

|  |  |
| --- | --- |
|  | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |
|  | |
| File Name: | Granite Studio Architecture and Design.docx |
| Version: | 0.9 |
| Status: | Draft |
| Originator: | Mark Freed |
| Approver: | Timo Pukinkorva |
| Date: | 24-Dec-2012 |
| Site | Oulu/Southwood |

# Change History

Version 0.9 24-Dec-2012 Mark Freed Draft

Reason: Updates for Competency Transfer.

Description: Updates and additional information added.

Version 0.8 30-Nov-2012 Mark Freed Draft

Reason: Minor updates.

Description: Minor corrections and improvements.

Version 0.7 26-Nov-2012 Mark Freed Draft

Reason: Updated document title

Description: Changed title since this is the Granite Studio architecture and design.

Version 0.6 10-Aug-2012 Mark Freed Draft

Reason: Document updated with additional information.

Description: Updated the development tips.

Version 0.5 09-Aug-2012 Mark Freed Draft

Reason: Document updated with additional information.

Description: Added general guidelines, testing information and development tips.

Version 0.4 11-Jul-2012 Mark Freed Draft

Reason: Document updated with tracking information.

Description: Updated with tracking design information.

Version 0.3 06-Jul-2012 Mark Freed Draft

Reason: Document updated for Granite 1.0

Description: Updated since Granite 1.0 has been released.

Version 0.2 02-Mar-2012 Mark Freed Draft

Reason: Document updated in meeting review.

Description: Appendix A includes porting Silverlight code to WPF.

Version 0.1 30-Feb-2012 Mark Freed Draft

Reason: Document initiated.

Description: Initial version.

# Contents

Change History 2

Contents 3

Figures 4

1 Introduction 5

2 Architecture Overview 6

3 Overview of Studio Assemblies and Classes 11

3.1 Granite UI 11

3.2 Granite Framework 12

4 Creating Workflow Activities in Studio 15

4.1 Workflow Activity User Interface 15

5 Workflow Tracking 18

5.1 Adornments 23

5.1.1 Code changes required for adornments 25

6 Phone Display Reference Images 28

7 Plug-ins 29

8 Updating the Framework Interface 30

9 Testing of Activities 31

10 Tips and Tricks 32

10.1 Accessing the Activity from the ActivityDesigner 32

11 References 33

11.1 Granite Studio Wiki page 33

11.2 XAML 33

11.3 Windows Workflow Foundation 4.5 33

11.4 STE and STA 33

12 Appendix A – Backlog Items 34

13 Appendix B – Granite Studio Ideas 37

Figures

[Figure 1 Overview of Granite 1.0 with Workflow 6](#_Toc344131716)

[Figure 2 Granite Studio UI Controls 7](#_Toc344131717)

[Figure 3 Adding a Studio test to the Test Lab 8](#_Toc344131718)

[Figure 4 Running a Studio test from the Test Lab 9](#_Toc344131719)

[Figure 5 Editing a Studio test from the Test Lab 10](#_Toc344131720)

[Figure 6 Workflow Activity Class Diagram 11](#_Toc344131721)

[Figure 7 Architecture for Activities calling Framework methods 13](#_Toc344131722)

[Figure 7 Workflow Engine Class Diagram 14](#_Toc344131723)

[Figure 8 Collapsed Granite Workflow Activities 15](#_Toc344131724)

[Figure 9 Expanded Granite Workflow Activities 16](#_Toc344131725)

[Figure 10 Output with workflow tracking information 18](#_Toc344131726)

[Figure 11 Studio with visual workflow tracking 19](#_Toc344131727)

[Figure 12 Workflow Debugging Architecture 20](#_Toc344131728)

[Figure 12 Overview of Granite with workflow tracking 21](#_Toc344131729)

[Figure 13 Workflow Tracking Class Diagram 22](#_Toc344131730)

[Figure 14 Workflow adorners showing activities that have successfully completed 23](#_Toc344131731)

[Figure 15 Workflow adorners showing activities that have failed 24](#_Toc344131732)

[Figure 16 Overview of Granite with plug-ins 29](#_Toc344131733)

# Introduction

This document provides an overview of the architecture for incorporating Windows Workflow Foundation (WWF) in Granite. The aim is to provide a workflow user interface in order to allow manual testers and test engineers to rapidly produce test cases using Granite.

Workflow 3.5 was originally used in the System Test Editor (STE) and the System Test Application (STA) in 2007.

Workflow 4.0 was released in 2010 and provides a different workflow designer to that available in 3.5. There are some aspects to Workflow 4.0 that more closely resemble programming. For users that do not come with any programming expertise, some additional training and best-practice documentation may need to be provided in order to guide them on how to use some of these newer workflow features (e.g. variables, arguments, types, VB expressions, etc.), or otherwise we will need to provide the user with a more simplified user interface that does not require them to have some programming expertise.

At the time of writing this document .NET 4.5 is still in Beta, however Workflow 4.5 contains a number of improvements and enhancements. The main Workflow 4.5 enhancements that affect Granite are as follows:

* Add comments to activities in the designer surface.
* Expressions in the language of the project (C# Expressions).
* Use the WF designer more effectively, especially navigating large workflows.
* Search integration in the workflow designer.
* Improved runtime performance.
* Basic building blocks for versioning.

# Architecture Overview

Granite consists of two main applications; the Granite User Interface (UI) and the Granite Framework. The Granite UI allows a user to interact with the system and provides a user interface to carry out functionality such as specifying the phone connection, flashing the phone, capturing the phone display and running tests from the Test Lab.

The Granite Framework is a separate application that provides the required functionality to run the tests. Communication between the Granite UI and the Granite Framework is done using TCP sockets in Granite 1.0. The Granite Framework is essentially an IronPython script, therefore the process that is running is the IronPython executable (i.e. ipy.exe) running the \framework\granite.py script when the Granite Framework is started. The Granite UI starts the Granite Framework IronPython process when a user performs some action, e.g. executes a test, captures a phone display image, etc.

The workflow designer has been added into the Granite UI in order to allow a user to create tests by dragging and dropping workflow activities onto the designer. The workflow activity properties can be set by the user from the property grid and typically also through the activity designer. In the future, additional logic may be able to be applied using other activities such as if-then-else conditions, while loops, try-catch blocks as well as variables and arguments.

The workflow designer provides the workflow test case as Extensible Application Markup Language (XAML) which is an XML format that represents the workflow test case. The XAML file is saved by the Granite UI and a message is sent to the Granite Framework where it will be executed (see Figure 1). It is important to understand that the workflow activities never execute in the Granite UI; they always only execute from within the IronPython Granite Framework, which is an entirely separate process.



Figure 1 Overview of Granite 1.0 with Workflow

The Studio GraniteWorkflowDesigner control consists of three main controls, i.e. toolbox, workflow designer and a property grid (see Figure 2).

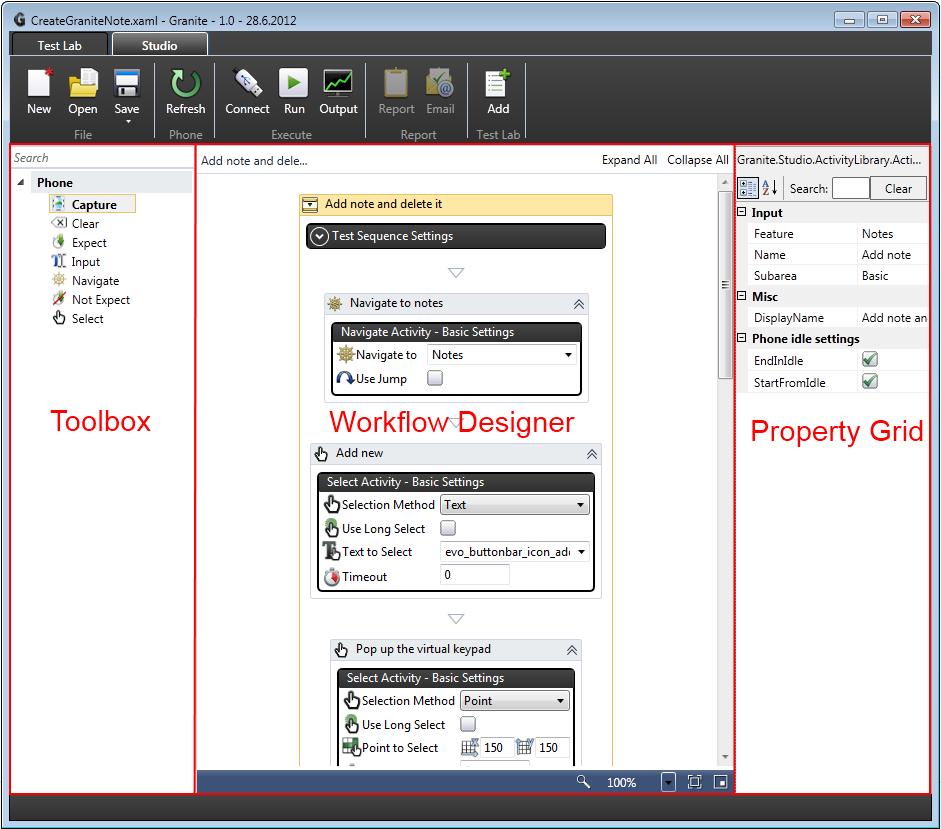


Figure 2 Granite Studio UI Controls

Studio tests that have been created in the workflow designer can be added into the Test Lab, so long as they have first been saved by the user (see Figure 3).

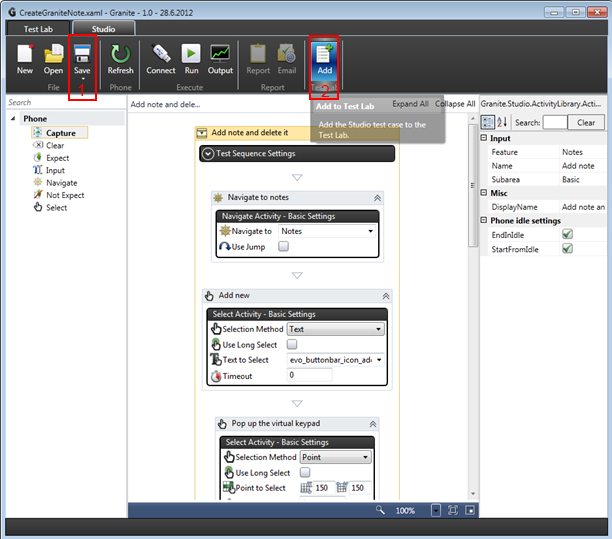


Figure 3 Adding a Studio test to the Test Lab

From the Test Lab the user can select any of the Studio tests and run them (see Figure 4). When workflow tests are run from the Test Lab, the workflow designer is not updated in any way, with the workflow from the test that is running in the Test Lab.

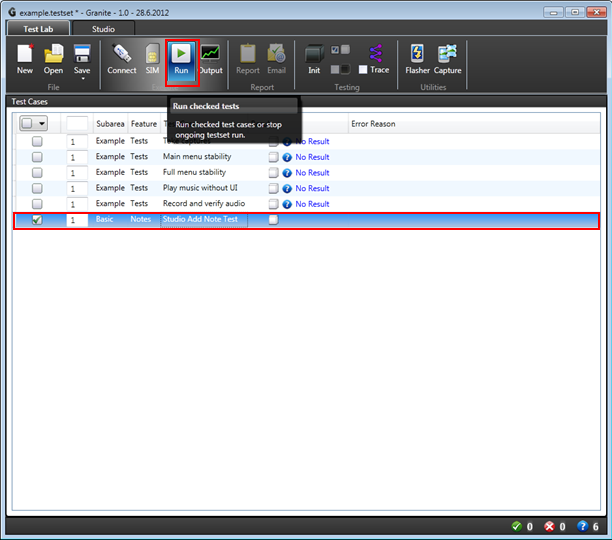


Figure 4 Running a Studio test from the Test Lab

Users can select a Studio test and right-click on the test and select an option to edit the test, which will automatically open the workflow test in the Studio tab.

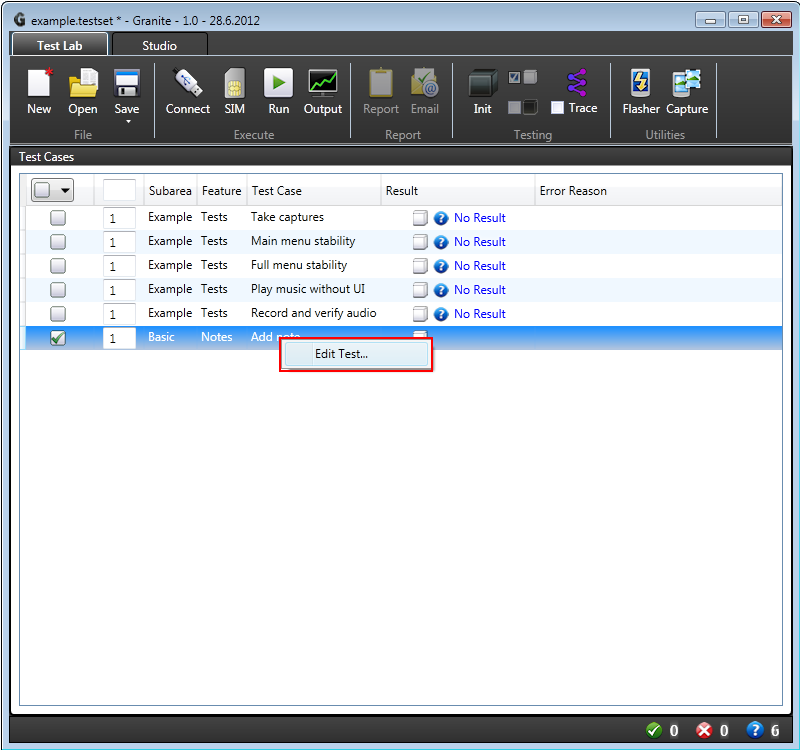


Figure 5 Editing a Studio test from the Test Lab

New workflow activities will need to be developed for Granite. Most of these will simply call existing Granite functionality that has been written in IronPython. The existing STE activities will need to be reviewed to determine which ones would be beneficial to migrate across so that they will run in Granite. This should not be a very complicated task, since STE/STA use a lot of the same libraries that the Granite UI and Granite Framework uses (e.g. Fuse, CTT communication layers, Message API + PMD functionality).

# Overview of Studio Assemblies and Classes

From the Granite UI the workflow activities are loaded into the workflow designer so that a user can view, edit or create a new Studio test. These workflow activities are however only ever executed from within the Granite Framework; they are never executed within the Granite UI process. When developing new Granite workflow activities or updating existing Granite workflow activities, if you want the Granite Framework to run the latest workflow activities, then you will need to make a Release build and run the \Studio\copy\_studio\_assemblies\_to\_framework.cmd command file which will copy the assemblies over to the Granite Framework.

## Granite UI

The Granuite UI uses the following Studio assemblies:

**Granite.Studio.ActivityLibrary.dll**

This assembly contains the workflow activities with their designer as well as associated classes such as converters, the FrameworkServiceClient used to call into the Framework, property editors, data providers, etc. This assembly references the **GraniteLibrary.dll** assembly to make use of some common code, such as controls, images and utilities.

Most of the workflow activities in this assembly derive from the GraniteCodeActivity which provides some common functionality as well as provides access to the IronPython Framework.

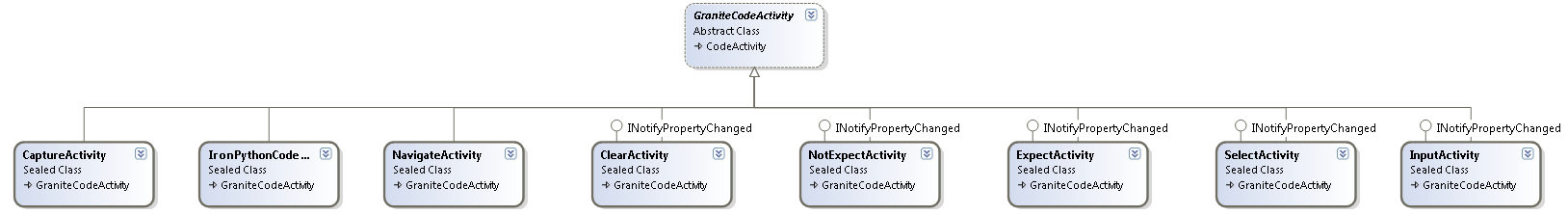


Figure 6 Workflow Activity Class Diagram

**Granite.Studio.WorkflowDesigner.dll**

This assembly contains the GraniteWorkflowDesigner control that provides the necessary functionality for displaying the workflow designer, toolbox and property grid.

The MainGUI project (Granite.exe) uses the Model-View-ViewModel design pattern. The main classes in the MainGUI project that are used to provide the Studio functionality are as follows:

StudioView - View

GraniteMainViewModel - ViewModel

GraniteStudioModel - Model

## Granite Framework

The Granite Framework uses the following Studio assemblies:

**Granite.Studio.ActivityLibrary.dll**

This assembly contains the workflow activities with their designer as well as associated classes such as converters, the FrameworkServiceClient used to call into the Framework, property editors, data providers, etc. This assembly references the **GraniteLibrary.dll** assembly to make use of some common code, such as controls, images and utilities.

The FrameworkServiceClient is the Windows Communication Foundation (WCF) client that calls into the FrameworkService provided by the Granite.Studio.WorkflowEngine.dll.

**Granite.Studio.WorkflowEngine.dll**

This assembly contains the functionality to execute the workflow provided by the WorkflowExecutor class. This assembly includes the FrameworkService class that provides the IFrameworkService WCF interface and routes any incoming requests to the IFramework interface (which is implemented in the IronPython workflow.py class).

The way the workflow engine works is as follows:

1. The Execute() method on the WorkflowExecutor is called (supplying the XAML file path and an object that supports the IFramework interface).
2. The WorkflowExecutor will then start up the FrameworkService (which support the IFramework WCF service) if it is not already started. The FrameworkService is updated with the IFramework interface to route any calls through to.
3. The workflow XAML is executed and each activity calls the relevant method on the WCF FrameworkService, which then in turns calls the method on the IFramework interface.

This means that we could run the workflow from any host that supports the IFramework interface, including being able to unit test the workflow engine by creating a stub that supports the IFramework interface. In the Granite Framework the Workflow class in the workflow.py file supports the IFramework interface. Refer to Figure 7 for the component layers of the architecture that are involved when making calls from the activities through the framework.

BackActivity

GraniteCodeActivity

CaptureActivity

CheckActivity, etc.

Workflow IronPython Class

FrameworkService

FrameworkServiceClient

IFrameworkService

IFramework

Figure 7 Architecture for Activities calling Framework methods

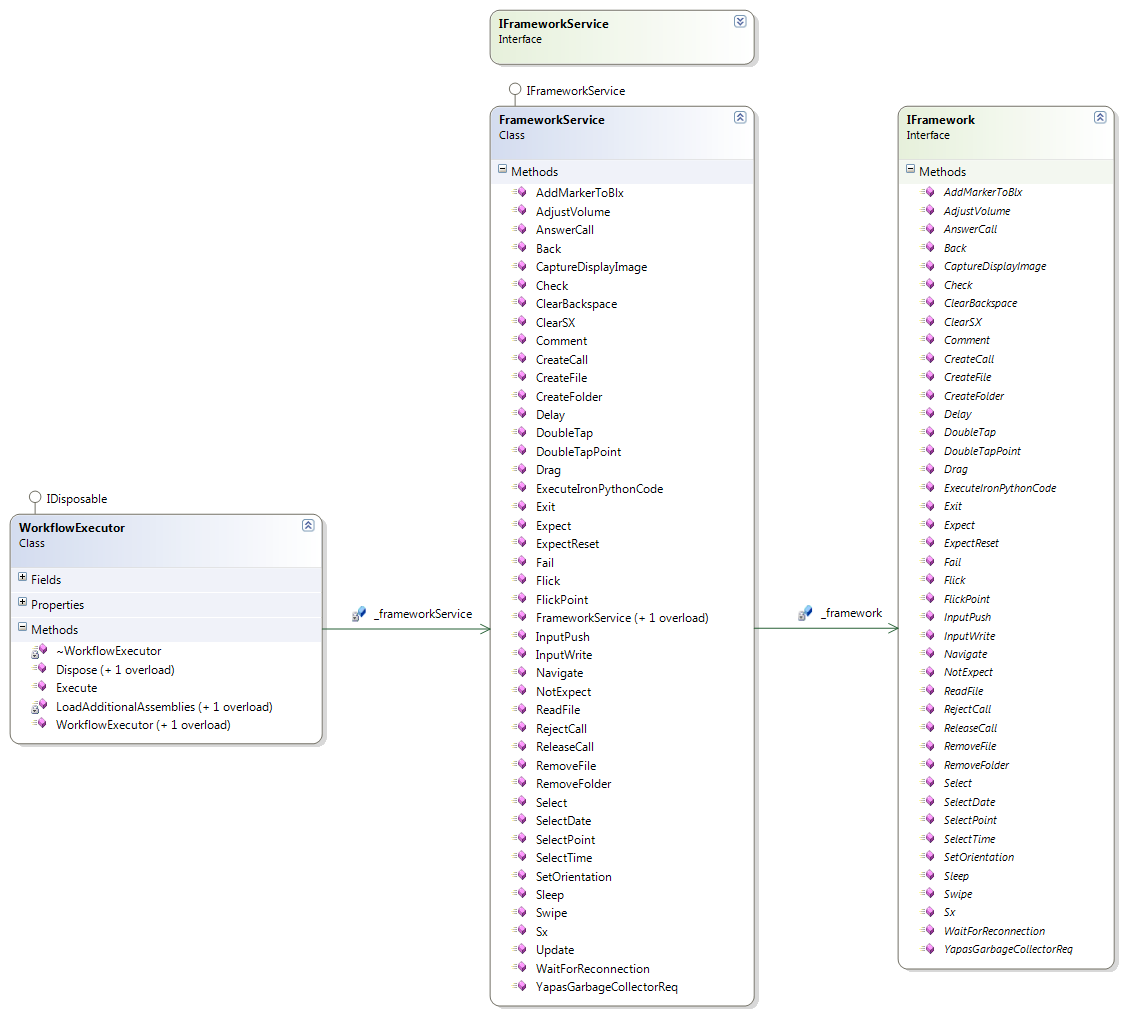


Figure 8 Workflow Engine Class Diagram

# Creating Workflow Activities in Studio

## Workflow Activity User Interface

One difference between Workflow 3.5 and Workflow 4.0 activities is that the activity designers typically have a rich user interface that allows the user to enter in or select property values.

This is same model has been used in Granite with the Workflow Activities having rich user interfaces. The style used in any activity designers should match the same theme used in Granite, which is currently a black background.

The Granite activities can all be collapsed or expanded. When collapsed they show a single line containing the most important properties. This guideline should ideally be followed for all activities when they in collapsed mode.

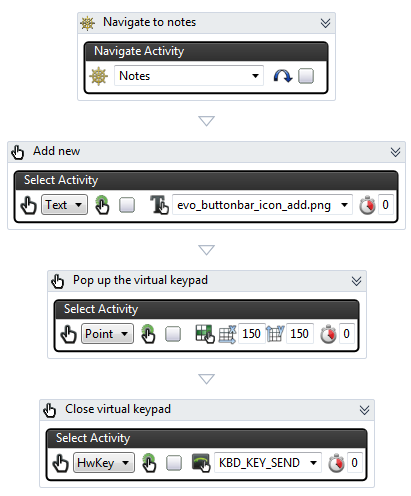


Figure 9 Collapsed Granite Workflow Activities

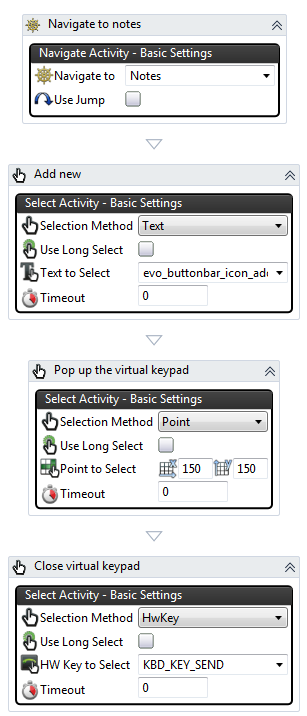


Figure 10 Expanded Granite Workflow Activities

The ActivityDesigner for an activity can be created and tested from within the Granite GUI. The general guidelines when creating a new ActivityDesigner is that everything the user needs to be able to be done should be done from within the designer control itself. It is not recommended to add buttons that pop up other dialogs. For IronPython test scripts there may be custom tools with dialogs that are created, but the ActivityDesigner should where possible provide this same functionality from within the designer itself rather than adding pop up dialogs.

# Workflow Tracking

There are two areas where workflow tracking is used:

1. Logging which activities are executing, completed and any that have faulted.  
     
   This information appears on the Output window.

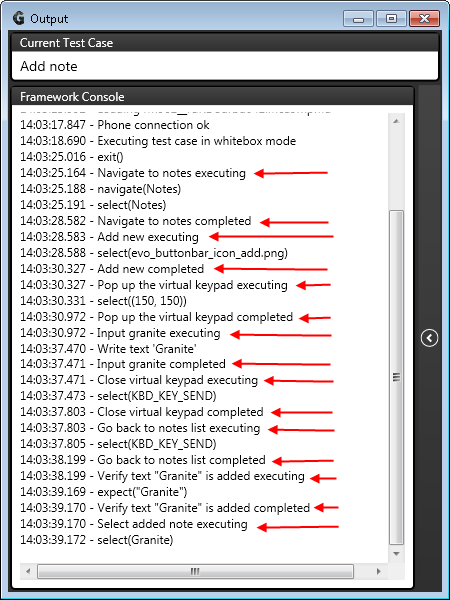


Figure 11 Output with workflow tracking information

1. Studio workflow designer shows the current activity being executed.

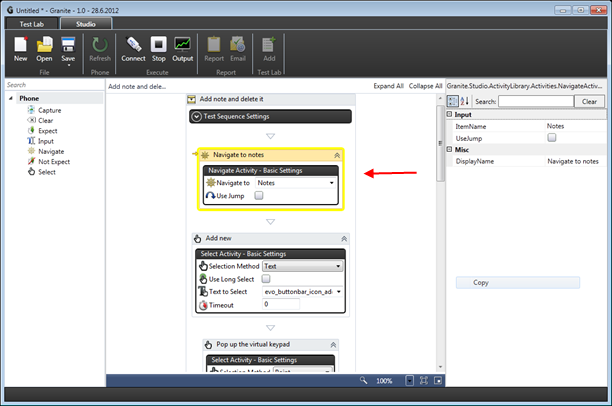


Figure 12 Studio with visual workflow tracking

When running the Studio test from the Studio tab, the tracking information will provide a visual graphical indication to show the user the current activity that is running.

The workflow activity being executed is highlighted by supplying the DebugManagerView with the source location of the activity in the XAML file (see the ShowDebug() method in the GraniteWorkflowDesigner class where we set the DebugManagerView current location). In Granite Studio we generate dummy source locations for each activity (one activity per line), since we have no need to show the real XAML file and source code. A source location has startLine, startColumn, endLine and endColumn to specify the location in a file. These dummy source locations are generated in the SourceLocationManager class and we create dictionaries that map to these dummy source locations; one dictionary that maps the activities to source locations and another one that maps activity ids to source locations.

We update the source locations that the DebugManagerView uses with the dictionary mapping activities to their dummy source location. Since the workflow tracking activities that we receive from the Framework refer to the activity being executed by its id, we use the dictionary that maps the activity ids to source locations when processing tracking events to identify the dummy source location for the activity in the tracking record. We can then provide the DebugManagerView with the current source location, which it will then use to highlight the correct activity on the workflow designer.

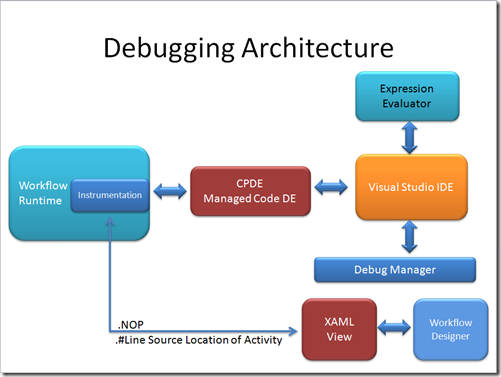
[](http://blogs.msdn.com/blogfiles/kushals/WindowsLiveWriter/VisualWorkflowTrackingWithStepService_B776/DebuggingArchitecture_2.png)

Figure 13 Workflow Debugging Architecture

Figure 13 shows the Workflow debugging architecture. The most important aspect is that we have a 1:1 mapping between the activity objects on the designer surface and their corresponding line numbers. Similarly we can also have a 1:1 mapping between the activity object that is executed and their unique activity id. We therefore can have the following mapping:

Activity object <-> XAML line number (i.e. source location) <-> activity id

The DebugManagerView uses the mapping from activity object to source location and for tracking we use the mapping from activity id to source location, so that we can set the current source location on the DebugManagerView based on the tracking information. Setting the current source location on the DebugManagerView will then highlight the activity on the workflow designer as shown in Figure 12.

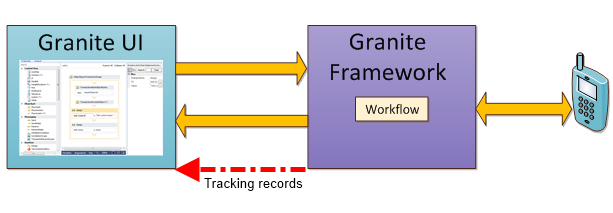


Figure 14 Overview of Granite with workflow tracking

When a user designs a test in the workflow designer of the Granite UI, he/she needs to be able to run the test in the Granite Framework, but still see workflow tracking information appear in the workflow designer in the Granite UI. For this reason, the workflow tracking information needs to be sent back from the Granite Framework to the Granite UI.

When run from the Test Lab, the workflow designer is not updated with the workflow tracking information. As a result, there needs to be a mechanism where we can run a workflow test case optionally with or without tracking.

If the user runs the test from the Studio workflow designer, it is useful to see visual tracking information. Granite allows the user to start running the Studio test case and while the Granite Framework is running the test, the user can open another XAML workflow file or create a new workflow. In this situation, it would be illogical to show visual tracking information for the workflow that is running.

There are two possible solutions:

1. Disable the New or Open buttons while running the Studio test case.
2. Provide a mechanism where the tracking information from the Granite Framework is ignored if the user has opened a different workflow file or created a new workflow Studio test.

With the current implementation a new workflow designer is created whenever the user presses New or Open and as a result any tracking events received from a Studio test that is running in the framework, shall be ignored.

An instance of the TrackingService class is created by the GraniteWorkflowDesigner when the control is loaded for the first time. Once this WCF tracking service has been created, it is available to receive any tracking messages. There is code in the GraniteWorkflowDesigner to handle the tracking events that are received from the Granite Framework. When we receive a tracking event that indicates we have started, the SourceLocationManager class is used to map activities and activity identifiers to locations in a XAML file. As long as there is always a one-to-one mapping, there was no need to provide real source location information for the location of the activity in the XAML file. This class therefore generates dummy source location information. Refer to Figure 15 for the main classes that are involved in the workflow tracking (apart from the GraniteWorkfowDesigner).

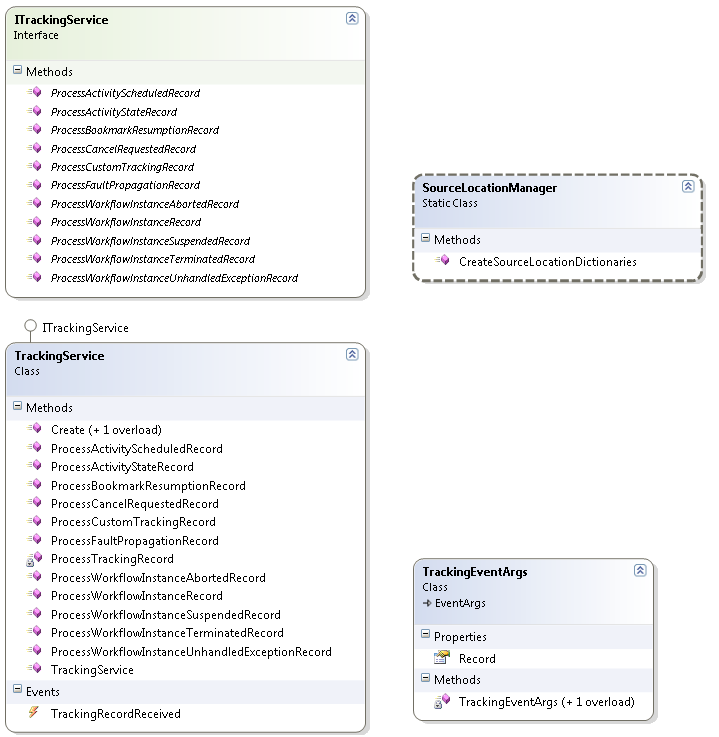


Figure 15 Workflow Tracking Class Diagram

## Adornments

After each activity has been executed, it should be updated with an adornment to indicate whether it was successfully executed or whether a failure occurred.

An example of such an adornment is show below:

Success: C:\Users\mfreed\My Code\Granite\GUIRelease\GraniteLibrary\Images\passed.circle.icon.16x16.png

Failure: C:\Users\mfreed\My Code\Granite\GUIRelease\GraniteLibrary\Images\failed.circle.icon.16x16.png

These adorners need to be shown while a test is executing in Granite Studio as well as after the test has completed in order to show the user which activities passed and failed.

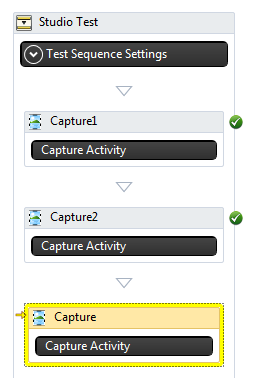


Figure 16 Workflow adorners showing activities that have successfully completed

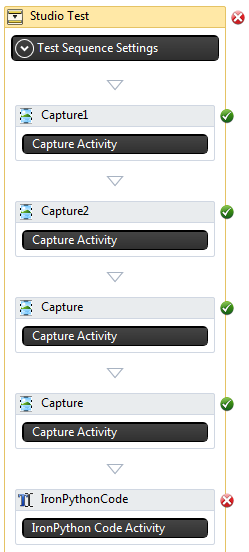


Figure 17 Workflow adorners showing activities that have failed

### Code changes required for adornments

Adding adornments to every activity designer required a custom workflow designer template to be specified and this was not a straight-forward task. For this reason a detailed explanation of the code changes are described in this section.

Firstly, the original workflow designer default template was extracted from the System.Activiities.Presentation.dll assembly under Resources -> System.Activities.Presentation.g.resources -> system/activities/presentation/workflowelementdesignerdefaulttemplate.baml

This can be extracted using Reflector and then added to our Granite.Studio.WorkflowDesigner.dll assembly.

The GraniteWorkflowDesigner user control was then updated to use this template as a style:

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary Source="./Templates/WorkflowElementDesignerDefaultTemplate.xaml" />

</ResourceDictionary.MergedDictionaries>

<Style TargetType="{x:Type sap:ActivityDesigner}">

<Setter Property="Template" Value="{StaticResource WorkflowElementDesignerDefaultTemplate}" />

</Style>

</ResourceDictionary>

In the WorkflowDesigner class code this activity designer style is then obtained from our Resources (see (1) below) and for any types that derive from the ActivityDesigner class, we apply this custom style for that type (see (2) below):

(1)

// Get our custom style to use for activity designers

Style activityDesignerStyle = Resources[typeof(ActivityDesigner)] as Style;

(2)

foreach (Type type in assembly.GetTypes())

{

// If this type is an activity designer

if (typeof(ActivityDesigner).IsAssignableFrom(type))

{

// Use our custom activity designer style

Application.Current.Resources[type] = activityDesignerStyle;

}

}

The workflow designer default template was then updated with adorners to show when an activity has passed or failed:

<Viewbox x:Uid="viewBoxPassedAdornment" Grid.Row="0" Margin="-20,0,-20,0" Width="16" Height="16" IsHitTestVisible="False" HorizontalAlignment="Right" Visibility="{Binding Path=ModelItem.ShowPassed, RelativeSource={RelativeSource AncestorType={x:Type swd:WorkflowViewElement}}}">

<Rectangle x:Uid="rectanglePassedAdornment" Width="16" Height="16" >

<Rectangle.Fill>

<ImageBrush ImageSource="/GraniteLibrary;component/Images\passed.circle.icon.32x32.png" />

</Rectangle.Fill>

</Rectangle>

</Viewbox>

<Viewbox x:Uid="viewBoxFailedAdornment" Grid.Row="0" Margin="-20,0,-20,0" Width="16" Height="16" IsHitTestVisible="False" HorizontalAlignment="Right" Visibility="{Binding Path=ModelItem.ShowFailed, RelativeSource={RelativeSource AncestorType={x:Type swd:WorkflowViewElement}}}">

<Rectangle x:Uid="rectangleFailedAdornment" Width="16" Height="16" >

<Rectangle.Fill>

<ImageBrush ImageSource="/GraniteLibrary;component/Images\failed.circle.icon.32x32.png" />

</Rectangle.Fill>

</Rectangle>

</Viewbox>

Properties were then attached to our workflow designer so that we could set the visibility of these adorners. For brevity, only the code below for the property that controls the adorner when an activity has passed has been shown:

/// <summary>

/// Attached property for the adornment on the designer when an activity has executed and passed.

/// </summary>

AttachedProperty<Visibility> \_attachedPropertyPassed = null;

// Create the attached property to show an adornment on the activity that has passed

\_attachedPropertyPassed = new AttachedProperty<Visibility>()

{

Name = "ShowPassed",

Getter = (modelItem) => ShowExecuteStatus.GetPassedVisibility(modelItem),

Setter = (modelItem, show) =>

{

if (show != ShowExecuteStatus.GetPassedVisibility(modelItem))

{

if (show == Visibility.Visible)

{

ShowExecuteStatus.AddPassed(modelItem);

}

else

{

ShowExecuteStatus.RemovePassed(modelItem);

}

// Notify that this property has changed

\_attachedPropertyPassed.NotifyPropertyChanged(modelItem);

}

},

OwnerType = typeof(Activity)

};

// Get the service for the attached properties

AttachedPropertiesService attachedPropertiesService = \_workflowDesigner.Context.Services.GetService<AttachedPropertiesService>();

// Add the attached property that show whether an activity has passed to the designer

attachedPropertiesService.AddProperty(\_attachedPropertyPassed);

The ShowExecutionStatus class is used to store the list of model items that have passed or failed and convert this to a Visibility value.

We can then set the visibility of the “ShowPassed” attached property based on a tracking record state once we have obtained the ModelItem for that activity. Note that we need to use the dynamic keyword in order to access the “ShowPassed” attached property that we have attached to the ModelItem:

// Get the model item with the attached properties

dynamic modelItem = \_trackingActivityIdModelItemDictionary[activityId];

if (activityStateRecord.State == ActivityStates.Closed)

{

// Update the attached property to indicate that the activity has finished executing and has passed

modelItem.ShowPassed = Visibility.Visible;

}

# Phone Display Reference Images

*This functionality was implemented after the Granite 1.0 release.*

There is a requirement to be compare phone display images with reference images and provide the same functionality that is available in STE, such as masking capabilities and multiple reference images.

# Plug-ins

*This functionality has not yet been implemented.*

The same workflow activities need to be made available from the Granite UI (when creating the test) and the Framework (when running the test). The two applications should therefore load the workflow assemblies and any dependencies from the same directory.

There is the possibility that some users of Granite that have the technical expertise to develop workflow activities could create their own activities and add them to the folder, or we could provide functionality for users to specify additional folders to load the workflow activities from. This above functionality is in STE/STA and it has worked very well for some groups that have special testing requirements. STE/STA also has functionality where workflow activities can be automatically retrieved from the cloud and we can release new versions that are immediately downloaded whenever a user starts up STE or runs STA, irrespective of where in the world they are running the test application.

It would be beneficial if the plug-in architecture could load the workflow activities in separate AppDomains. This would allow the assemblies to be unloaded. Functionality could then be provided to users so that they can select which plug-ins or plug-in directories they wish to use.



Figure 18 Overview of Granite with plug-ins

# Updating the Framework Interface

Updating the interface that is used to make calls back into the IronPython framework requires two interfaces to be updated. The first interface is the IFramework interface that is definied in the Granite.Studio.WorkflowEngine project. The workflow.py IronPython class in the Framework inherits from this IFramework interface. This is a standard .NET interface. If the IFramework interface is changed then the workflow.py code will also need to be updated accordingly. The second interface is the IFrameworkService interface that is a WCF interface (i.e. ServiceContract) that the workflow activities call. This provides the same methods and parameters as the IFramework interface except each method in an OperationContract.The FrameworkService class implements the IFrameworkService interface and calls the IFramework interface on the IronPython code. Essentially the FrameworkService links these two interfaces together so that calling the IFrameworkService WCF interface will call back on the IFramework interface. The IFramework interface, IFrameworkService interface and the FrameworkService class are all defined in the Granite.Studio.WorkflowEngine.dll assembly.

If you change all of these, such as when adding a new method, then we also need to update the client code that calls the IFrameworkService WCF interface. Since in WCF terms, we need to update the client code (i.e. the calls made from the workflow activities), we must follow the instructions listed in the \Studio\ActivityLibrary\FrameworkService\ create\_framework\_service\_client.cmd. Essentially this command script requires us to rebuild the project with a conditional compilation symbol and then run a unit test, so that the svcutil.exe program can generate the client code. If the instructions have all been followed correctly, then it should be possible to update everything to use modified interfaces. Once this has all been done, it is still necessary to run the \Stusio\copy\_studio\_assemblies\_to\_framework.cmd command script to copy the updated Granite.Studio.ActivityLibrary.dll over to the Framework so that the updated assembly will run from within the Framework.

During development the ExecuteIronPythonCode(string ironPythonCode) method can be used as a short-term measure for calling some custom IronPython code, but long-term it is better to update the interfaces.

# Testing of Activities

Activities should be tested in Granite Studio by checking that they work as expected when the user is interacting with the workflow designer, e.g. setting properties, collapsing or expanding the activities, switching from Studio to Test Lab and back, opening an existing workflow that contains the activity, etc.

In addition, before pushing any changes to GIT the following tests should be created and run:

1. A basic unit test of the Activity by creating a unit test class under the Activities folder in the Granite.Studio.ActivityLibrary.UnitTest project.
2. A basic unit test of the ActivityDesigner by creating a unit test class under the ActivityDesigner folder in the Granite.Studio.ActivityLibrary.UnitTest project.
3. A basic unit test that the Activity calls the correct method in the Framework by adding a workflow XAML file with the Activity in the Data folder of the Granite.Studio.WorkflowEngine.UnitTest project and creating a unit test in the WorklowExecutorTest class.
4. Build a Release version of the Granite solution, run \Studio\copy\_studio\_assemblies\_to\_framework.cmd that copies the Granite.Studio.ActivityLibrary.dll to the Framework and then start up Granite and run a worklow with the new Activity.  
   This verifies that no new assemblies are needed in the Framework in order to run the worklow. Test running from both Studio and the Test Lab.
5. Run all unit tests for the Granite solution and verify they all pass.  
   This helps to verify that the code changes have not introduced any side-effects.

# Tips and Tricks

This section provides some general tips and tricks when creating Activities and ActivityDesigners.

## Accessing the Activity from the ActivityDesigner

In some situations it may be necessary to get or set some properties on the Activity from within the ActivityDesigner code.

If for instance we have an Activity that has the following property:

public string IronPythonCode

Below is an example of how to set a property the IronPythonCode property on the Activity from within the ActivityDesigner:

string ironPythonCode = "self.capture()";

ModelProperty property = ModelItem.Properties["IronPythonCode"];

if (property != null)

{

// Set the value of the property on the activity

property.SetValue(ironPythonCode);

}

Here is an example of getting the IronPythonCode property value from the Activity:

ModelProperty property = ModelItem.Properties["IronPythonCode"];

if (property != null)

{

// Get the value of the property from the activity

string ironPythonCode = property.ComputedValue as string;

}

In theory, it is possible to have different types of activities using the same ActivityDesigner since the ActivityDesigner only uses binding in the XAML to bind the controls with properties on the Activity. When using the above mechanism to access the properties, ideally the code should allow for the possibility of the property being null.

# References

## Granite Studio Wiki page

<http://wikis.in.nokia.com/Granite/GraniteStudio>

## XAML

<http://en.wikipedia.org/wiki/Extensible_Application_Markup_Language>

## Windows Workflow Foundation 4.5

<http://msdn.microsoft.com/en-us/magazine/hh781025.aspx>

## STE and STA

<https://in.nokia.com/sites/DeliveryHub/SitePages/ste.aspx>

<https://in.nokia.com/sites/DeliveryHub/SitePages/sta.aspx>

# Appendix A – Backlog Items

**8.1 Workflow designer user control**

8.1.1 Basic workflow designer user control functionality

8.1.1.1 Create the Workflow Designer 4.0 user control and host it within a tab in the Granite UI.

8.1.1.2 Add a public method on the control to save the workflow as .xaml.

8.1.1.3 Add a public method on the control to load the workflow .xaml file.

8.1.1.4 Add a public method / property on the control to get and set the workflow .xaml.

8.1.1.5 Add a public method to clear out the existing workflow and give the default workflow.

**8.2 Granite UI**

8.2.1 Basic functionality to load and save workflow

8.2.1.1 In the Granite UI implement functionality to save the xaml from the workflow designer as a .xaml file.

8.2.1.2 In the Granite UI implement functionality to load the xaml a file into the workflow designer.

**8.3 Workflow in the Framework**

8.3.1 Basic workflow functionality in the framework

8.3.1.1 Add a socket message to run the workflow xaml.

The xaml should be inside the message that is passed to the Framework.

8.3.1.2 Add a socket message to run the workflow from a .xaml file.

8.3.1.3 Implement functionality to create the workflow runtime and run the xaml, or alternatively implement this functionality using the WorkflowApplication.

**8.4 Connect the Granite UI workflow designer to the Framework**

8.4.1 Add a UI control to the Granite UI for executing the workflow

This could be implemented as a button in a toolbar:



This could also be implemented in a menu:



8.4.2 Implement functionality in the Granite UI so that when a user chooses to execute the workflow, then the xaml is obtained from the workflow designer and sent to the Framework for execution.

8.4.3 Test and verify that a simple workflow can be created in the Granite UI and executed in the Framework.

**8.5 Loading workflow assemblies**

8.5.1.1 Add functionality in both the Granite UI and the Framework to load the assemblies containing the workflow activities.

8.5.1.2 The Granite UI needs to be able to update the designer toolbox with the workflow activities that have been loaded.

8.5.1.3 STE/STA allowed workflow activity plug-ins to be downloaded from the cloud as well as for the user to specify multiple paths where plug-ins could be loaded from. *We should consider whether any of this should be implemented on Granite or not.*

**8.6 Verify we can create and run workflow activities that call IronPython scripts**

8.6.1.1 Create a workflow activity that calls an IronPython script and verify it runs from within the Framework.

**8.7 Create a custom activity that allows a user to enter IronPython code directly in the activity**

8.7.1.1 The Granite team would like to be able to have a special workflow activity that would allow them to enter in IronPython code directly into the activity. When the activity executes, it should run the IronPython code that has been entered in.

8.7.1.2 There is a possibility that the activity itself may be able to modify its own appearance based on additional information that it is provided with.

This could include a bitmap image as an icon and potentially even dynamically add new arguments that are used in the IronPython script.

**8.8 Workflow designer tracking**

8.8.1 In workflow designer control add workflow designer glyphs that will be used to indicate to the user which part of the workflow is executing and which parts successfully executed or failed:

8.8.2 In the workflow designer control add methods to the workflow designer control to update the status of the activities on the designer.

8.8.3 In the Framework add message events sent via sockets to the Granite UI, to update the status with the workflow tracking information. Log the workflow tracking information.

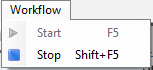
**8.9 Implement functionality to stop the workflow**

8.9.1 Add functionality in the Graphite UI to allow a user to stop a workflow that is running.

This could be implemented as a button in a toolbar:



This could also be implemented in a menu:



**8.10 Saving workflow test for a test run**

8.10.1.1 There should be some functionality to add the newly created workflow test to the test lab.

8.10.1.2 Create our own TestSequence activity that derives from Sequence and has properties to allow the user to enter in the test case id and title. Make this the root workflow for the workflow designer.

Add validation (error or warning) if the test case id and title has not been entered.

**8.11 Allow user to add breakpoints or step through the workflow**

8.11.1 Add functionality for a user to step into and step over workflow similar to how we would do this when debugging code.

This requires messages to be sent between the Granite UI and the Framework.

8.11.2 Add functionality to be able to add a breakpoint in the workflow.



This requires messages to be sent between the Granite UI and the Framework.

**8.12 Add a phone UI verification workflow activity**

8.12.1.1 Create a workflow activity that would retrieve information from the UX data and present the user with the display image, highlighting all of the controls, and allowing the user to select which controls to validate as well as what to validate (e.g. X and Y co-ordinates, text, etc.).

**8.13 Storing test cases on the cloud**

8.13.1.1 There is a requirement to store the test cases on the cloud.

This could be done using Microsoft Windows Azure. The tests could either be stored in a SQL Azure database, or in Windows Azure table storage. Technical solution would need to be designed as well as cleared with Nokia security.

8.13.1.2 The Granite UI would need to implement a mechanism where test cases can be published and downloaded from the cloud.

Test case versioning may be required and the Workflow 4.5 versioning information may potentially be able to be utilized.

**8.14 Using functionality from STE for reference image storage and comparison**

Background information:

STE has functionality to allow users to easily upload new reference images to a SQL Azure database stored on the Microsoft Windows Azure cloud. There is functionality in Silverlight web pages that allow a user to apply masks to areas of the reference image that they want to mask out. There is a STE DisplayImageComparer workflow activity that will compare the actual image from the phone with the reference image, taking into account any masks. Multiple reference images can be stored if there are a number of UI changes that are valid. An expiry data can optionally be specified for a reference image, so that older reference images will not be used after a particular date. The DisplayImageComparerActivity compares the current display image with the reference image. If no match is found, the activity produces an image showing the difference, with the reference image that had the closest match. It is possible to specify a tolerance on the activity if an exact match is not always required. It is also possible to adjust what is deemed as the ‘acceptable’ the Red, Green and Blue values if some minor differences in colour are deemed acceptable.

8.14.1.1 Produce a new cut-down database schema for Granite containing only the information that would be used by Granite.

8.14.1.2 Update the Silverlight web pages to produce a cut-down user interface that allows a user to manipulate the data in the new Granite SQL Azure database.

8.14.1.3 Implement functionality in the Granite UI to host a browser control in order to allow the user to manipulate the reference image and masks, or port the Silverlight code to WPF and provide the same functionality in the Granite UI.

8.14.1.4 Port the existing STE DisplayImageComparerActivity over to Granite.

8.14.1.5 Implement functionality so that a user can specify the area of interest (i.e. invert mask).

# Appendix B – Granite Studio Ideas

**From:** Utriainen Sami (Nokia-MP/Oulu)   
**Sent:** 23. helmikuuta 2012 8:22  
**To:** Pukinkorva Timo (Nokia-MP/Oulu); Pakanen Risto (Nokia-MP/Oulu); Pernu Jani (Nokia-MP/Oulu); Lepisto Pertti (Nokia-MP/Oulu)  
**Subject:** FW: Granite Studio ideas

Jou, listailin ylös asioita mitä mieleen juolahti.

-Sami

Granite Studio:

* Test activity workflow as WPF control (in order to place it to Granite GUI as tab page)
  + Test activity flow -> test case xaml -> test case python script
  + Test case xaml –> test activity flow
  + Test case xamls to MS Cloud
  + Schema for test case xamls
* Real time Ipy code editor
  + Updates when test case xml updates
  + Can be edited by users
    - Edits to be stored in test case xml as custom code (at least)
  + Syntax highlighting
* Phone Application xml designer:
  + Picture box for phone display captures
    - Intelligent object selection from captured image (with help of captures xml data)
    - Mouse event on image -> touch event on real device + capture and picture box refresh
  + Simple UI component editor for application xml (e.g. component name, position, size, content)
  + Import from Adobe illustrator application xmls, PNGs
  + Connection to localization DB needed for string ids?
  + Output: application xml
* Activities:
  + Intelliget recorder functionalities: navigator, etc.
  + Usable ones from STE
  + UI verification specific actions
  + Phone settings read/write
  + GUI settings read/write
  + Austere HW activities: sim switch etc.
  + REBO device activities
  + Trace activation/deactivation -> trace xml selection
  + SX script launcher (with simple, embedded sx script editor with syntax highlighting (nice to have))
  + Etc.

Capture Tool:

                User can navigate to a state in the phone and then take capture from that state.

                No need to run any test case -> No idle verification etc.

                Displays:

                -              real capture as a bitmap

                -              generated display layout (captured display elements drawn and positioned by the coordinates)

                -              all display elements captured by the granite server in a table

                -              properties listed for each element: position, size, string/anim id, text properties (highlight, font, bold, italic) etc.

                Highlight the correct area from the real bitmap with a red rectangle.

                Easy sending of the capture(s) e.g. via e-mail.

                Saving of individual captures and capture-sets (as e.g. an xml-file).

                Possibility for naming and commenting captures.

                Rapid capture functionality (take series of captures one after another).

                Let user adjust amount of total captures and delay between captures.

Testing helpers:

                Helpers and server components can be used without running a test case.

                e.g. Configuring phone settings, copying contacts, adding messages etc.

File browser:

                Phone file system can be browsed via GUI.

                Files can be copied or deleted.

Test script debugger:

                Possibility to set break points and observe execution status:

                Show script objects content (what we expect to see on the screen)

                Show captured display elements (what Granite server sees on the screen)

                Show real display capture (what the end user sees on the screen)

Immediate window

                Test case code can be written and run on the fly.

                Like python interactive interpreter.

Custom Verification steps:

                Show rectangle of size of the phone display.

                Verify points can be added by drawing a rectangle over the area.

                Parameters of the area can be modified (text, position, size etc.)

                Capture from the phone can be used as a base (see capture tool).

JaniP