabelOperator labelOperator = new ULCLabelOperator(fLabel);
        assertEquals("Hello", labelOperator.getText());
        runVoidCommand(new ServerSideCommand() {
            @Override
            protected void proceedOnServer() throws Throwable {
                fLabel.setText("Goodbye");
            }
```

```
        });
        assertEquals("Goodbye", labelOperator.getText());
    }

    @Override
    public void start() {
        ULCFrame rootPane = new ULCFrame("Hello World");
        fLabel = new ULCLabel("Hello");
        rootPane.add(fLabel);
        rootPane.setVisible(true);
    }
}
```

3.13 How to run a Test Case in different environments

If you like to run a test in more than one environment, you can do this by subclassing from *com.ulcjava.testframework.AbstractTestCase*. The class must define a static suite method in which you create a test suite for the test class in each environment. Use the static factory method defined on *AbstractTestCase* to do this.

```
suiteForEnvironment(Class testClass, ITestEnvironmentAdapterProvider[] environments)
```

The *testClass* should be the class of your *TestCase*.

As for the *AbstractSomeEnvironmentTestCase*s, there is a *ITestEnvironmentAdapterProvider* for each environment (e.g. *DevelopmentTestEnvironmentAdapterProvider*). The constructors for the integrated environments need the class of the Application to be tested. The remote environments need the URL string of the deployed application.

Examples

This example creates a suite for the Applet and Standalone integrated environment.

```
suiteForEnvironments(RunTestInTwoEnvironmentsExampleTest.class,
  new ITestEnvironmentAdapterProvider[] {
    new StandaloneTestEnvironmentAdapterProvider(HelloTestFramework.class),
    new AppletTestEnvironmentAdapterProvider(HelloTestFramework.class)
  });
```

If you want to customize the environment you can do this by overwriting the *configure* method.

```
suiteForEnvironments( RunTestInTwoCustomizedEnvironmentsExampleTest.class,
  new ITestEnvironmentAdapterProvider[] {
    new StandaloneTestEnvironmentAdapterProvider(HelloTestFramework.class) {
      protected void configure(ServletTestEnvironmentAdapter servletAdapter) {
        servletAdapter.setCarrierStreamProviderClass(
            TrivialCarrierStreamProvider.class);
        }
      protected void configure(StandaloneTestEnvironmentAdapter clientAdapter) {
        clientAdapter.setCarrierStreamProviderClass(
            TrivialCarrierStreamProvider.class);
      }
    },
    new AppletTestEnvironmentAdapterProvider(HelloTestFramework.class) {
      protected void configure(ServletTestEnvironmentAdapter servletAdapter) {
        servletAdapter.setCarrierStreamProviderClass(
            TrivialCarrierStreamProvider.class);
      }
      protected void configure(AppletTestEnvironmentAdapter clientAdapter) {
        clientAdapter.setCarrierStreamProviderClass(
            TrivialCarrierStreamProvider.class);
      }
    }
  })
```

If you want to run the Tests in all integrated or remote environments use the convenience methods:

```
AbstractTestCase.createTestSuiteForAllIntegratedEnvironments(Class testcaseClass,
                                                             Class applicationClass)

AbstractTestCase.createTestSuiteForAllRemoteEnvironments(Class testcaseClass,
                                                         String url)
```

3.14 How to use a more modern test framework like JUnit4 or TestNG

ULC Test Framework is based on **JUnit3** and extends *TestCase*. Because it does not overrride *TestCase.runBare*, it could be easily used within a different testing framework like, for instance, **JUnit4**. Below find an example how to do this.

Example

```
@RunWith(BlockJUnit4ClassRunner.class)
public class ULCWithTestApplicationTest extends AbstractDevelopmentTestCase {

    @Override
    @Before
    public void setUp() throws Exception {
        super.setUp();
    }

    @Override
    @After
    public void tearDown() throws Exception {
        super.tearDown();
    }

    @Test
    public void someExample() {
        ULCFrameOperator frame = new ULCFrameOperator();
        ULCLabelOperator label = new ULCLabelOperator(frame);
        assertEquals("Hello", label.getText());
    }
}
```

Note that if you do not specify any runner (with the *@RunWith* annotation), *JUnit38ClassRunner* will be used by default and because *AbstractDevelopmentTestCase* extends *TestCase*.