LabVIEW

Software Engineering Demonstrations

NATIONAL INSTRUMENTS™

CONTENTS

INTRODUCTION TO SOFTWARE ENGINEERING

National Instruments LabVIEW is a complete programming language, suitable for handling the biggest and most complex applications that engineers and scientists face today. In particular, programmers creating mission-critical applications – embedded control applications, industrial monitoring applications, and high-performance test systems – cannot afford to introduce errors or uncertainty into the system. For these kinds of applications, a very structured, and in some cases externally certified, programming process must be followed to ensure quality and repeatability of the code developed.

Software engineering typically refers to a regimented and procedural methodology for developing software. As software has gotten more complex and team sizes have grown, various software engineering process models have evolved to encourage efficient development that ensures quality, follows timelines, and meets the expectations of the end-user.

This guide will examine the development life-cycle and explain some of the tools that can improve and automate common software engineering practices.

SOFTWARE CONFIGURATION MANAGEMENT

Many developers have experienced the frustration of unmanaged environments, where people overwrite each other's changes or are unable to track revisions. Managing a large number of files or multiple developers is a challenge in any language. In fact, it's often a challenge to manage an application even if it's just one developer working on a small to medium application. Large development projects rely upon configuration management tools to satisfy the following goals:

1. Define a central repository of code
2. Manage multiple developers
3. Detection and resolution of code collisions
4. Tracking behavioral changes
5. Identification of changes are who made them
6. Ensuring everyone has latest copy of code
7. Backing up old code versions
8. Managing *all* files, not just source code

Perhaps the most important and commonly known SCM tool is source code control (SCC). However, in addition to many third party SCC tools, we'll see that there are a number of additional tools available in the LabVIEW development environment that is designed to help with these goals.

Establishing guidelines for storing and managing files requires foresight into how the application will be structured, how functionality will be divided, and the types of files beyond source code that will be important to keep track of. Devote time to making decisions about how functionality will be divided among code and to working with developers on file storage locations and the additional files or resources they will need to function properly.

Tracking Changes to VIs Using Source Code Control

Goal

We want to be able to download, track and manage our source code using a third-party source code control tool. For this example, we will be using TortoiseSVN as our source code control client.

Scenario

We are developing a LabVIEW application with the help of a team of developers. In preparation for a code review, we want to compare our most recent changes with the previous version.

Description

We are going to download the most recent code and be able to compare changes we make with previous versions using the graphical differencing feature of LabVIEW. After making undesirable changes and saving them, we will be able to revert to a previous version.

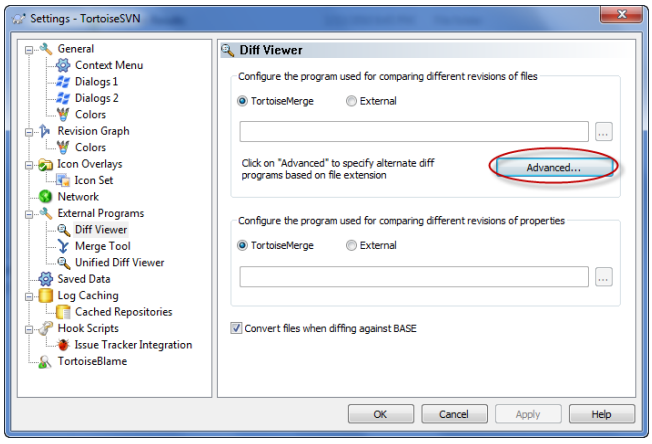
Concepts Covered

- The Project Explorer
- Tracking changes with source code control
- Graphical differencing from outside the development environment
- Reverting to a previous version

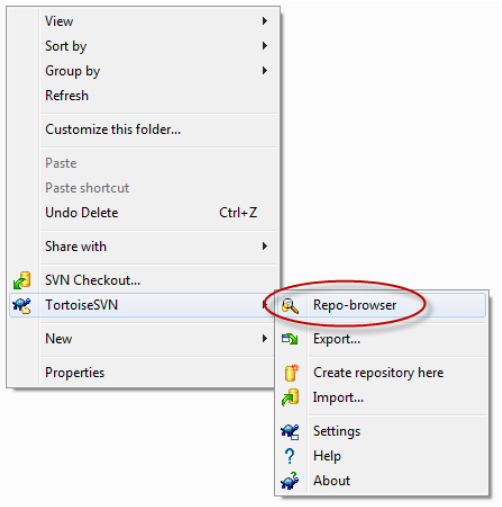
Setup

- Ensure that LabVIEW and TortoiseSVN are installed
- If using LabVIEW 2009, enter 'skipSVNFolders=true' into LabVIEW.ini token (does not apply to 2010)
- Make sure TortoiseSVN is calling LVCompare.exe for graphical differencing
    - Right click in Windows Explorer
    - Select **TortoiseSVN > Settings**
    - Select **Advanced** and enter the following for a .vi file type (this can also be used for a .ctl)

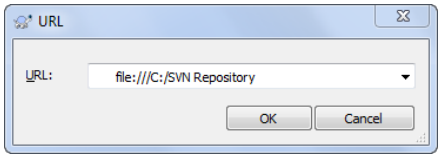
"C:\Program Files\National Instruments\Shared\LabVIEW Compare\LVCompare.exe" %mine %base -nobdcosm -nobdpos



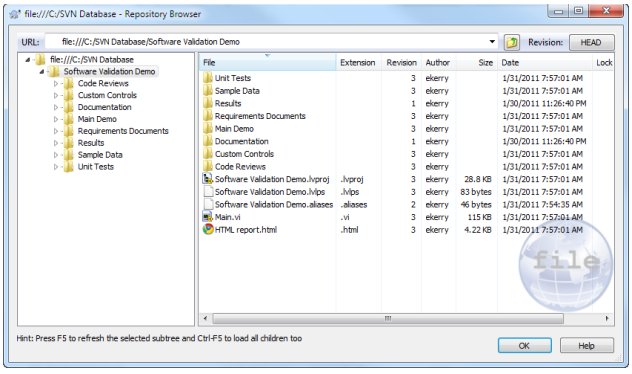
1. Introduce the TortoiseSVN interface and download the latest version of the LabVIEW project from the source code control repository.
    a. Open the folder where you would like to download the application
    b. Right-click in a blank explorer window and select **TortoiseSVN > Repo-browser** from the right-click menu



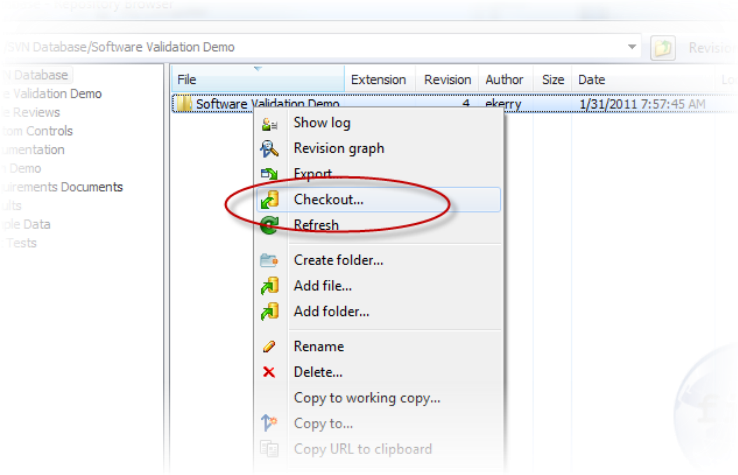
    c. Type the location of the Subversion repository in the dialog that appears. Note, for a local location, use the following syntax: 'file:///C:/SVN Database/'



    d. The dialog that appears will allow you to navigate and view the contents of the Subversion repository. By default, the browser will show you the most recent revision (also referred to as 'head' revision).

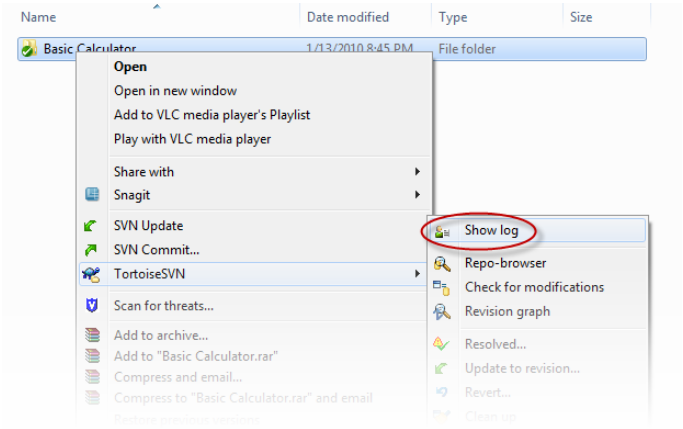


    e. Right-click on the 'Software Validation Demo' folder and select **Checkout** to download a copy of the head revision

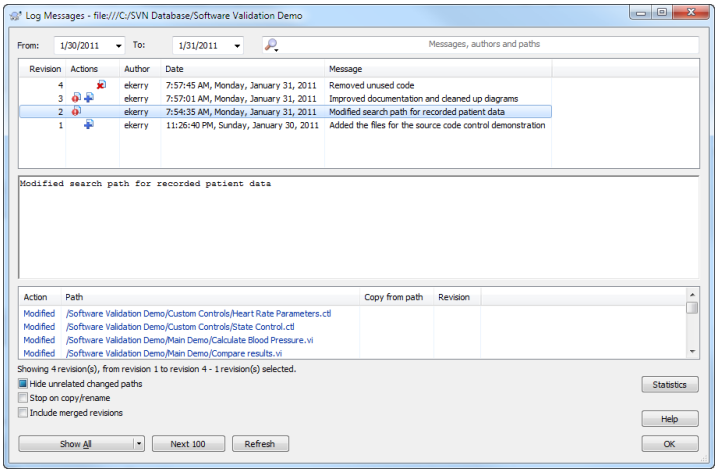


f. Clicking **OK** in the checkout dialog will download the most recent version of the code to the folder you right-clicked in

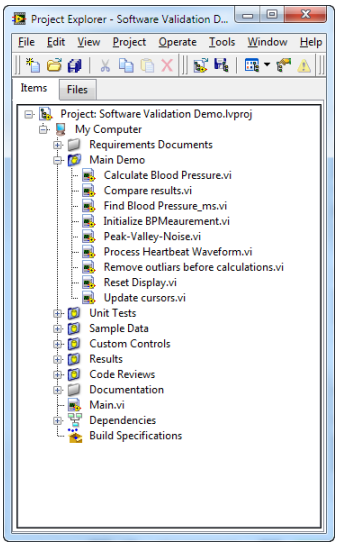
2. View the history of revisions
   a. Right click on the root folder and select **TortoiseSVN > Show Log**



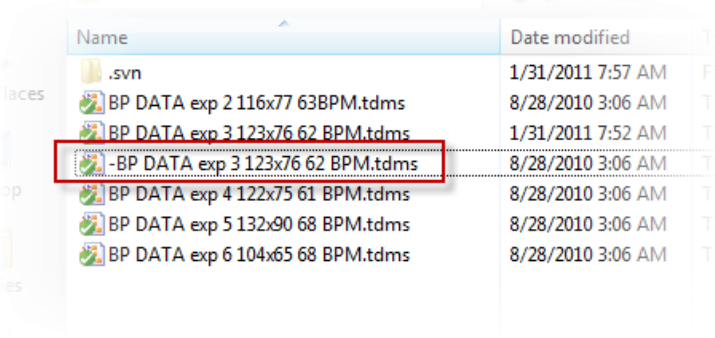
   b. The window that appears shows a history of revisions and details including developer, time, date and notes that were entered. We can download any and all of these older versions and compare them with our current working copy.



3. Introduce the Application
    a. Open **Software Validation Demo.lvproj**
    b. Note that the project contains subVIs, test configuration, build specifications and various other files. Open **Main.vi.**



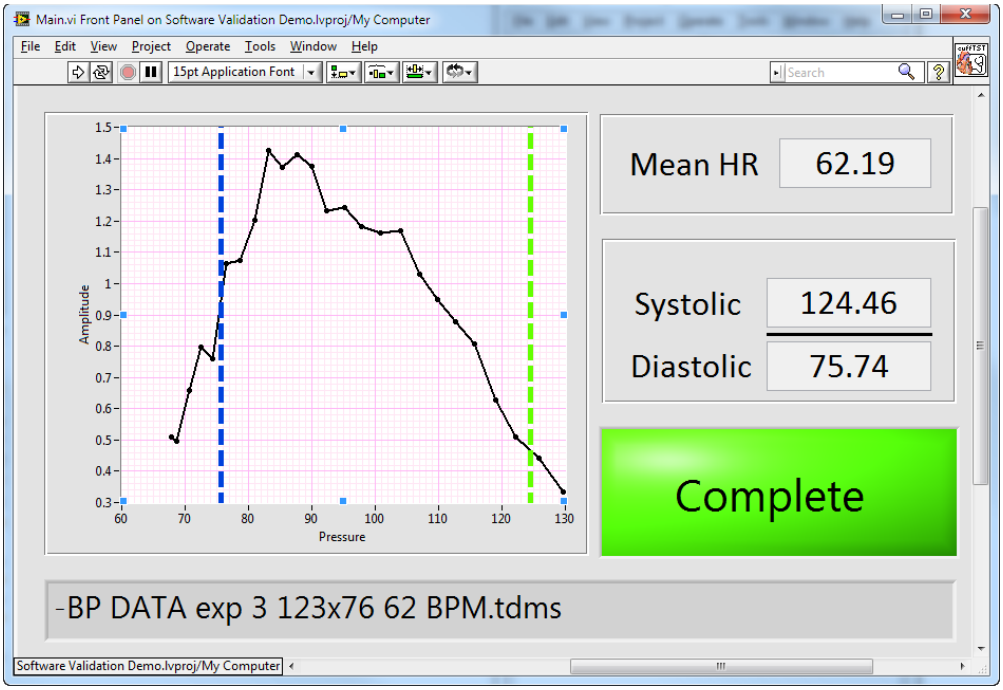
    c. This application simulates a device that computes blood pressure based upon input from a pressure transducer. Instead of connecting hardware, this version will compute the ratio of diastolic to systolic pressure based upon recorded patient data. In order to compute this value, the application uses a very simple state machine that includes some basic acquisition (in this case, from a file), simple filtering and signal processing. Click **run.**
    d. When prompted, select a recorded dataset. ('BP DATA exp 3 123x76 62 BPM.tdms' is recommended, as the front panel defaults are calibrated for this patient).



e. Click 'Take Blood Pressure' to begin playback of the recorded acquisition.

f. When ready to move on to the next step, de-select 'Timming' *Note: this control is intentionally mis-spelled, as we will detect this in a later analysis step.*

g. The final screen should display the results of the analysis, as shown below:



4. Make changes and compare them with the previous version.

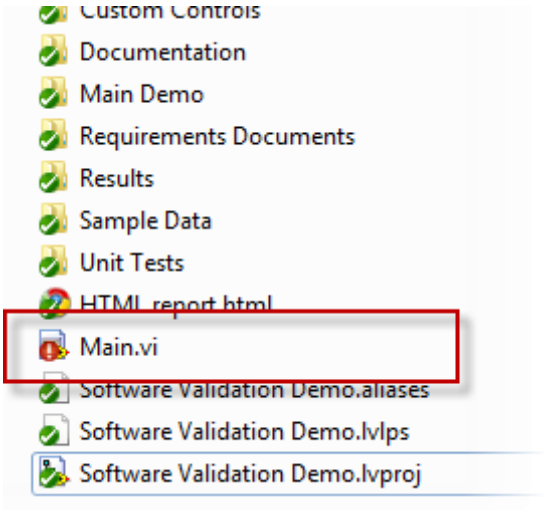
   a. Switch to the block diagram of the application.

   a. Make several changes that could introduce bugs, unexpected behavior or even break execution. Suggested modifications:

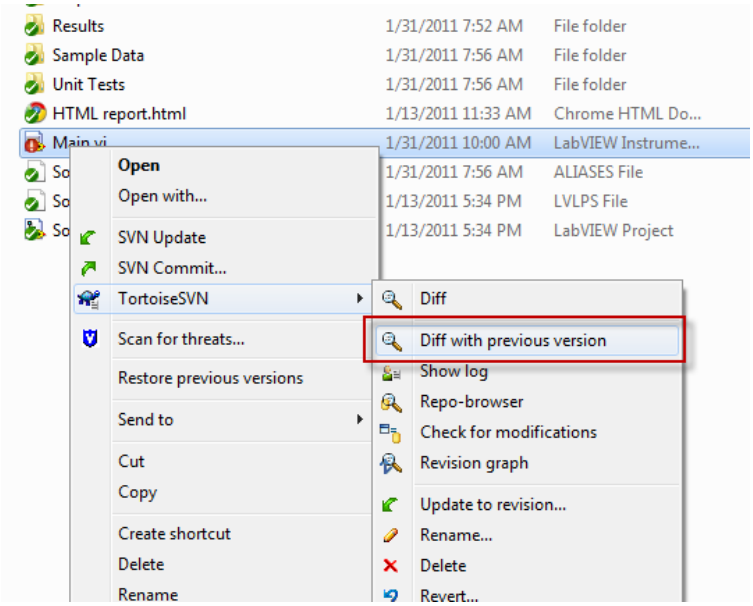
      1. Change timing parameters (especially hard to find, but can cause significant problems)
      2. Add cases to case structures
      3. Change block diagram constants
      4. Delete and/or move code

   b. Save the modifications by selecting **File > Save**, thereby overwriting the VI on disk.

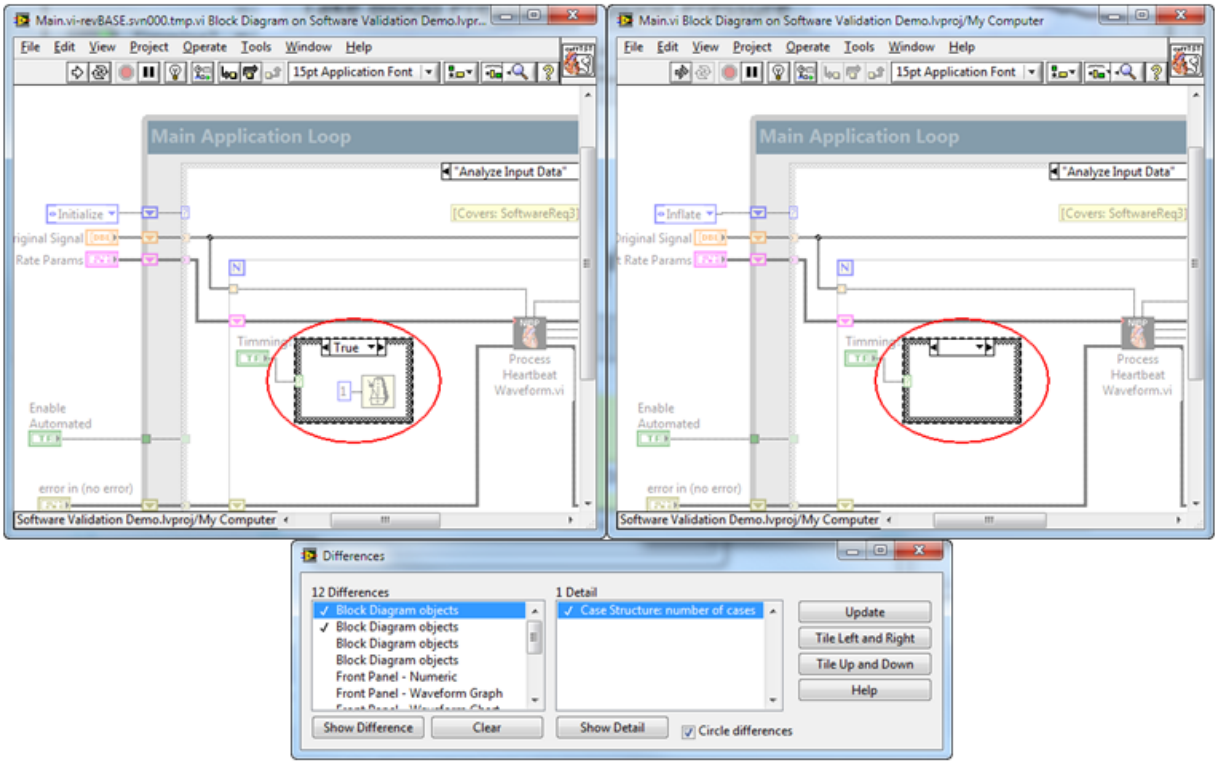
   c. Examine the files on disk and note that **Main.vi**, which has been modified, has a red exclamation mark over the icon of the file. This is how Subversion indicates that a file has modifications that have not been added to the repository.



d. We can compare the changes we've made with the latest version in source code control by right clicking on the modified file and select **TortoiseSVN > Diff with previous version**



e. This will launch LVCompare.exe, showing a side-by-side comparison of objects on the front panel and the block diagram.



f. Double clicking on the items in the list will place a check mark next to them, indicating that you have examined and reviewed every change.

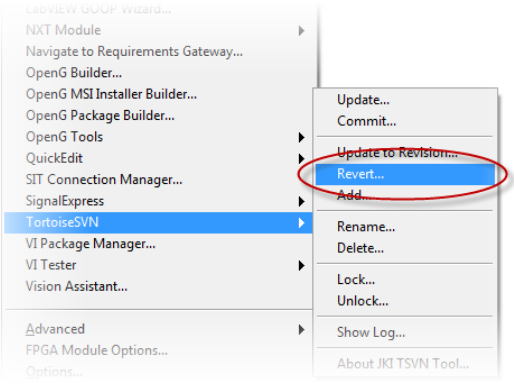
g. Click the 'X' in the 'Differences' window to close the dialog

2. Revert the VI to the previous version.

a. We can undo the changes to the VI by recalling the last version from source code control. There are two ways to do this:

1. Revert from inside the LabVIEW development environment *Note: requires TortoiseSVN Plugin from JKI Software* jkisoft.com

a. Select **Tools > TortoiseSVN > Revert**

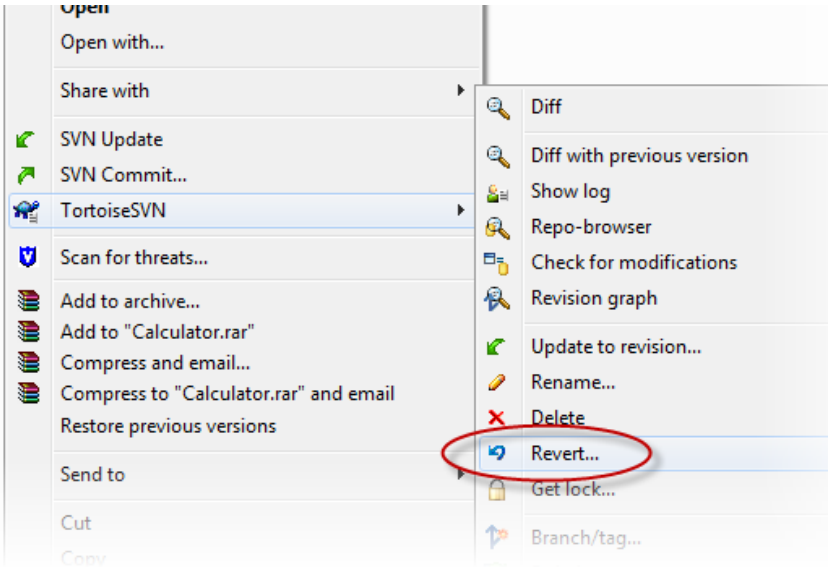


b. Click **OK**

c. The unmodified, working version of **Main.vi** will replace the modified one.

2. Revert from Windows Explorer using the TortoiseSVN interface

a. In Windows Explorer, right-click on Main.vi and select **TortoiseSVN > Revert…**



b. Click **OK**

c. The unmodified, working version of **Main.vi** will replace the modified one.

TRACKING REQUIREMENTS COVERAGE

Most engineering projects start with high-level specifications, followed by the definition of more detailed specifications as the project progresses. Specifications contain technical and procedural requirements that guide the product through each engineering phase. In addition, working documents, such as hardware schematics, simulation models, software source code, and test specifications and procedures must adhere to and cover the requirements defined by specifications.

Requirements gathering is important in order to ensure that you and your customer have come to the same agreement about what the application will do. The granularity of the documents directly depends upon the needs of your application and the criticality of it. For mission-critical systems, it's typical to go as far as to define the requirements for individual modules of code, code units, and even the tests for those units. Part of this process requires having reached an agreement of what is expected behavior and how the system should perform under any and all conditions.

Nebulous or vague specifications for a project can lead to a result that does not meet customer expectations. Consider an example where you are asked to build an automobile, but given no additional information. It's unlikely that the finished product would resemble what the customer had in mind. They may have expected a two-door car with a sunroof, but you built a convertible. Even in scenarios where they aren't required, insisting on extensive documentation of requirements, complemented by reviews of proof of concepts and prototypes, greatly increases a project's likelihood of success.

Prototyping and proof of concepts are a very important step towards developing requirements. It can be very hard to account for all contingencies and foresee all the ways in which the software will behave. Proof of concepts are also extremely valuable because they give the end-user or customer a feel for what the product will do, which helps developers and users come to a consensus. It is largely this principle upon which the Agile development method was derived, which emphasizes repeated and frequent iterations between development.

One of the biggest challenges of development, in any language, is tracing the implementation to the requirement or specification it was supposed to fulfill. From a project management standpoint this is important in order to gain inside into how far into the project you are. When requirements change, it is also valuable to have record of what other specifications or the implementation covering them may also be affected.

The software industry has a wide variety of tools at their disposal for managing specifications and requirements. Common tools include Telelogic DOORS and Requisite Pro. National Instruments provides a tool to automate integration with these products, called NI Requirements Gateway. NI Requirements Gateway facilitates the tracking of requirements coverage for these three types of documents.

EXERCISE: TRACING CODE TO REQUIREMENTS DOCUMENTS

GOAL

Developers who have been given or defined requirements should be able to document when and where the requirements are covered in their application to show that they have done what they were supposed to do. Our goal is to track and understand the percentage of the requirements that have been met and where. We also need to be able to create traceability matrices and other forms of documentation.

SCENARIO

We've been given requirements for a simple application – we need to document that we've implemented it.

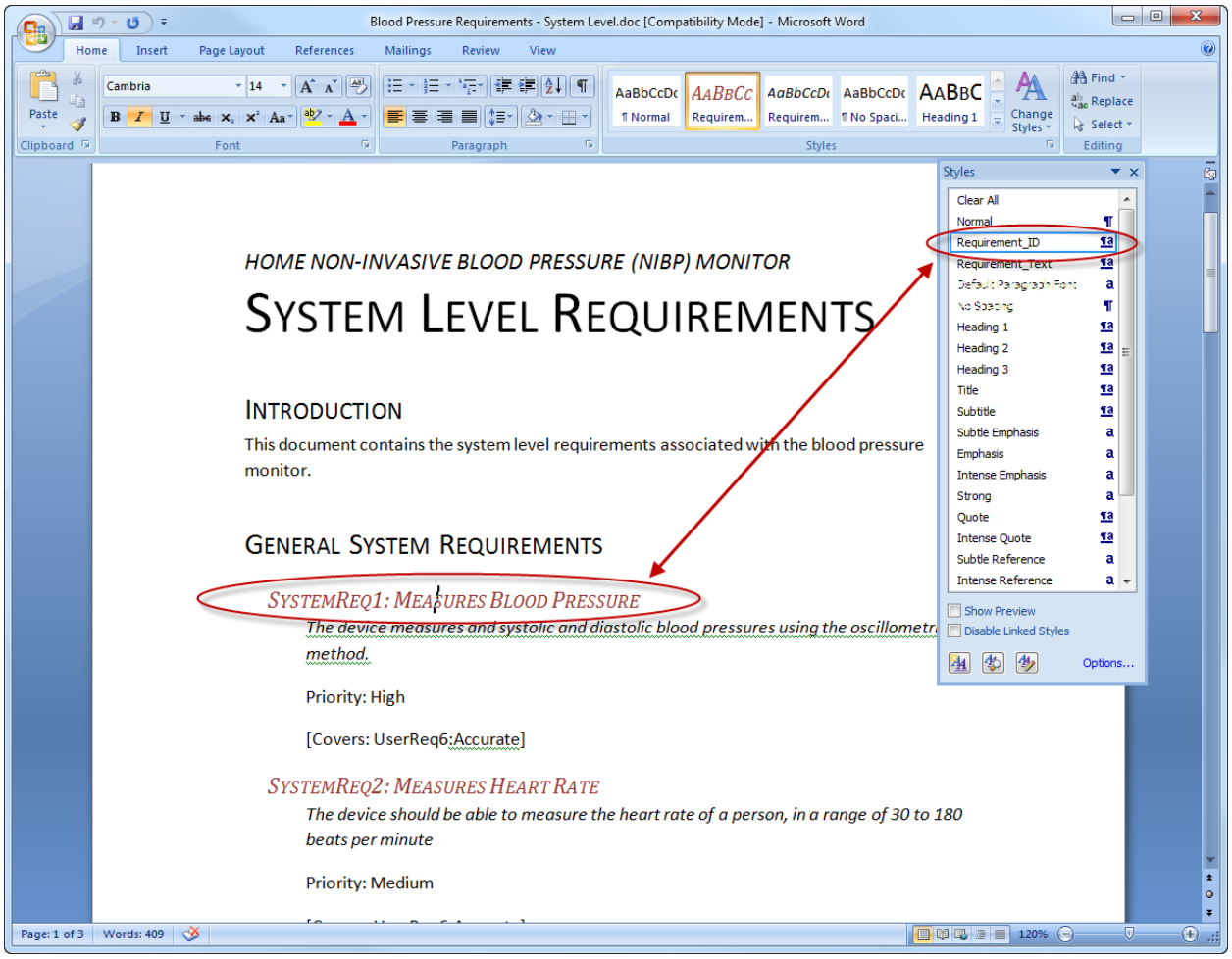
DESCRIPTION

We are going to use NI Requirements Gateway to parse requirements documents written in Microsoft Word and generate reports. *Keep in mind that requirements could also be stored in DOORS, Requisite Pro, Excel, PDF, any many other standard formats.*

CONCEPTS COVERED

- Documenting code and requirements coverage
- Tracking requirements coverage percentage
- Generate traceability matrices and documentation

1. Document a new function in an application as having covered a requirement
    a. Open the requirements document, 'Blood Pressure Requirements – System Level' in Microsoft Word. It should be stored on disk within the hierarchy, under the folder 'Requirements.'
    b. Familiarize yourself with this simple requirements document. Note that these requirements are extremely high-level (and therefore difficult, if not impossible to test against or to 'cover' with an implementation. As a result, it will be necessary to use these high-level requirements to derive lower-level, more specific requirements.

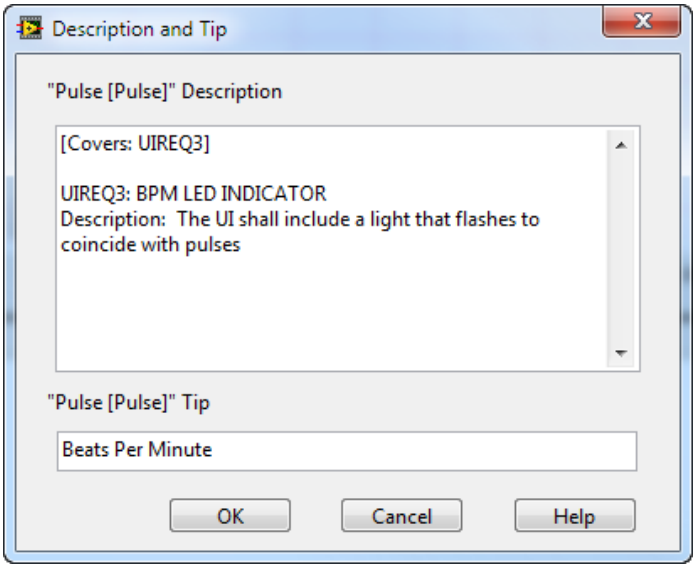


    c. Select one with your cursor and click on 'Styles' in the ribbon to observe that this text has been selected as a Requirements_ID. This will be used to automate the parsing of this document in later steps.
    d. Return to the folder containing the requirements document and open 'Detailed Software Design.docx.' This contains very specific requirements for the implementation and design of the software, which will actually be covered by the implementation in code. Note that it's divided into two main sections: State Implementations and GUI Components
    e. Scroll down to the last section, on GUI Component Requirements. In this scenario, we were given the requirements and asked to implement a UI component for the BPM Indicator. The requirement as stated is, "Description: The UI shall include a light that flashes to coincide with pulses"

f. Open 'Software Validation Demo.lvproj' and open the Front Panel of 'Main.vi.' This version of the code has the required function successfully implemented for the BPM Indicator, we just need to document it.

g. To document that this functionality has been implemented, right-click on the border of the indicator and select 'Description and Tip.' This is where we will place the appropriate tag such that we can automatically parse and trace the relationship between this component and the actual implementation.

h. In the description field, type "[Covers: UIReq3]." Note that you should also include any other relevant information in this field.



i. Close the Description and Tip dialog by clicking **OK**

j. Save the VI by pressing [CTRL + S]

2. Create a project in NI Requirements Gateway

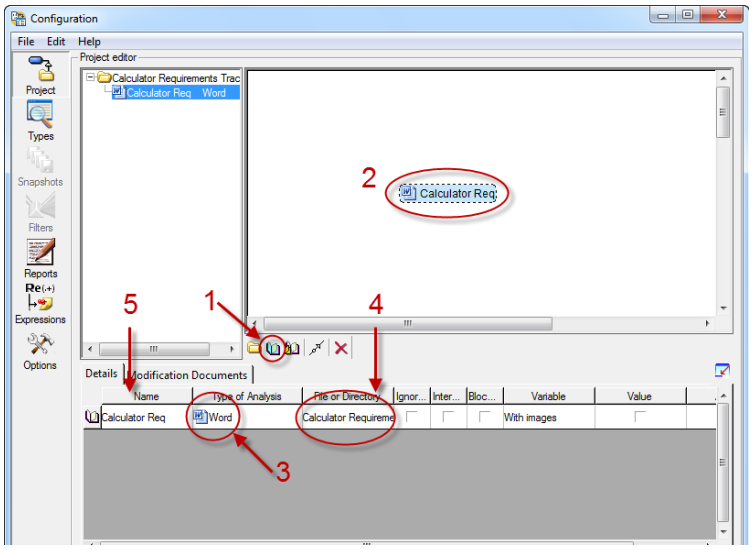
a. From the Windows Start Menu, launch Requirements Gateway 1.1.

b. Select 'File > New' and save a new Requirements Gateway project in the same folder as the requirements document named 'Calculator Requirements Tracking.' Do not save the project on the deskop, as this is not supported due to windows UAC.

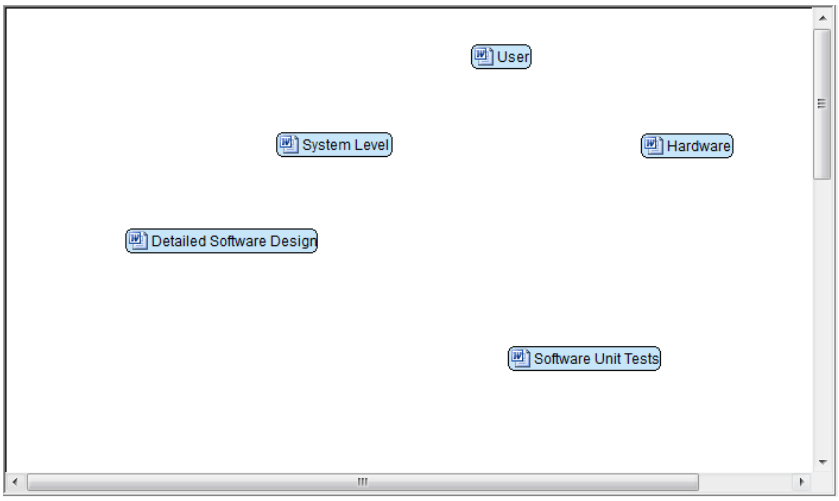
NOTE: *A completed version of this project has been included. If time is running out, open the pre-built copy of 'NIBP Monitor.rqtf' to see a working Requirements Gateway solution.*

c. The Configuration Dialog will appear, follow the steps below to import the Requirements Document as shown in the image below

1. Click on 'Add a Document'
2. Place the container for the document in the main window
3. Select the type of document from the drop-down list as 'Word' (note, not WordX). Use this as an opportunity to browse the numerous other types that are supported.
4. Select the location of the requirements document on disk
5. Type a name for this document

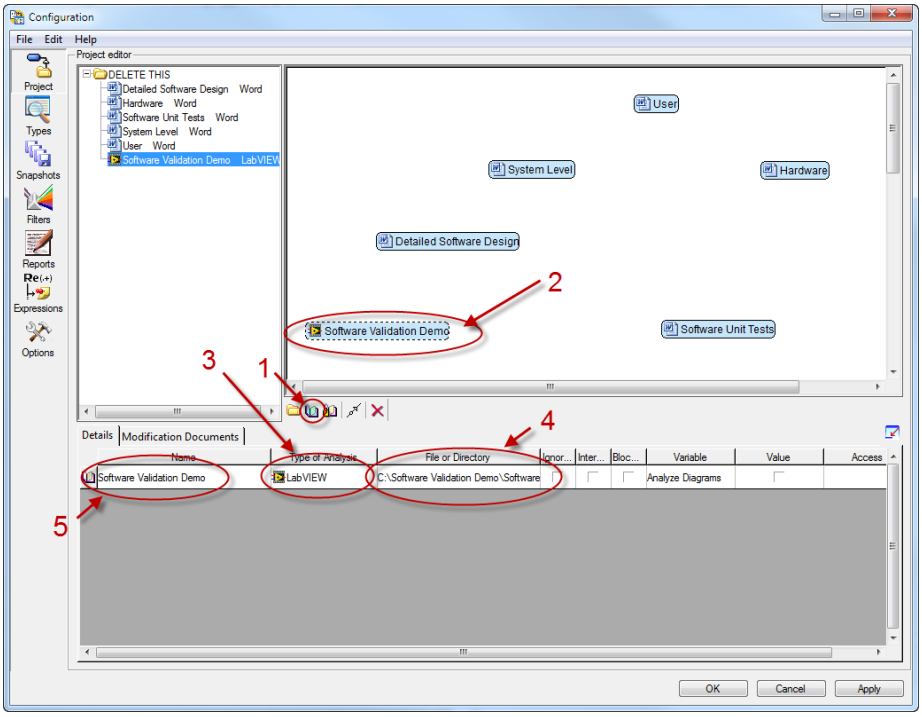


b. To add the remaining documents position the folder containing the word documents so that you can see them and the NI RG Configuration dialog. Select and drag all of them into the display and arrange as shown below (Note, the Unit Test Requirements will not be used until a later exercise).



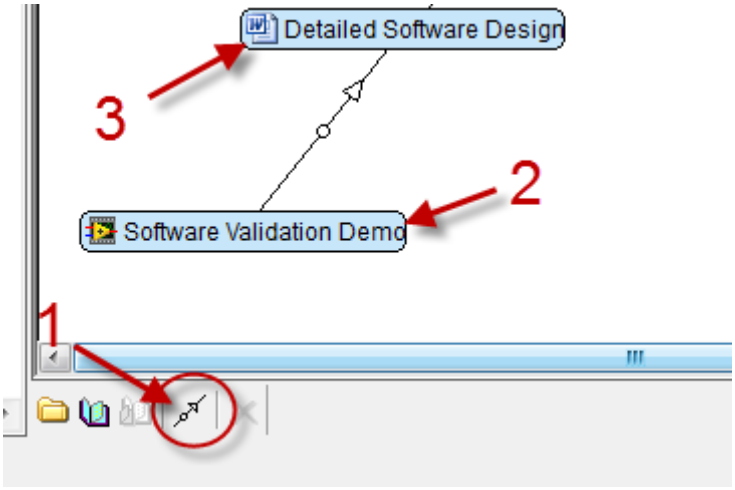
d. Repeat this process to add the LabVIEW Project to the Requirements Gateway Project:
   1. Click on 'Add New Document'
   2. Place the container for the LabVIEW Project in the main window
   3. Select the type of document from the drop-down list as LabVIEW
   4. Locate the LabVIEW Project File on disk (Basic Calculator.lvproj)
   5. Create a custom name for this item

***Note**: If you decide to open the pre-build NI RG solution, be sure that the directory path for the LabVIEW Project is correct.*

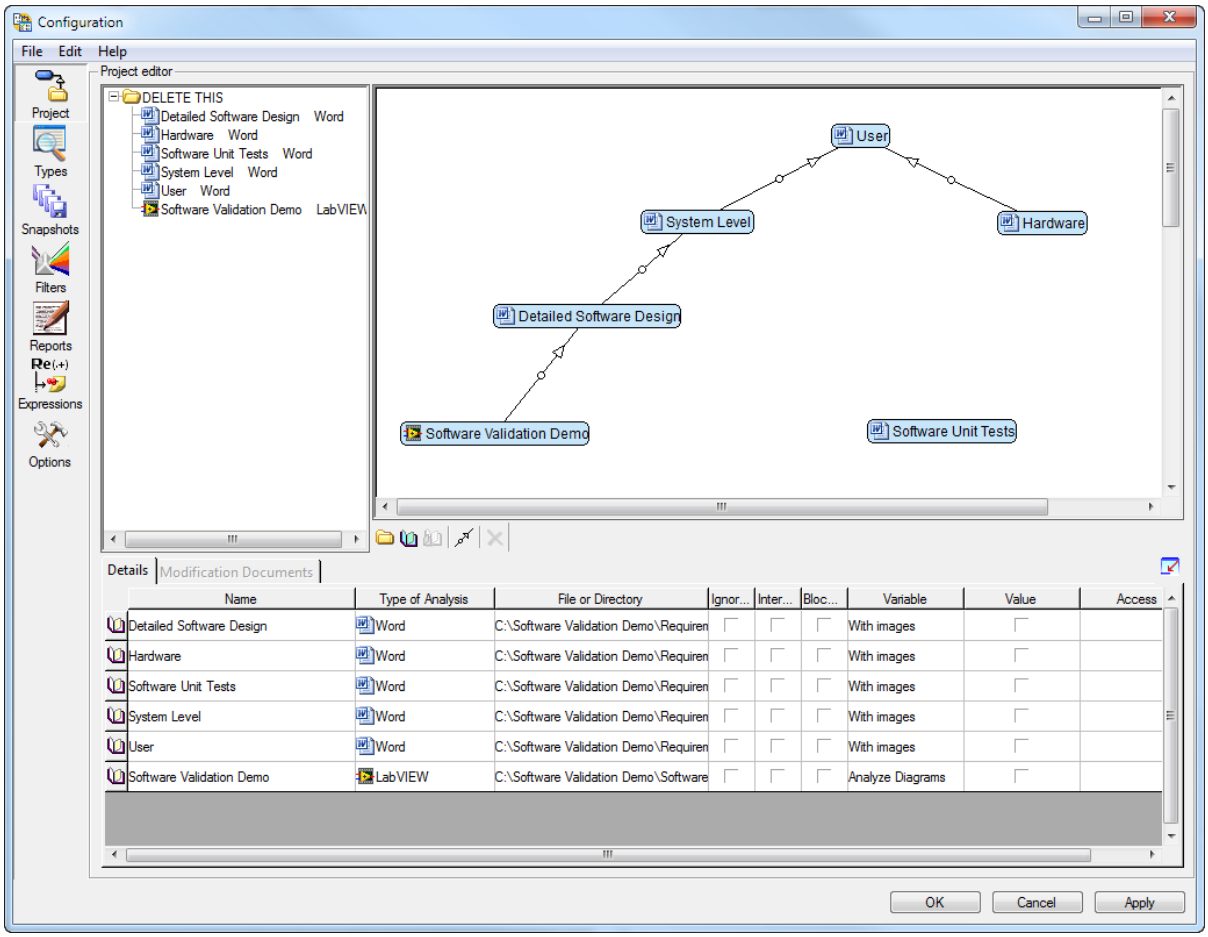


e. Define the relationship between these two documents. The code covers the requirements documents, so we need to draw an appropriate link between them. Follow the directions below:

1. Click on 'Add a Cover'
2. Click on the LabVIEW document to begin drawing the arrow
3. Click on the Requirements document to indicate that it is covered by LabVIEW

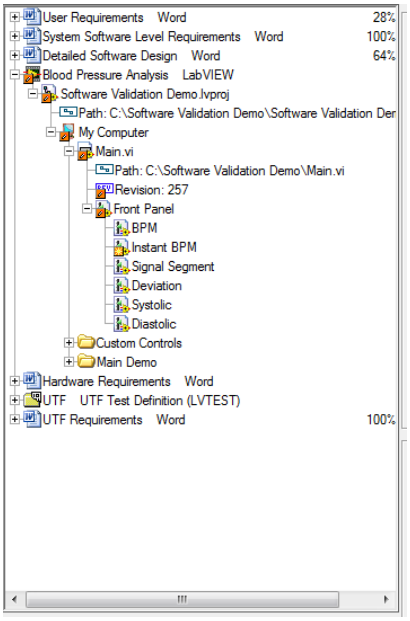


f. Repeat this process until the following relationships have been built:

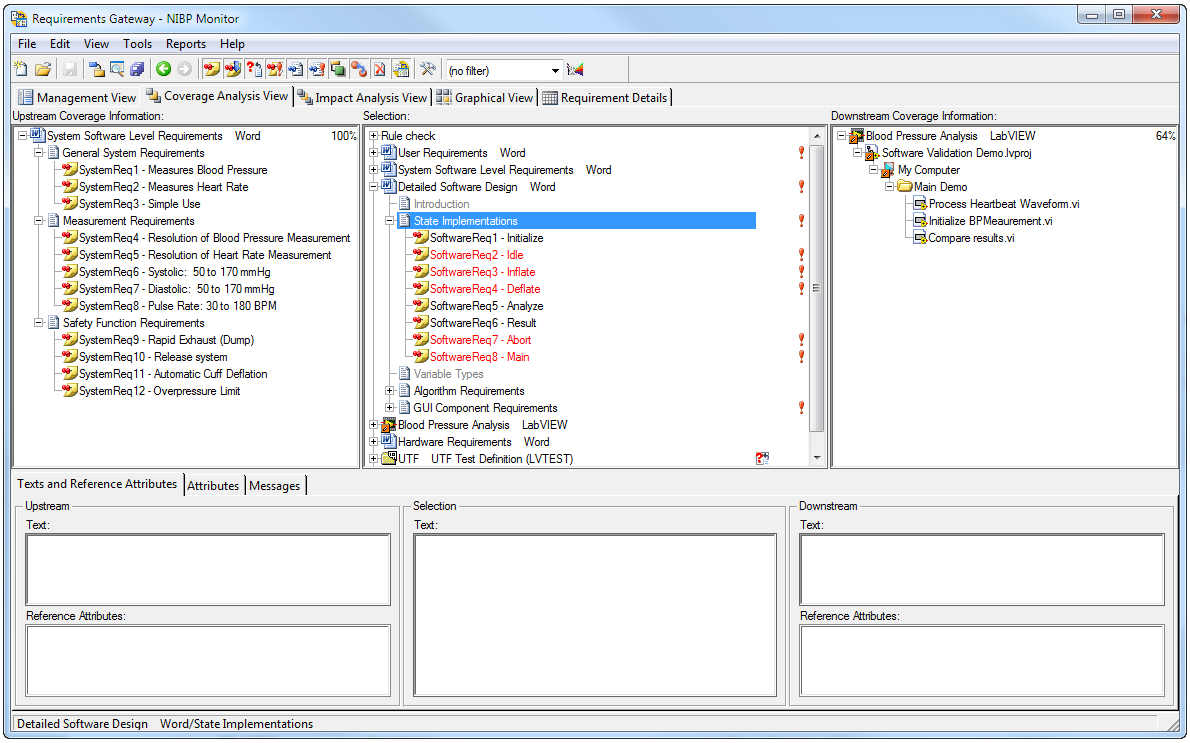


g. Click 'OK' to exit the configuration dialog. Press [CTRL + S] to save the Requirements Gateway project.

h. In the Management View Tab, expand the two documents to verify that Requirements Gateway has successfully parsed their contents.



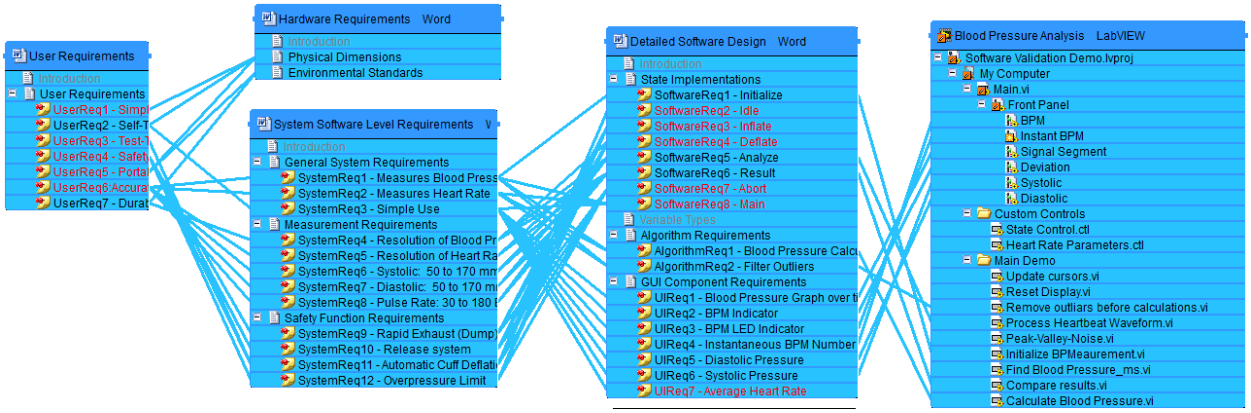
i. Notice that requirements coverage is less than 100%. Click on the 'Coverage Analysis View' to see the list of uncovered requirements.



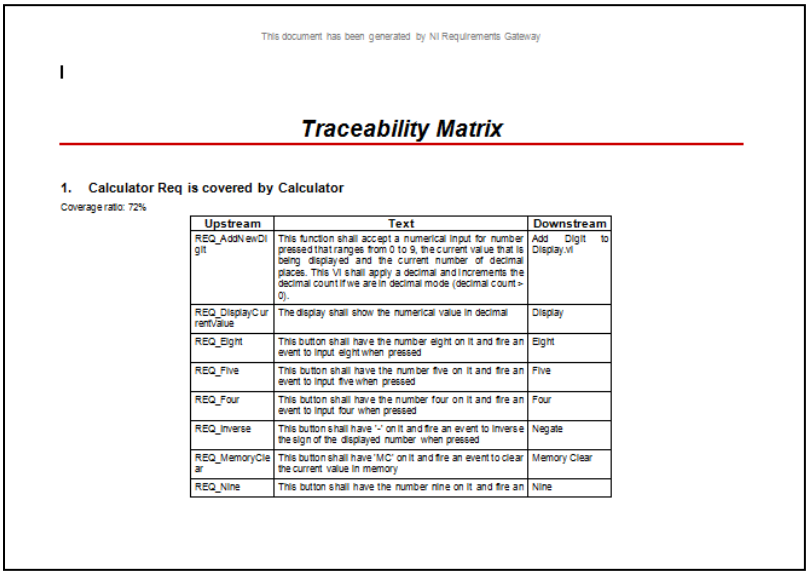
j. Click on the 'Graphical View' to see a graphical relationship between the requirements

3. Generate documentation showing requirements coverage

a. In the graphical view, highlight what you want to include in the report. Hold CTRL while selecting both the requirements document, and the LabVIEW Project.



b. Click on 'Reports > Library Reports > Traceability Matrix'
c. Ensure that you've selected all the items to include and click 'Continue'
d. In the save dialogue that appears, note the different formats that are available. Select PDF and select the desktop and type 'Blood Pressure Traceability Matrix'
e. The traceability matrix will appear



This document has been generated by NI Requirements Gateway

Traceability Matrix

1. Calculator Req is covered by Calculator

Coverage ratio: 72%

Upstream	Text	Downstream
REQ_AddNewDigit	This function shall accept a numerical input for number pressed that ranges from 0 to 9, the current value that is being displayed and the current number of decimal places. This VI shall apply a decimal and increments the decimal count if we are in decimal mode (decimal count > 0).	Add Digit to Display.vi
REQ_DisplayCurrentValue	The display shall show the numerical value in decimal	Display
REQ_Eight	This button shall have the number eight on it and fire an event to input eight when pressed	Eight
REQ_Five	This button shall have the number five on it and fire an event to input five when pressed	Five
REQ_Four	This button shall have the number four on it and fire an event to input four when pressed	Four
REQ_Inverse	This button shall have '-' on it and fire an event to inverse the sign of the displayed number when pressed	Negate
REQ_MemoryClear	This button shall have 'MC' on it and fire an event to clear the current value in memory	Memory Clear
REQ_Nine	This button shall have the number nine on it and fire an	Nine

PERFORMING CODE REVIEWS

Regular and thorough code reviews are an important and common practice for software engineers seeking to mitigate the risk of unforeseen problems, identify the cause of bugs that are difficult to find, align the styles of multiple developers, and demonstrate that the code works. These reviews are an opportunity for a team of qualified individuals to scrutinize the logic of the developer and analyze the performance of the software.

Peer reviews are sometimes referred to as a code 'walk-through.' The reviewer is typically guided through the main path of execution through the program by the developer, during which they should be examining the programming style, checking for adequate documentation, and reviewing questions that can be common stumbling blocks, such as:

- How easily can new features be added in the future?
- How are errors reported and handled?
- Is the code modular enough?
- Does the code starve the processor or use a prohibitive amount of memory?
- Is an adequate testing plan in place?

One of the most common reasons for not performing a code review is the amount of time needed to prepare for and then perform the review. In order to simplify the process, you need to take advantage of tools that can help automate the code inspection and help identify improvements. One example is the LabVIEW VI Analyzer tool, which is an add-on for LabVIEW 7 and 7.1 that analyzes any LabVIEW code and then steps the user through the test failures. You can also generate reports that allow you track code improvements over time, and can be checked into source code control software along with your VIs.

EXERCISE: ANALYZING CODE QUALITY

GOAL

We want to analyze our code on a regular basis to identify any potential problems or coding errors that could cause inappropriate or incorrect behavior.

SCENARIO

We're going to be configuring a series of tests, examining the results, and generating a report to document the results.

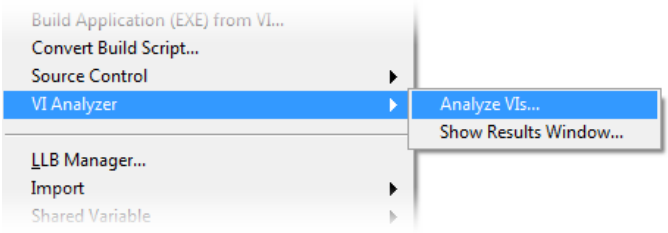
DESCRIPTION

The NI LabVIEW VI Analyzer Toolkit will be used to run 70+ tests on our application hierarchy and generate an HTML report.

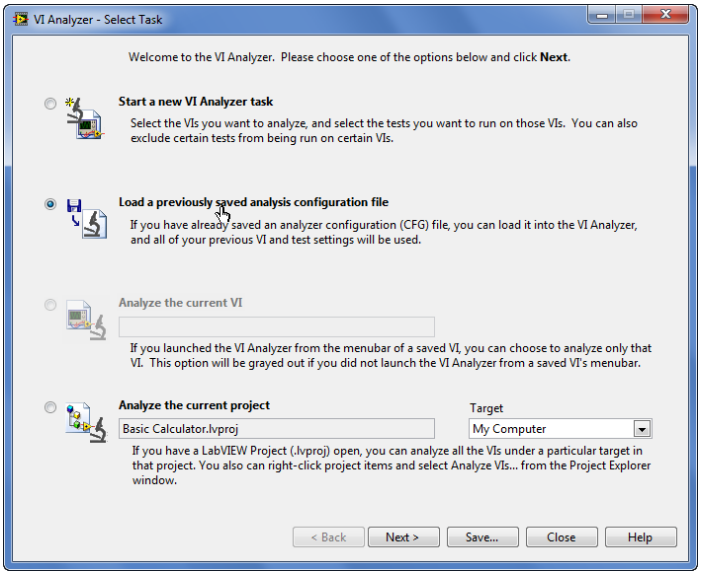
CONCEPTS COVERED

- Loading pre-configured test configuration
- Report generation

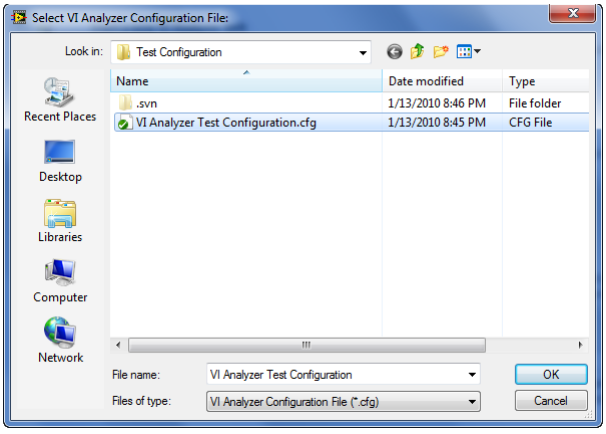
1. Launch, configure and run the analyzer tests
    a. From within LabVIEW, select **Tools > VI Analyzer > Analyze VIs…**



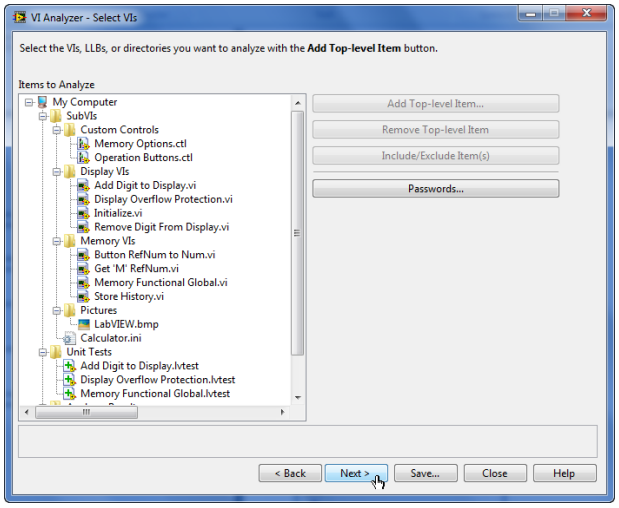
    b. Select the task labeled ‘Load a previously saved analysis configuration file’ and click **Next**



    c. VI Analyzer allows us to customize test settings and save the configuration for future use. Navigate to ‘Software Validation Demo > Code Reviews,’ and load the **Demo Configuration.cfg** file.



    d. VI Analyzer will display a list of files that will be analyzed. We can use this window to add or remove objects. Since all the files we want to analyze have been selected, click **Next**.



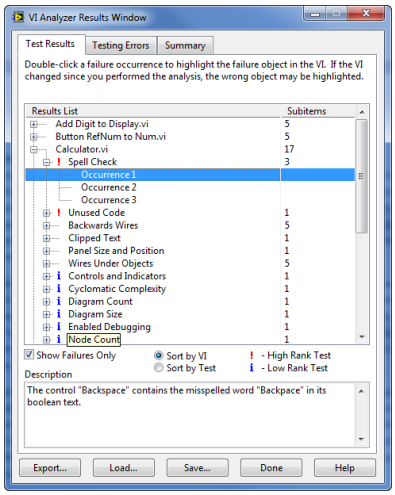
e. A list of over eighty tests will be displayed. Select a test to view configuration information and set the priority. Recommendations include:
    1. Documentation > User > **Spell Check** – this can help mitigate the risk of misspelled words amongst the documentation and most importantly, the user interface
    2. General > VI Properties > **Driver Usage** – when building an application, it can be useful to know what drivers have been called by the application and should be included in the installer
    3. Complexity Metrics > **Cycolmatic Complexity** – this industry-standard code metric helps evaluate the amount of paths through code, which is useful when developing test plans.
    4. Block Diagram > Performance > **Arrays and Strings in Loops** – this is one of several tests that can point out programming practices that could detract from execution speed.

f. Click **Next**

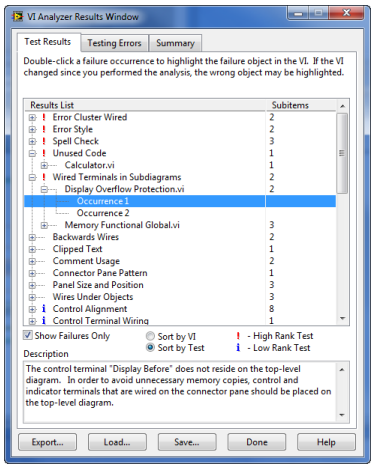
g. We can now save the test configuration, or we can perform the analysis on our VIs. Click **Analyze** to begin testing the entire hierarchy of VIs.

Note: The tests should take roughly thirty seconds to complete on a fast computer if you run them on the entire application hierarchy.

2. Review the VI Analyzer Results and Correct Errors
    a. The dialog that appears after running the tests shows the list of VIs that was analyzed. The number shown in the right column indicates the number of items that require attention and review. Begin by expanding the items under 'Calculator.vi.' A total of 17 items should be listed.



b. The high importance test failures will be indicated using a red exclamation mark. As an example, expand the 'Spell Check' test and select 'Occurrence 1.' The description should explain that, 'The control "Timing" contains the misspelled word "Timing" in its Boolean text.'
c. Double-click on Occurrence 1. LabVIEW should highlight the button on the front panel with the misspelled word.
d. Select 'Sort by Test'
e. Select 'Wired Terminals in Subdiagrams > Display Overflow Protection.vi > Occurrence 1.'



f. Double-click 'Occurrence 1.' The controls 'Heart Rate Params,' are inside the case structure, which violates proper style-guidelines and potentially creates unnecessary copies in memory. Moving them outside of the case structure will correct this error.
g. Explore the remaining results and consult the description for details on how to correct the error.

3. Generate an HTML report
   a. Click 'Export' in the VI Analyzer Results Window.
   b. Change the location to the 'Analyzer Results' folder in the project and type in the name of the file you wish to save it as.
   c. From the drop-down, select **HTML**.
   d. Click **Export**
   e. Click **Done** on the VI Analyzer Results Window. Click **No** to dismiss the save dialog.

f. When prompted to return to VI Analyzer, click **No**

g. From within the Project Explorer, expand the 'Analyzer Results' folder

h. Double-click the new html document to see the results in a browser. Note that this dialog includes links to tests for navigation.

Failed Tests (sorted by VI)

Memory Options.ctl (C:\Users\ekerry.AMER\Desktop\Software Engineering Hands-On\Source Code Control\Basic Calculator\Calculator\SubVIs\Custom Controls\Memory Options.ctl)

Test	Failure Message
VI Documentation	This VI has no VI Description.
Dialog Controls	The control labeled "Memory Options" is not a dialog-style control. Because of this, its appearance will not be platform-specific.

Operation Buttons.ctl (C:\Users\ekerry.AMER\Desktop\Software Engineering Hands-On\Source Code Control\Basic Calculator\Calculator\SubVIs\Custom Controls\Operation Buttons.ctl)

Test	Failure Message
VI Documentation	This VI has no VI Description.
Dialog Controls	The control labeled "Operation Button" is not a dialog-style control. Because of this, its appearance will not be platform-specific.

Add Digit to Display.vi (C:\Users\ekerry.AMER\Desktop\Software Engineering Hands-On\Source Code Control\Basic Calculator\Calculator\SubVIs\Display VIs\Add Digit to Display.vi)

Test	Failure Message
Enabled Debugging	This VI has debugging enabled, which can reduce performance slightly. Consider disabling debugging in the VI Properties >> Execution dialog box for this VI.

ADVANCED DEBUGGING AND DYNAMIC CODE ANALYSIS

Identifying the source and fixing the cause of unexpected or undesirable behavior in software can be a tedious, time-consuming and expensive task for developers. Even code that is syntactically correct and functionally complete is often still contaminated with problems such as memory leaks or daemon tasks that can impact performance or lead to incorrect behavior. These oversights can be difficult to reproduce and even more difficult to locate, especially in large, complex applications.

With the NI LabVIEW Desktop Execution Trace Toolkit, we can trace the execution of LabVIEW VIs on a Windows target during run-time to detect and locate problems in code that could impact performance or cause unexpected behavior. This will be helpful when struggling to locate the source of difficult to find, or difficult to reproduce issues. The Desktop Execution Trace Toolkit provides a chronological view of system events, queue operations, reference leaks, memory allocation, un-handled errors, and the execution of subVIs. Users can also programmatically generate user-defined events from the block diagram of a LabVIEW application.

Dynamic code analysis refers to the ability to understand what software is doing 'under-the-hood' during execution. In other words, it provides details about events and the context in which they occur in order to give developers a bigger picture and more information that can help solve problems.

Dynamic code analysis has a number of different use-cases throughout the software development life-cycle, including:

- Detecting memory and reference leaks
- Isolating the source of a specific event or undesired behavior
- Screening applications for areas where performance can be improved
- Identifying the last call before an error
- Ensuring the execution of an application is the same on different targets

Problems such as memory leaks can have costly consequences for systems that are required to sustain operation for extended periods of time or for software that has been released to a customer. If software that needs debugging has been deployed and the LabVIEW development environment is not installed on the current machine, it may be beneficial to perform dynamic analysis of the code with the Desktop Execution Trace Toolkit over the network. For deployed systems, even if the development environment is available, it may be impractical or difficult to locally troubleshoot or profile the execution of a running system.

Debugging Unexpected Behavior

Goal

We want to profile the execution of a LabVIEW application we've developed to find the source of un-desirable behavior.

Scenario

Consider that you have software in use that appears to work fine, but it eventually begins to get slower and less responsive, or eventually quits unexpectedly. You suspect a memory leak, but the application is very large and it could take an extremely long time to track down the source of this problem.

Description

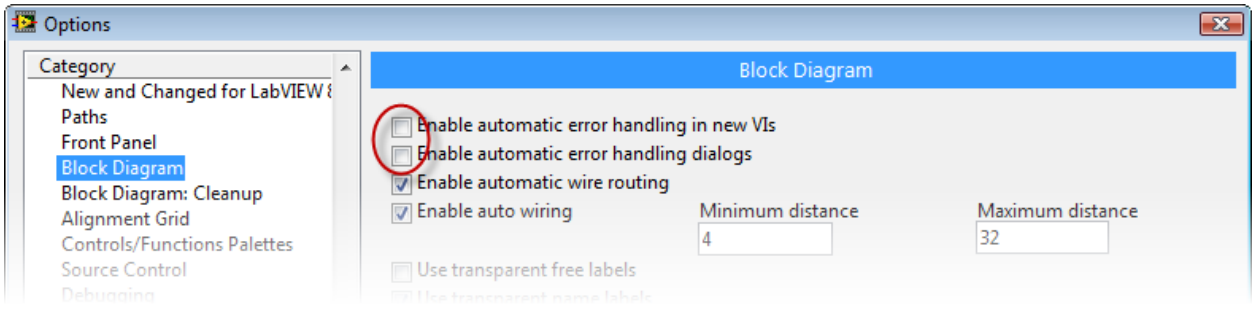
We're going to use the Desktop Execution Trace Toolkit to monitor the execution of our suspect application and see if we can find the source of these problematic behaviors.

Concepts Covered

- How to setup and configure a trace
- How to filter the information
- User-defined trace data
- Finding the source of an event
- Identifying memory leaks
- Discovering un-handled errors

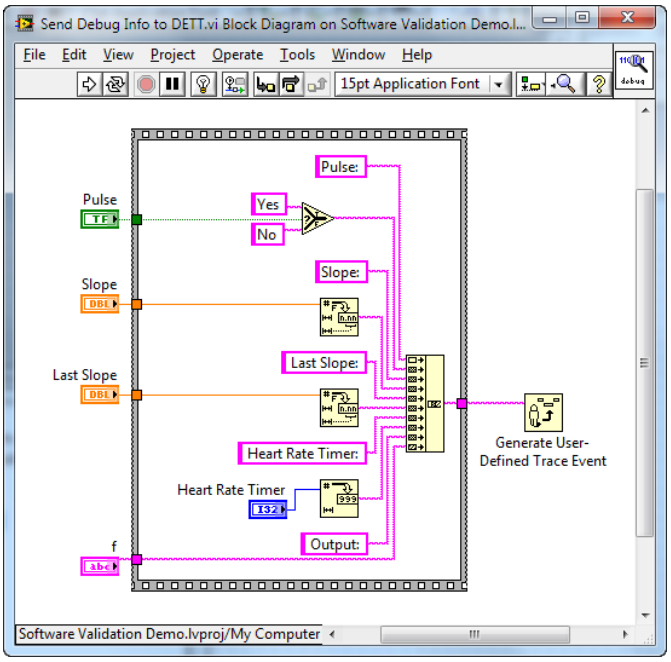
First Steps

- Make sure that LabVIEW and the Desktop Execution Trace Toolkit are installed.
- Make sure that a firewall is not preventing communication between the tool and LabVIEW
- In LabVIEW, go to Tools >> Options and select the 'Block Diagram' category. De-select **Enable Automatic Error Handling in new VIs** and de-select '**Enable Automatic Error handling dialogs**.



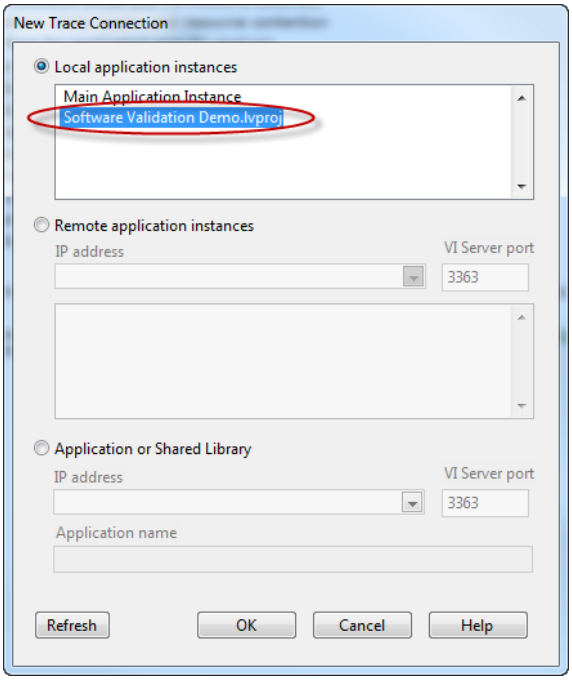
TRACE AN EXAMPLE IN THE DEVELOPMENT ENVIRONMENT

1. Demonstrate that application does not appear to have any obvious defects and show how custom trace data has been programmed into the application
    a. In the Project Explorer, open the VI entitled **Main.vi**. As has been demonstrated in prior steps, the application works correctly and does not have any obvious bugs; however, errors have been intentionally coded into this application for the sake of demonstration, including a memory leak that will degrade performance over time.
    b. In the Project Explorer, expand the folder 'Main Demo' and open the VI 'Send Debug Info to DETT.vi.' This VI uses the 'Generate User-Defined Trace Event' primitive to stream data to the Desktop Execution Trace, which is extremely helpful when trying to debug code that quickly iterates through a large amount of data. The desktop execution trace toolkit will allow us to see all of the values in each iteration.



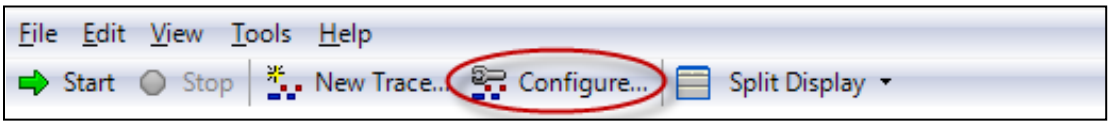
2. Setup the Trace
    a. Launch the Desktop Execution Trace Toolkit from the start menu by navigating to **National Instruments > LabVIEW Desktop Execution Trace Toolkit**
    b. Select 'New Trace' from the toolbar at the top. In the dialog that appears, we could chose to trace a deployed executable or shared library over a network, as well as a remote development environment. To get started, select **Local Application Instance.**

    c. In this menu, you should see the name of the current Project listed. Select it.

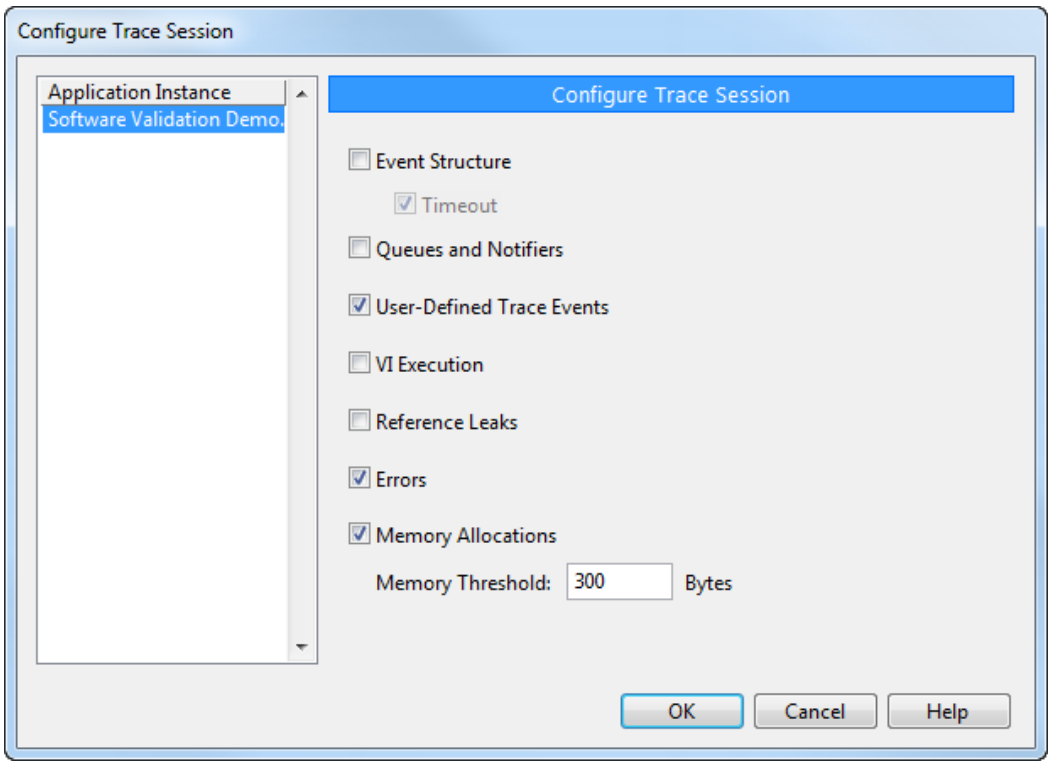


d. Click **OK**

3. Configure the Trace
    a. Select **Configure** from the toolbar at the top. We can capture a lot of information from this tool, and we'll see later on how to setup filters to help us parse the information, but for now we can actually configure what data we want to record

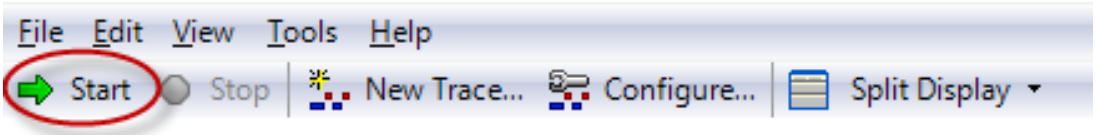


    b. Explain that we suspect a memory leak, so start by turning on **Memory Allocations**.
    c. By default, the threshold is set to 0 bytes. As a result, we're going to see a memory allocation for a lot of little things that the LabVIEW compiler does. Up this threshold to eliminate some of the noise to roughly **300 Bytes**
    d. Set the rest of the checkboxes as shown in the image below

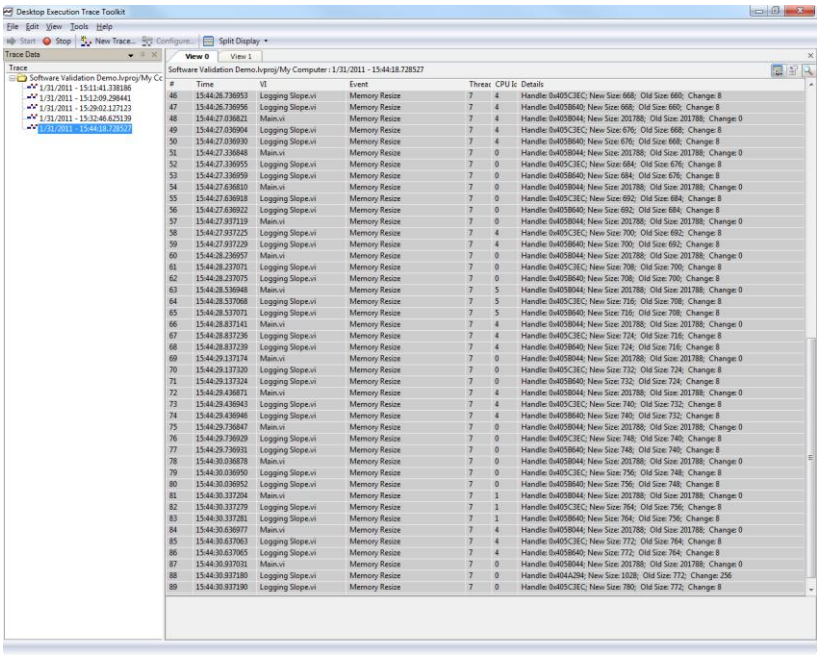


e. Click **OK**

4. Begin tracing execution
   a. Select **Start** from the toolbar

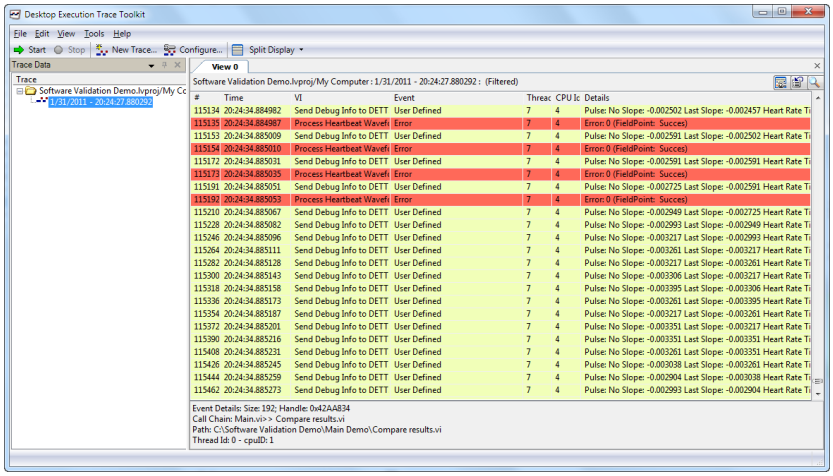


   b. Switch back to the front panel of **Main.vi**
   c. Position the front panel and the Desktop Execution Trace Toolkit on the screen so that both can be seen simultaneously
   d. **Run** the VI and when prompted, select the same sample data as last time
   e. Note that you should see data appear in the trace as shown below after selecting the dataset.



f. The screen will continue to populate with 'Memory Resize' as we have code in our application that has a leak in it.

g. Return to the application and click **Take Blood Pressure**. As a result, we should now see some user-generated data and an error message



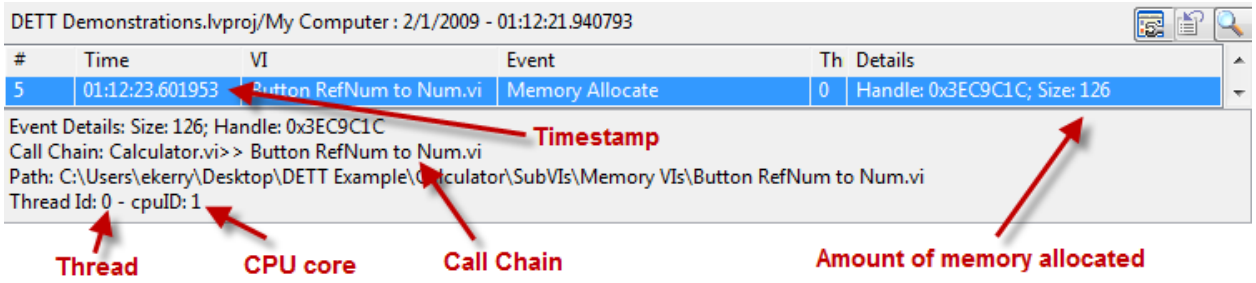
h. **Stop** the trace

5. Understanding the Data

a. Before we try to find our memory leak, we should stop and examine the data we're getting to see what it means.

b. We get several columns of data back, including the VI it occurred in, the description of the event, and other information such as the timestamp

c. Highlight an event by clicking on it, such as a memory allocation.



d. Highlight the items shown in the image above

6. Finding the un-handled error
   a. The trace data from the main.vi should have yielded an un-handled error, which will be highlighted in red. We had no inclination of this happening when our program ran, but the Desktop Execution Trace Toolkit has pointed out a potential problem that could be impacting the behavior of our code.
   b. Clicking on it will give you the details of the event.
   c. Double-clicking on the trace identifies the property node by bringing up the block diagram and highlighting it.
   d. The property node is throwing an error because you're trying to write the **SyncDisp** property, which cannot be set during run-time.

TESTING AND VALIDATION

The idea behind unit testing is elegant and simple, but can be expanded to enable sophisticated series of tests for code validation and regression testing. A unit test is strictly something that 'exercises' or runs the code under test. Many developers manually perform unit testing on a regular basis in the course of working on a segment of code. In other words, it can be as simple as, 'I know the code should perform this task when I supply this input; I'll try it and see what happens.' If it doesn't behave as expected, the developer would likely modify the code and repeat this iterative process until it works.

The problem with doing this manually is that it can easily overlook large ranges of values or different combinations of inputs and it offers no insight into how much of the code was actually executed during testing. Additionally, it does not help us with the important task of proving to someone else that it worked and that it worked correctly. The cost and time required is compounded by the reality that one round of testing is rarely enough; besides fixing bugs, any changes that are made to code later in the development process may require additional investment of time and resources to ensure it's working properly.

Large projects typically augment manual procedures with tools such as the NI LabVIEW Unit Test Framework Toolkit to automate and improve this process. Automation reduces the risk of undetected errors, saves costs by detecting problems early in the development lifecycle, and saves time by keeping developers focused on the task of writing the software, instead of performing the tests themselves.

UNIT TESTING AND VALIDATION OF CODE

GOAL

We want to automate the process of testing VIs in order to make sure they exhibit correct behavior.

SCENARIO

Consider that you've been given requirements for implementing a subroutine and you want to make sure it works as expected. Automating the tests makes it possible to re-run them on a regular basis and thereby mitigate the risk of making a change that could introduce a problem. We can also generate reports and get additional information about our code that can help further improve the quality and reliability of the application.

DESCRIPTION

We're going to use the NI LabVIEW Unit Test Framework Toolkit to generate a test case for a simple VI, examine the results, and generate a report.

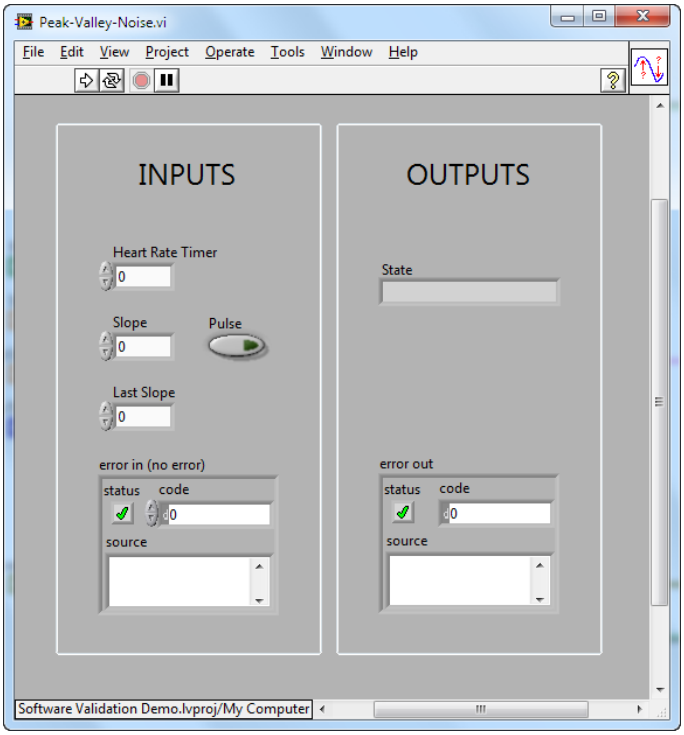
CONCEPTS COVERED

- Creating a unit test
- Defining test cases
- Tracking tests in the Project Explorer
- Importing test parameters from the front panel
- Executing tests
- Interpreting the test results dialog
- Report generation

FIRST STEPS

- Ensure that LabVIEW and the Unit Test Framework Toolkit are installed.
- Make sure the UTF Directory is setup properly
    - Right click on the project file in the Project Explorer and select properties.
    - Select 'Unit Test Framework'
    - Scroll down to 'Test Creation' and make sure that the correct directory is selected

1. Perform a Manual Test
    a. In the Project Explorer, expand **SubVIs > Main Demo** and open **Peak-Valley-Noise.vi**. This VI executes in a loop to identify the peaks and valleys based by comparing the instantaneous slope with the previous slope.
    b. Review the requirements document to get the test vector.
    c. **Peak-Valley-Noise.vi** has the following inputs:
        i. **Slope** – this is the rate at which the pressure is changing between the two most recent points
        ii. **Last Slope** – this is the rate at which the pressure is changing between the previous two points
        iii. **Heart Rate Timer**– this corresponds to the number of data-points that have been analyzed since the previous valley. If the number of samples is less than 300, it should return 'Noise'
        iv. **Pulse** – set to True after a Valley and until a Peak
        v. **error in (no error)** – if an error is passed into this VI, it should return the same error and zeros for both indicators.
    b. **Peak-Valley-Noise.vi** has the following outputs:
        i. **State** – the options are Peak, Valley or Noise
        ii. **error out** – if invalid inputs are received by this VI, it will output an error. It will also pass any errors that are input to it.

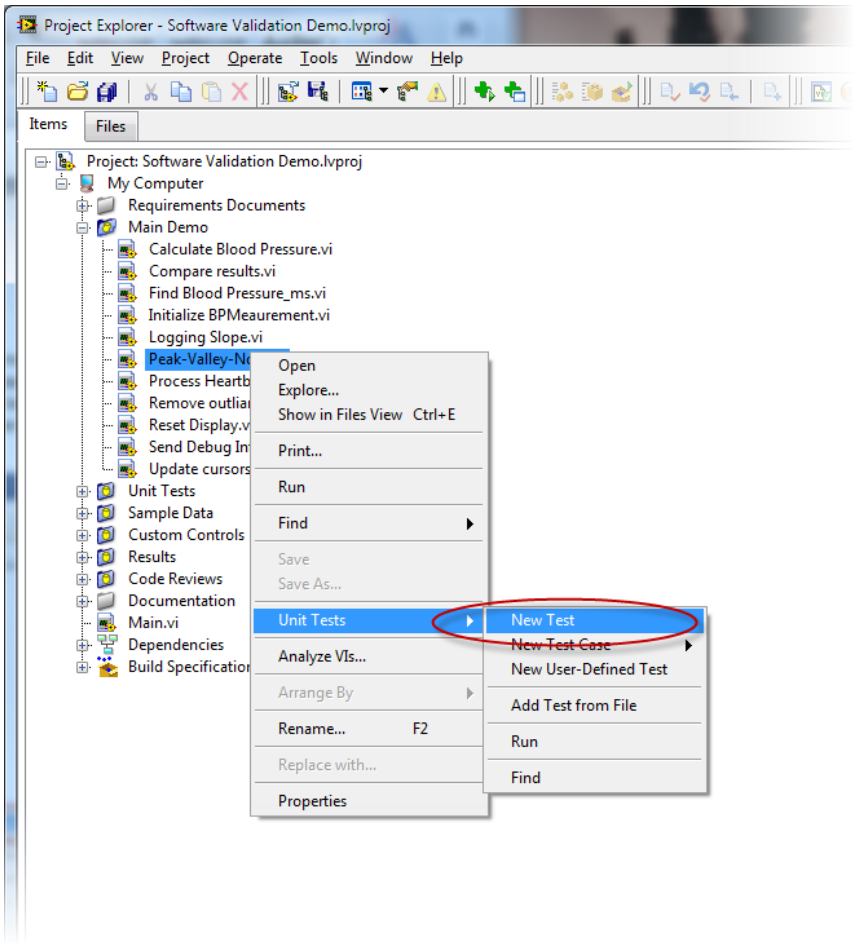


    d. Configure the VI for a manual test
        i. Set **Heart Rate Timer** to '500'
        ii. Set **Slope** to '4
        iii. Set **Last Slope** to '-9'
    b. Run the VI. The outputs should be:

i. **State** should be 'Valley'
ii. **Error Out** status should not indicate an error

2. Define a Unit Test for **Peak-Valley-Noise.vi**

e. Though we've manually run the VI to check and make sure it works, there are numerous conditions that we want to make sure this VI can properly handle. Several tests have already been created for this VI, but we're going to start by creating a new test for positive values. Right-click on the VI and select '**Unit Tests > New Test**.'



a. LabVIEW will generate a new file on disk with an .lvtest extension. The test will be created next to the VI under test by default, though we can specify a separate location or move the test on disk from within the 'Files' tab. Double-click on the unit test in the Project Explorer to open the Test Properties dialog.

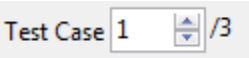
b. The first category shows the basic configuration of the unit test. The information displayed includes the following:

i. **VI Under Test** – this will automatically be configured, but we can change it at a later date if we move or rename a VI under test outside of the LabVIEW Project Explorer.
ii. **Test Priority** – this number can be used to group tests and test results based upon importance. As an example, you can tell the Unit Test Framework to only run tests that are at least a certain priority.

iii. **Requirements ID** – this ID can be read by NI Requirements Gateway for the sake of automated traceability to requirements documents.

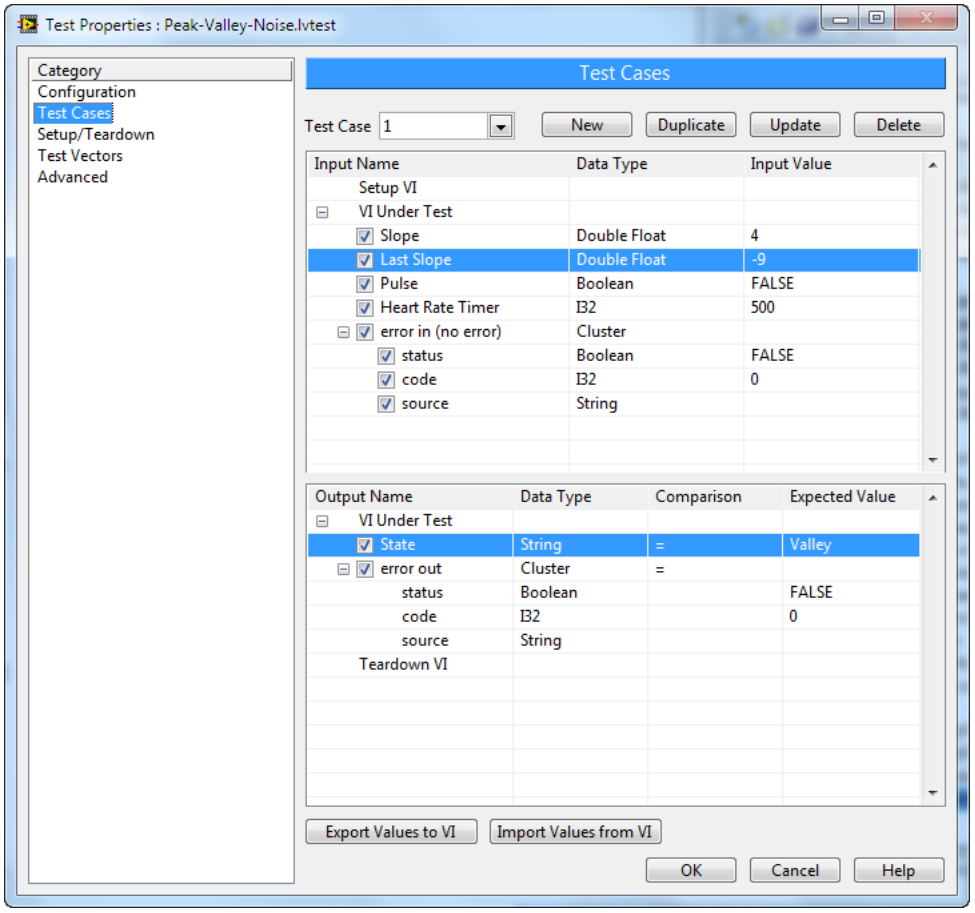
b. Enter the requirement ID: SwTestReq1 into the **RequirementsID** field

c. Select the 'Test Cases' category. Note that one unit test can contain multiple test cases.

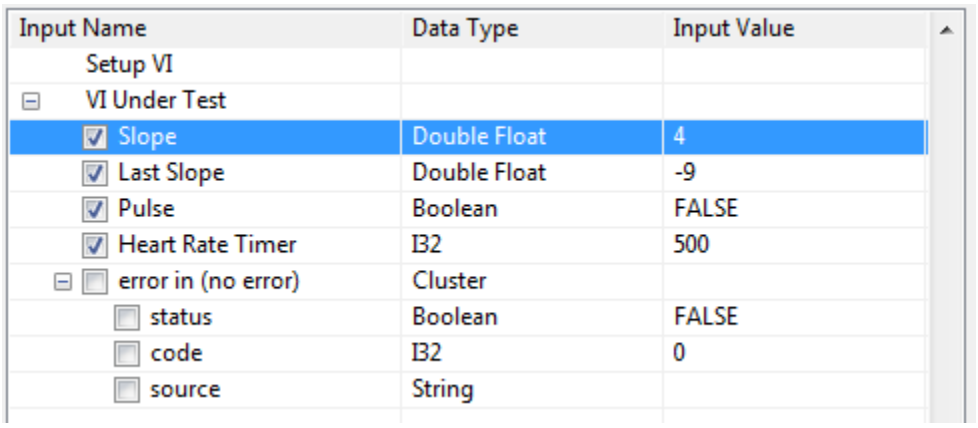


d. The right side of the 'Test Cases' dialog will display the inputs and outputs of the VI Under Test. From this dialog we can configure the following:
   i. The inputs to set
   ii. The input values
   iii. The excepted outputs
   iv. The outputs to compare
   v. The comparisons to be made between the actual results and the expected results

e. Only the controls and indicators connected to the connector pane will be shown in the Test Case dialog by default, but we can adjust the settings from the **Advanced** category to use any and all controls and indicators. Tests can be created for any data-type in LabVIEW, including arrays and clusters. For complex datasets, it may be better to define values in the .lvtest file, via the front panel or a setup VI.



f. Define the input values for the values as shown below:



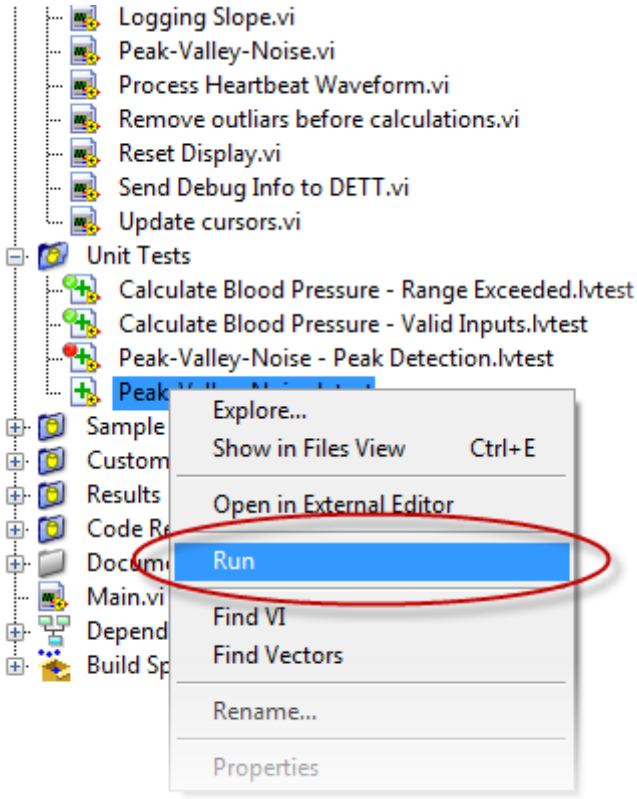
Input Name	Data Type	Input Value
Setup VI		
VI Under Test		
Slope	Double Float	4
Last Slope	Double Float	-9
Pulse	Boolean	FALSE
Heart Rate Timer	I32	500
error in (no error)	Cluster	
status	Boolean	FALSE
code	I32	0
source	String	

g. Define the output string as 'Valley'

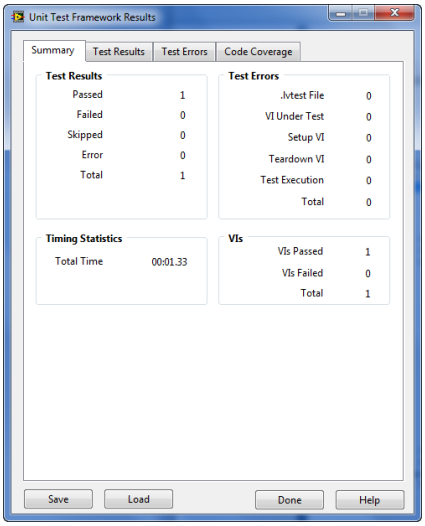
h. Feel free to define additional test cases from this dialog by clicking **New** at the top.

i. Click **OK**

j. Right click on the test in the Project Explorer and select **Run**

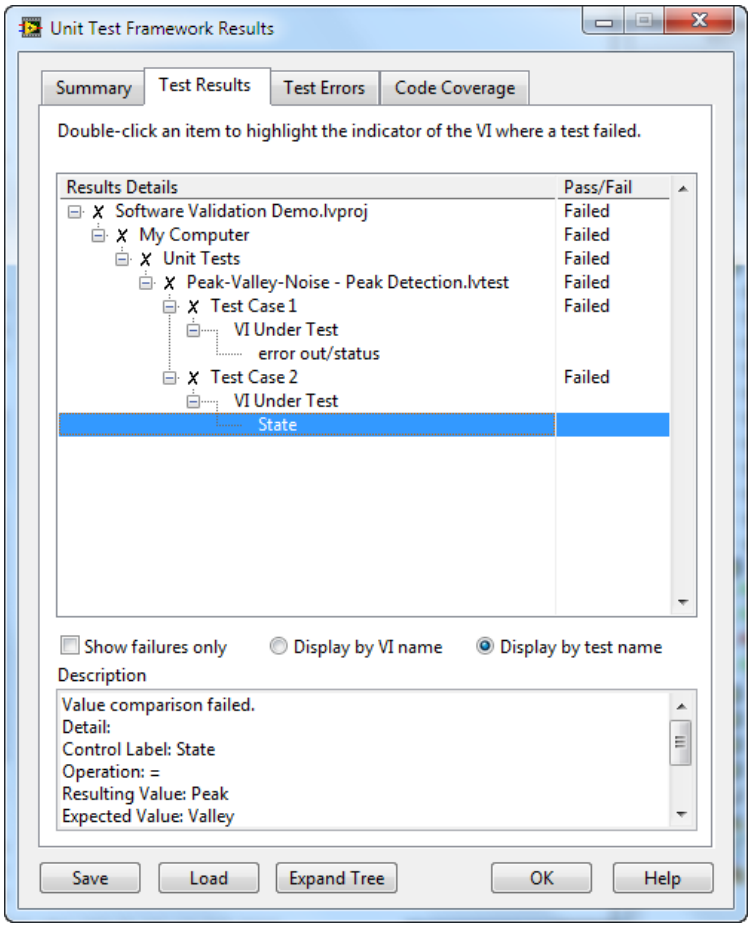


k. The test should pass, which will be indicated by a green icon that is overlaid on the test in the Project tree. A dialog will also appear explaining the results.

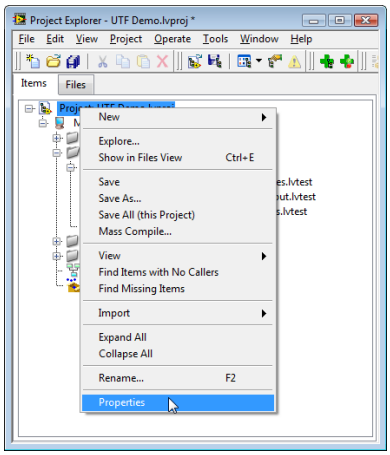


l. Click **Done**

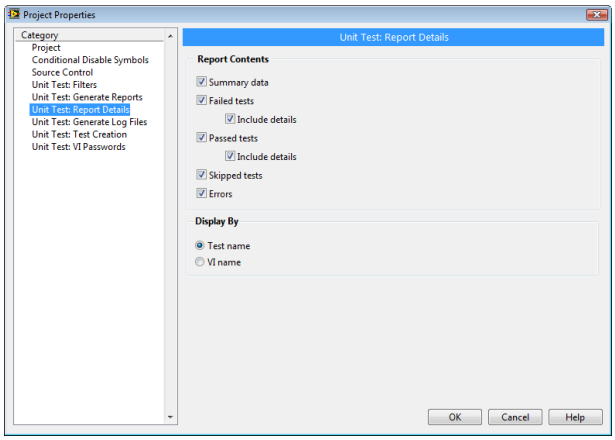
3. Running multiple tests and troubleshooting test failures
    a. Expand the folder titled 'Unit Tests' in the Project tree to see the pre-defined tests for Vis in this Project.
    b. Right-click on the folder entitled **Unit Tests** and select **Unit Tests > Run**. In addition to running all the tests in a folder, we could also run all the tests for just a particular VI, all of the tests in the Project Explorer, or we could run tests programmatically using the documented API and VI palette.
    c. When the test is complete, notice that the icon in the Project Explorer for **Add Digit to Display – Negative Values.lvtest** now has a red dot next to it, which indicates that the test failed. Additionally, the Test Results dialog has appeared which indicates that one of the tests has failed.
    d. Click on the **Test Results** tab to find out what went wrong.



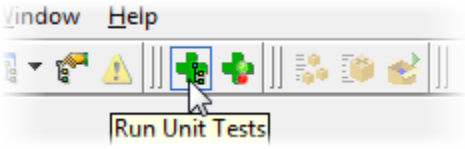
4. View the traceability in Requirements Gateway
    a. Launch the completed NI Requirements Gateway project
    b. Switch to the traceability view
    c. Show that the requirement for Peak-Valley-Noise is now covered by this unit test that was just implemented
5. Turn on Report Generation
    a. In the Project Explorer, right click on the project file and select properties



    b. The properties dialog contains various settings and preferences for the Unit Test Framework, including test filters and the default location for new tests
    c. Select **Unit Test: Report Details** and check everything



    d. Select **Unit Test: Generate Reports** and select **Generate HTML Report** and **View Report after Execution**
    e. Click on the icon in the toolbar to **Run Unit Tests**



    a. After the tests are complete, the HTML report should display in your browser.

More Information

Online Resources

- **ni.com/largeapps** – find best practices, online examples and a community of LabVIEW users
- **ni.com/softwareengineering** – download evaluation software and read more about the tools in this guide

Customer Education Classes

- Managing Software Engineering with LabVIEW
    - Learn to manage the development of a LabVIEW project from definition to deployment
    - Select and use appropriate tools and techniques to manage the development of a LabVIEW application
    - Recommended preparation for Certified LabVIEW Architect exam