Extending SilverMark's Test Mentor for VisualWorks Smalltalk to Record and Play Back User Interactions, and Validate State in Custom Widgets

Contents:

Introduction

There are four facets to automating user interface-based testing that SilverMark's Test Mentor provides:

- **Recording** user interface interactions
- **Playing back** user interface interactions
- **Capturing** widget state
- **Verifying** widget state

The processes of recording user interface interactions and capturing widget state both result in automatically adding test steps to a scenario within the Test Mentor Test Editor.

SilverMark's Test Mentor performs this with a minimum of intrusiveness to your code. In fact, **Test Mentor makes no permanent changes to any methods, whatsoever**.

Test Mentor is designed to be able to perform the above with all standard VisualWorks widgets as they are delivered by Cincom. SilverMark realizes that many people make modifications to the base widgets and often create their own widgets, or use 3rd party widgets. To accommodate this, Test Mentor was designed to be able to be extended to support changes to base widgets, as well as custom widgets.

This document describes how to extend Test Mentor to support custom widgets. As an example, we will show how to record a VisualWorks tree widget (HierarchicalSequenceView) *expand* and *collapse* node interactions. This widget is used in the VisualWorks 5i.4 System browser:

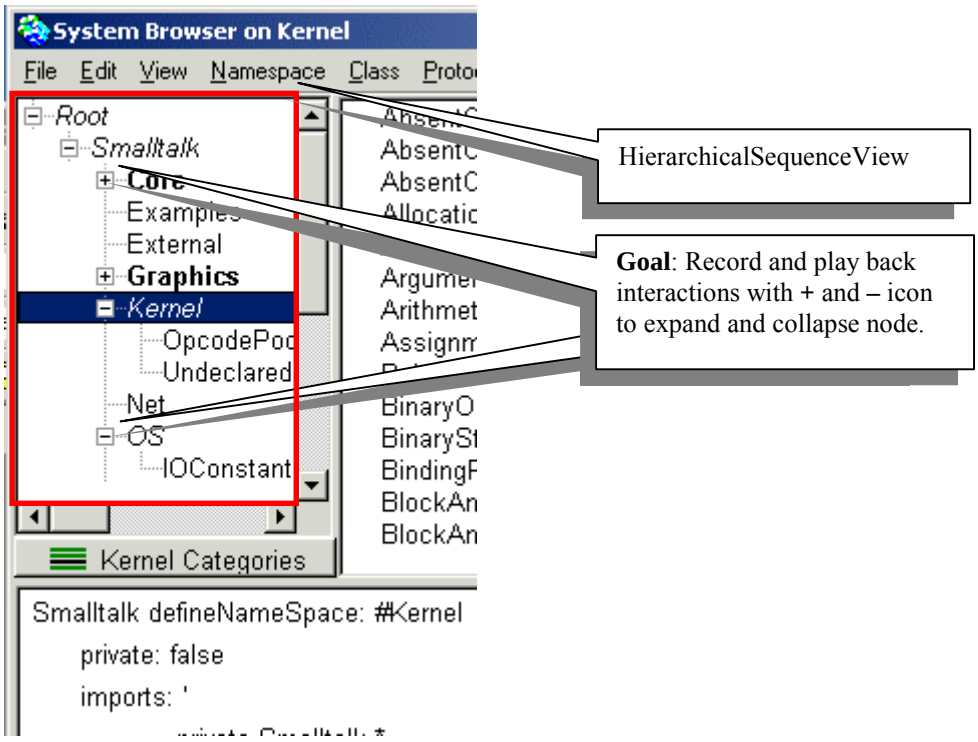

Figure 1 - View that uses example widget

Widget Extension Process overview

The process to extend a widget so that Test Mentor can record and play back interactions with it, as well as extract and verify state are fairly straight-forward. Most of the work is clerical in nature.

1. Identify or create a playback method. This method will programmatically cause the recorded interaction to occur during text execution. For example, if we want to record a user interaction that expands a node in a tree list, a playback method for this interaction will programmatically expand that same node within the tree list.
2. Register the playback method as one to be used to when generating a playback step for the recorded interaction.
3. Find the best place to add a recording hook. This is usually a method in a Controller class that has all the information required for recording an interaction, such as the index within a list.
4. Add the recording hook. Create a version of the Controller method identified above that calls one of Test Mentor's playback step generation methods.
5. Add widget verification

You will be creating methods in existing widget and Test Mentor classes, that will be saved and loaded by way of your own widget extensions parcel. That way the widget extensions code is kept separate from both the VisualWorks system and Test Mentor.

Estimating time to extend widgets

A developer who has experience with the implementation of a widget to be recorded, and has a reasonable familiarity with this process should expect to spend approximately 30 minutes setting up for recording and playback each interaction.

A widget that does not already have methods that can be used to programmatically play back the interaction may take s little longer. Likewise, if the developer is not at all familiar with the implementation of the widget, it will take a little longer for them to find the best place in the Controller to add a recording hook.

Adding User Interface Playback Extensions

Each recorded widget interaction must have a corresponding method that will recreate that interaction programmatically. So rather than creating playback statements that move the mouse pointer to a coordinate and then click or type a character, you will create playback statements that are semantically meaningful. For example, it makes much more sense to send #expandAtNode: to a hierarchy list widget rather than #clickAtPoint:

For many interactions the widgets already have these methods built in. In some cases you may need to create your own.

Example playback extension

As you may recall from above, we will be adding *expand* and *contract* interaction recording to HierarchicalSequenceView. After hunting around HierarchicalSequenceView we could not find APIs to programmatically perform explicit *expand* and *contract* operations on a tree node, however there is an API for toggling the state of a node.

We could use the *toggle* operation because that is what is actually called during recording. The toggle operation expands if a node is contracted and contracts if the node is expanded. This seems reasonable enough on the surface and saves us some work. Unfortunately the disadvantage to using toggle is that tests would be sensitive to the state of the nodes at the time of playback. That is, if we recorded an expand and played back using a toggle when the node was already expanded, it would contract, which would have an effect opposite to our intention.

This is the kind of thing that happens if you are debugging a test and run it several times on the same widget without ensuring a consistent initial state. It is best to make your playback APIs insensitive when possible. So in this case it is better to record an *expand* as an *expand* and play back it that way as well. That way you are guaranteed the outcome will be an expand and not a contract.

After some experimentation putting in halts and walking through the widget code with a debugger we determined that the following methods will enable us to programmatically expand an contract given nodes.

Playback Methods added HierarchicalSequenceView

```
stmContractNodeAtIndex: index
        "Contract the tree node at the given index. Do nothing if already closed"

        self indentedSelectionInList stmContractNodeAtIndex: index.
```

```
stmExpandNodeAtIndex: index
        "Expand the tree node at the given index. Do nothing if already open"

        self indentedSelectionInList stmExpandNodeAtIndex: index.
```

```
stmIsExpandedNodeAtIndex: aNumber
        "Return whether the node at the given index is opened or closed"

        ^self indentedSelectionInList stmIsExpandedNodeAtIndex: aNumber
```

Supporting methods added to IndentedTreeSelectionInList

```
stmExpandNodeAtIndex: aNumber
        "Expand the tree node at the given index. Do nothing if already open"

        self selectionIndexHolder value = nil ifTrue: [^self].
        self selectionIndexHolder setValue: aNumber.
        aNumber = 0 ifFalse: [| tmp |
                tmp := self
                findChildForIndex: self selectionIndexHolder value
                currentIndex: 0
                child: self root.
                (self isOpen: tmp) ifFalse:[self open: tmp] ].
```

```
stmContractNodeAtIndex: aNumber
        "Contract the tree node at the given index. Do nothing if already closed"

        self selectionIndexHolder value = nil ifTrue: [^self].
        self selectionIndexHolder setValue: aNumber.
        aNumber = 0 ifFalse: [| tmp |
                tmp := self
                findChildForIndex: self selectionIndexHolder value
                currentIndex: 0
                child: self root.
                (self isOpen: tmp) ifTrue:[self close: tmp] ].
```

```
stmIsExpandedNodeAtIndex: aNumber
        "Return whether the node at the given index is opened or closed"

        | element |
        aNumber = 0 ifTrue: [ ^self ].
        element := self
                findChildForIndex: aNumber
                currentIndex: 0
                child: self root.
                ^(self isOpen: element )
```

Specifying playback step generation

At this point we've created playback API methods. The next step is to create an internal API that will generate a Test Mentor step that is configured with a Smalltalk statement that calls the playback method.

The most common case of a user interface interaction playback step is one that contains the following code:

```
(ActiveWindow widgetNamed: <widgetID>) <playbackMethod> <optional arguments>
```

For example:

```
(ActiveWindow widgetNamed: #launchCustomerQuery) stmClick
or
(ActiveWindow widgetNamed: #customerList) stmSelectString: 'Wilkins McCawber'
```

We would like to generate steps configured with something like the following:

```
(ActiveWindow widgetNamed: #hierarchyList) stmExpandNodeAtIndex: 5
```

Mechanics of playback step generation

When Test Mentor begins recording user interface interactions it replaces the active instance of InputState with its own, called StmRecordInputState. This class defines methods in the **events** protocol for recording when specific events have occurred. Here's a few, for example:

Interaction	StmRecordInputState method	Generated playback method
a button has been pressed	#stmPressEventFrom:	#stmClick
An accept interaction has occurred	#stmAcceptEventFrom:value:	#stmEnterText:
A window close interaction has occurred	#stmCloseEventFrom	#close
A double-click interaction has occurred in a list at the given index	#stmDoubleClickEventFrom	#stmDoubleClick
An extended selection interaction has occurred in a given list	#stmExtendSelectEventFrom:	#stmExtendSelectItem:
A window has focus set to it	#stmFocusEventFrom:	#setFocusToWindowNamed:id:
record a keyboard event	#stmKeyboardHookFrom:key:	#stmKeyboardHookWith:
A specific menu item has been selected	#stmMenuItemSelectedEventFrom:value:	#stmPerformMenuAction:
Etc.	…	…

These methods account for 99% of the events resulting from user interface interactions that you might want to record. In general, it is best to simply implement a common playback API in your own widget and then reuse the standard playback step generation, rather than invent your own.

For example, if you were to invent an entirely new type of push button, it would be wise to simply implement #stmClick as a playback API. Then you could reuse #stmPressEventFrom: in StmRecordInputState to generate a playback step, rather than inventing your own method.

Some times widgets present novel ways to interact. If you have modified or created new widgets that define new types of interactions, you may need to add methods to StmRecordInputState to represent them and generate the right playback code. If this is the case, just do the following:

1. Implement a method in StmRecordInputState on the instance side that takes as arguments any information that needs to be passed to the playback method, and the controller for the widget.
2. Add code to the method that generates a step configured for the correct playback statement.

This is a lot easier than it sounds ☺

To illustrate the concept, consider the built-in method used to record *accept* events in text widgets in StmRecordInputState:

```
StmRecordInputState>>#stmAcceptEventFrom: aTextEditorController value: aString
	"Indicate an 'accept' event in a text widget as an interaction to be recorded"

	| view |
	view := aTextEditorController view.
	self recordEvent: (StmGuiRecording80SetUpHelper acceptIdentifierFor: aString) in: view.
```

This method is the starting point for generating a step that plays back entering a String into a text field. Most of this is 'boiler plate' code except for the call to #acceptIdentifierFor:, which calls the following method:

```
StmGuiEventsRecording80SetUpHelper>>#acceptIdentifierFor: aString
	"Return an action identifier configured for text entry"

	^StmGuiActionIdentifier forSelector: self enterTextSelector value: aString
```

Most of this method is 'boiler plate' code as well. It returns an instance of StmGuiActionIdentifier configured for a particular playback selector, which is retrieved from #enterTextSelector, which contains the following:

```
StmGuiEventsRecording80SetUpHelper>>#enterTextSelector
	^#stmEnterText:
```

These are all the pieces you need to specify playback code to be generated for a widget. If you implement methods like these, the rest of the recording framework will take care of creating a step and setting up the code for it with the proper widget identifier.

Example playback step generation

Let's try the above with our hierarchy list *expand* and *contract* interactions:

```
StmRecordInputState>>#stmExpandHierarchyNodeFrom: aController index: aNumber
	"Record 'expand hierarchy node' in hierarchy list, for a given node index"
	| view |
	view := aController view.
	self recordEvent: (StmGuiRecording80SetUpHelper expandNodeAtIndexIdentifierForIndex: aNumber) in: view.
```

```
StmGuiEventsRecording80SetUpHelper>>#expandNodeAtIndexIdentifierForIndex: anInteger
	"Return an action identifier configured for playing back an expand interaction"
	^StmGuiActionIdentifier forSelector: self expandNodeAtIndexSelector value: anInteger
```

```
StmGuiEventsRecording80SetUpHelper>>expandNodeAtIndexSelector
	^#stmExpandNodeAtIndex:
```

```
StmRecordInputState>>#stmContractHierarchyNodeFrom: aController index: aNumber
	"Record 'contract hierarchy node' in hierarchy list, for a given node index"
	| view |
	view := aController view.
	self recordEvent: (StmGuiRecording80SetUpHelper contractNodeAtIndexIdentifierForIndex: aNumber) in: view.
```

```
StmGuiRecording80SetUpHelper>>#contractNodeAtIndexIdentifierForIndex: anInteger
	"Return an action identifier configured for playing back a contract interaction"
	^StmGuiActionIdentifier forSelector: self contractNodeAtIndexSelector value: anInteger
```

```
StmGuiRecording80SetUpHelper>>#contractNodeAtIndexSelector
        ^#stmContractNodeAtIndex:
```

Notice that #expandNodeAtIndexSelector and #expandNodeAtIndexSelector return the names of the expand API methods we added to the widget earlier.

Note: Don't forget that you would add these methods to your own widget extensions parcel.

Adding User Interface Interaction Recording Extensions

The last piece to the puzzle is finding the right place to call one of the internal playback step generation APIs in StmRecordInputState discussed above. The way to do this is to insert calls to those step generation methods within various widget or controller classes.

Code inserted into a widget or controller method that calls a playback step generation method is called a *recording hook*. Typically methods such as **controlActivity** and **redButtonActivity** are likely candidates for containing recording hooks.

Test Mentor, however, does not actually modify system methods, nor does it affect the way in which the system works while it is not recording. The way it modifies methods is through ***temporary*** bytecode replacement. This sounds scary but its not. When Test Mentor begins recording interactions from the user interface, it swaps the bytecodes of the required controller methods with altered versions that contain recording hooks. When recording ends, Test Mentor swaps them back.

Adding Your Own "Recording Hooks"

Test Mentor uses a simple mechanism for recording interactions.

The key to adding a recording hook is to find a point in some method, usually in a widget's controller, in which all the information pertinent to the user interaction event is available. Then insert code in the identified method to send one of the above messages (or a new one of your own creation) to the current active input state (StmRecordInputState). The preceding statement is almost true. In reality, *you will not alter existing methods*. You will create a copy of an existing method under a different name, that contains your addition of a recording hook in it, and do so in such a way that Test Mentor automatically uses the copy of the method during recording.

The technique of creating a copy of a method to contain changes to the method in such a way as to enable Test Mentor to replace it during recording is called *Method Replacement*.

Method Replacement Mechanics

This section describes how to easily set up your replacement methods that contain recording hooks.

Step 1: Create a subclass of SilverMark.StmSetUpHelper to as a place to put your method registration methods.

Make sure you add this, as well as any subsequently created methods to a separate parcel for extensions to your widgets.

Step 2: Pick a prefix for the methods you modify in order to identify them to Test Mentor. We recommend that you use a three-letter abbreviation for the name of your company, product or project and add the word "mod" to it. The prefix "stmmod" for **S**ilverMark's **T**est **M**entor is the default. In this case, when you write replacement methods, they should all be prefixed such as in #**stmmodredButtonActivity** to replace #**redButtonActivity**.

If you don't like the default prefix, simply override SilverMark.StmSetUpHelper class>>#replacementMethodPrefix to return your own prefix String. You can return anything you like as long as it satisfies the rules for being part of a valid method selector.

Step 3: Register methods to be replaced during recording in a class method called #replacementSpecCollection.

This method returns an array of specifications for method replacements. Each element in the array is a three-element array that defines a specific method replacement.

Element	Description
1	The name of the class that contains the method
2	A Boolean indicating whether the method is a class method (true) or an instance method (false)
3	The method selector

To get an idea of what we're talking about, look in SilverMark. StmGuiEventsRecording80SetUpHelper class>>#replacementSpecCollection. You will see some of the methods that are replaced at the beginning of recording for standard VisualWorks widgets:

```
replacementSpecCollection

        ^#(#(#RedButtonReleasedEvent false #dispatchTo:)
         #(#ComboBoxButtonController false #openDropDownListWithEvent:)
         #(#SequenceController false #doubleClickEvent:)
         #(#RedButtonPressedEvent false #dispatchTo:)
         #(#EmulatedSequenceController false #toggleAt:withEvent:)
         #(#SequenceSelectionTracker false #exitDueToDragDrop))
```

Now implement a class method called #initialize that has the following in it:

```
initialize

        StmReplacementRegistry recordInstance
                registerReplacementSpecs: self replacementSpecs
                for: self name
                priority: StmReplacementRegistry baseMethodPriority.
```

This will register your replacement methods to Test Mentor as needing to be replaced during recording.

Step 4: Add code to the post-load actions of your widget extensions parcel to call this method. For test and debugging purposes you can execute this method from a workspace.

Note: When recording ends, the original methods are restored automatically.

Example method replacement

As you may recall, we've chosen to illustrate the process by extending the VisualWorks tree widget (HierarchicalSequenceView) *expand* and *collapse* node interactions.

Step 1: Not being familiar with this widget ourselves, we spent a few minutes exploring the code until we discovered that this widget uses HierarchicalSequenceController. After a few minutes exploring that class and we discovered that HierarchicalSequenceController>>#selectionNeedsToggle: is the best place to hook in to expanding and collapsing nodes because the method embodies both the information that the + or – button was pressed, as well as the item index location:

```
selectionNeedsToggle: event
```

```
| mousePoint elem |
mousePoint := self sensor mousePointFor: event.
elem := self findElementFor: mousePoint.

^(view isMouseOverToggleButton: elem point: mousePoint)
        ifTrue: [self view toggleListAt: elem. true] ifFalse:[false].
```

We know that a + or – was pressed here and the index of the list item. Looks like a good place to add a recording hook

Step 2: Next we insert a call to the internal playback step generation API described above in StmRecordInputState, to record this interaction, like this:

```
stmmodselectionNeedsToggle: event

        | mousePoint elem |
        mousePoint := self sensor mousePointFor: event.
        elem := self findElementFor: mousePoint.

        ^(view isMouseOverToggleButton: elem point: mousePoint)
                ifTrue: [
                        "Test Mentor recording hook"
                        self stmToggleListAt: elem.
                        self view toggleListAt: elem. true] ifFalse:[false].
```

Call a method that will record this interaction

Notice that we prefixed the replacement method with a "*stmmod"*. Now let's implement the code that records the interaction:

```
stmToggleListAt: aNumber
        "Request to record toggling state of node (by clicking on '+' or '-' in hierarchical sequenc
          Record it as expand or contract request, depending on current state"

        (view stmIsExpandedNodeAtIndex: aNumber)
                ifTrue: [ InputState default stmContractHierarchyNodeFrom: self index: aNumber ]
                ifFalse: [ InputState default stmExpandHierarchyNodeFrom: self index: aNumber ].
```

Method to determine current state of the node (implemented

Notice that it first checks to see if the node is already expanded. If it is, then it records a *contract* interaction. Otherwise it records an *expand* interaction.

We chose to put this code in a separate #stmToggleListAt: method instead of directly in #stmmodselectionNeedsToggle in order to make the code more resilient to future changes in the widget. This way, if #selectionNeedsTarget: ever changes and we need to move the recording hook, we only need to move one line of code.

Step 3: Create MyWidgetExtensions.SetUpHelper as a subclass of SilverMark.StmSetUpHelper. This class will house methods used to register the replacement methods.

Return a list of replacement methods in #replacementSpecCollection:

```
replacementSpecCollection

        ^#( #(#HierarchicalSequenceController false #selectionNeedsToggle:))
```

Add initialization:

```
initialize

        StmReplacementRegistry recordInstance
                registerReplacementSpecs: self replacementSpecs
                for: self name
                priority: StmReplacementRegistry baseMethodPriority.
```

For testing and debugging you can execute this method manually from a workspace. Ultimately you should add a call to it to the post-load actions for the parcel that loads it.

That's it! If you run the #initialize method above your replacement methods will be added to those that are replaced during recording. You can test it by recording expanding and collapsing nodes in the system browser described above, or any other view that uses the HierarchicalSequenceView widget. After recording you should UI recording steps containing code that calls the playback API methods you added.

Adding Widget State Verification Extensions

Test Mentor provides an automated mechanism for capturing and later comparing widget state. The idea is to capture the value of an attribute of a widget, and generate a Test Mentor verification step that when executed, extracts the same attribute and compares it to the captured value.

Specifically, during test execution when the step is eventually executed, the widget attribute value is extracted and compared against the captured reference value. This section describes how to extend Test Mentor to provide widget state verifications for custom widgets.

Test Mentor provides a mechanism for displaying the state of a widget and selecting attributes to capture for later verification. When you press the verify button in the Test Editor  the system enters a mode where it waits for you to click on a widget. When you do, the following window opens, displaying a list of attributes for the widget and their respective states (values) that you can capture for later verification:

Figure 2 - Verification wizard

When the user selects an attribute from the list above, The Test Editor adds a test step to the currently selected scenario that contains a selector to be used to extract the value of the widget's respective attribute, as well as the current attribute's value as reference.

If you have modified existing widgets or created your own widgets, you can easily enable them for widget verification.

Mechanics of enabling widgets for verification

A widget that is enabled for verification has the following properties:

- It has a state validation 'getter' method for each state verification attribute of interest
- It will display the verification wizard when clicked on during verification mode
- It displays verification attributes in the verification wizard

State validation 'getter' methods

State validation Getter methods are simply public methods that return some state value. If a widget does not provide a getter method for the attribute you want to verify, simply add one.

Example getter method

Step 1: For our example HierarchicalSequenceView we need a method that returns whether the currently selected not is expanded or contracted. To do this we implemented the following:

```
HierarchicalSequenceView>>#stmIsExpandedNodeAtSelection
        "Return whether the currently selected node is expanded"

        ^self indentedSelectionInList stmIsExpandedNodeAtSelection
```

```
IndentedTreeSelectionInList>>#stmIsExpandedNodeAtSelection
        "Return whether the currently selected node is expanded"

        | index |

        index := self selectionIndexHolder value.
        index = nil ifTrue: [^nil].
        ^self stmIsExpandedNodeAtIndex: index.
```

Step 2: We need to ensure that the state extraction 'getter' method HierarchicalSequenceView>>#stmIsExpandedNodeAtSelection is added to the list of attributes of the widget to appear in the verification wizard's list. To do that, simply implement #stmWidgetAttributesInto: in your widget class that adds an instance of StmVisualAttribute configured for the verification 'getter' method, to the passed collection:

```
HierarchicalSequenceView >>#stmWidgetAttributesInto: aCollection

        super stmWidgetAttributesInto: aCollection.

        aCollection add: (self stmNewAttributeName: 'Is Selection expanded'
                                valueSelector: #stmIsExpandedNodeAtSelection)
```

This method adds an instance of StmVisualAttribute configured to use the method #stmIsExpandedfNodeAtSelection with description, 'Is selection expanded', to a collection of attributes to be displayed.

Overriding the 'red button pressed' event

The last piece to the verification puzzle is overriding the red button press event processing to cause the verification wizard to open when a widget is clicked on instead of doing what it normally does. To do this we need to return to our technique of method replacement.

To override red button press event processing, simply create a copy of #redButtonPressedEvent: that calls #verifyWidget: in StmVerifyInputState, instead of the usual event processing.

Note: Test Mentor temporarily replaces the instance of InputState with an instance of StmVerifyInputState during verification (after pressing ):

```
stmvfyredButtonPressedEvent: event
        InputState default verifyWidget: self view.
```

To register this method use the technique you used earlier to register replacement methods. The only difference is that you will be registering this method as a *verification* replacement method rather than a *recording* replacement method.

Example registering overridden 'red button pressed' event method

In our example you would create a #verificationSpecCollection method in MyWidgetExtensions.SetUpHelper that looks like this:

```
verificationSpecCollection

        ^#(#HierarchicalSequenceController false #redButtonPressedEvent:))
```

You would then add code to #initialize to MyWidgetExtensions.SetUpHelper in order to add the verification specs:

```
initialize

        StmReplacementRegistry recordInstance
                registerReplacementSpecs: self replacementSpecs
                for: self name
                priority: StmReplacementRegistry baseMethodPriority.

        StmReplacementRegistry verificationInstance
                registerReplacementSpecs: self verificationSpecs
                for: self name
                priority: StmReplacementRegistry baseMethodPriority.
```

Again, you should make sure a call to #initialize is added to the parcel as a post-load action. For debugging and testing you can just execute #initialize from a workspace.

One you have these pieces in place, you can test it out by selecting any scenario and pressing the verification  button. If you pop up on a HierarchicalSequenceView widget you should see something like the following:

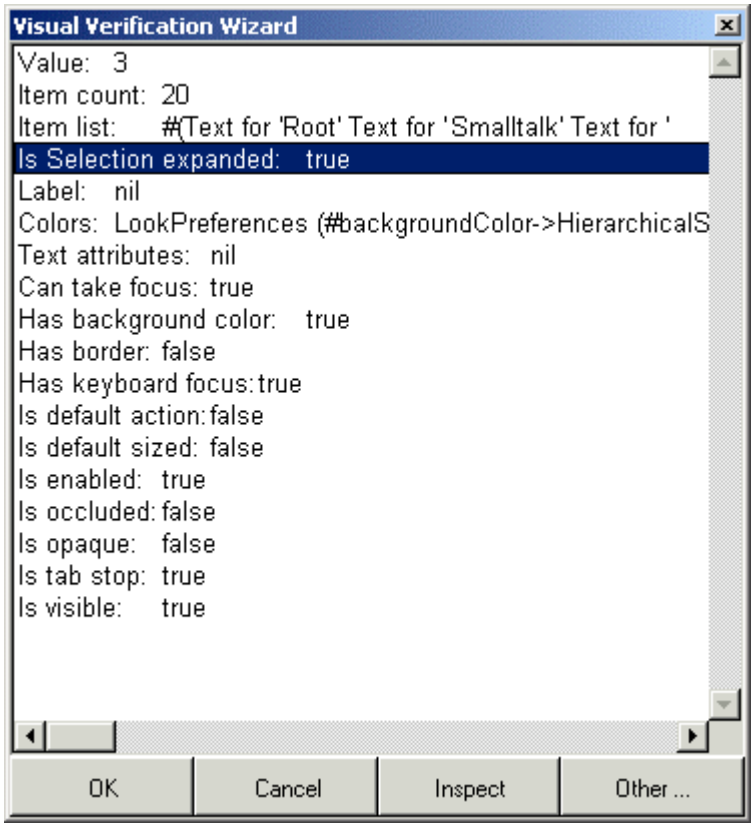

Figure 3 - Verification wizard for HierarchicalSequenceView in 5i.4 System browser

Notice that the attribute named 'Is Selection expanded' is shown.

Conclusion

Test Mentor provides an easily extendable framework for recording and playing back interactions with and verifying the state of any VisualWorks widget, whether it is a standard VisualWorks widget, modified VisualWorks widget or a completely new or 3rd party widget.

Extending widgets to make them work with Test Mentor requires neither imagination, nor a lot of skill. Once you do it a few times it becomes mostly a matter of book keeping.

For more information and advise about recording and playing back interactions with VisualWorks widgets, please contact support@silvermark.com