Toward a Platform for Visual Debugging

Rosario Molina, Alexandre Bergel
Pleiad Lab, DCC, University of Chile

1. Introduction
Debugging is essential in any software engineering activity. Programming environments are often shipped with (at least) one debugger to assist practitioners identifying and addressing execution anomalies [1].

Traditional debuggers communicate the interrupted application state to a practitioner by using three complementary perspectives: the execution stack on which one may crawl over the interrupted call flow, the source code of the selected stack frame annotated with the current position under interpretation, and the list of variables and their values for the selected stack frame. It has been shown that the current set of information provided by debugging tools is efficient to support non-trivial debugging tasks [2]. This paper sketches out Visual Debugger, a debugging framework that augments the traditional debugger with a visual and dynamic representation of the program internal state.

This short paper illustrates the core of visual debugger supported by the incremental construction of a debugger to keep track of side effect. Visual debugger is built on top of the Moldable Debugger, a framework for developing domain-specific debuggers [3] and the Roassal visualization engine [4].

2. Visual Debugger
Visual debugger is a simple instantiation of the Moldable debugger framework, implemented in Pharo1. Our visual debugger is composed of two classes:

- VSDDebugSession represents the model behind a visual debugger. Debugging operations (e.g., step-into, step-over, restart) are modeled as messages received by a debug session. The responsibility of this class is to (i) create and properly initialize the Roassal view and (ii) to offer the logic of the debugging operations.
- AbstractVisualStackDebugger contains the logical structure of the debugger graphical user interface.

The visual debugger framework is instantiated by subclassing these two classes. The code is distributed under the MIT license and is available on http://smalltalkhub.com/!/~RosarioM/VisualStackDebugger.

3. Step-by-step example
3.1 Framework instantiation
We will illustrate the visual debugger to visually represent side effects when performing a debugging operation. The example will visually highlight side effect occurring on the currently object for which code is stepped over. First, the two classes have to be subclassed:

```
AbstractVisualStackDebugger subclass: #VSDSideEffectDebugger
VSDDebugSession subclass: #VSDSideEffectDebugSession
instanceVariableNames: 'backgroundElements objectsAndValues'
```

The class VSDSideEffectDebugSession contains two variables:

- backgroundElements will contains the roassal elements that are currently visible. Keeping this list is useful to perform a layout at each debugging operation.
- objectsAndValues is a hash table that contains a snapshot of each object the application control flow is going through within the debugger. The snapshot is simply the set of instance variable values. A key is an object and the value is the values of each instance variables. This hash table is relevant to monitor side effects.

The variable initialization is simply carried out in the initialize method:

```
VSDSideEffectDebugSession >> initialize
super initialize.
objectsAndValues := Dictionary new.
backgroundElements := OrderedCollection new.
```

Subsequently, the two classes have to be hooked together and registered to be globally accessible. We then define the following three methods on the class side of VSDSideEffectDebugger:

VSDSideEffectDebugger class >> defaultTitle
"Title of the debugger"
'Side Effect Visual Debugger (example)'

VSDSideEffectDebugger class >> initialize
"Make sure the debugger is accessible"
sel register

VSDSideEffectDebugger class >> sessionClass
"Link the debugger to the session"
VSDSideEffectDebugSession

Figure 1: Accessing the side effect debugger

At that stage, our debugger is accessible within the default debugger as illustrated in Figure 1. Our debugger is accessible from a standard debugger.

3.2 Rendering the debugging session

A debug session class defines a number of methods representing debugging operations. Operations that are frequently used are stepInto:, stepOver:, and stepThrough:. These operations may be specialized to deal with visual elements by overriding them. In our situation, we will simply capture the step-over actions by overriding stepOver:. Each time the user will press the corresponding button in the debugger UI, the method will be invoked:

VSDSideEffectDebugSession >> stepOver: aContext
"Executed whenever the user press the step over button"

currentObject := aContext receiver.
variables := currentObject class instVarNames.
valueAssocs := variables collect: [:varName |
varName -> (currentObject instVarNamed: varName) ].
objectsAndValues
at: currentObject ifAbsentPut: [ valueAssocs asDictionary ].

The argument of stepOver: is an instance of the Context class, classes part of the Pharo runtime. The variable currentObject is the object receiver of the last message sent. We will therefore keep track of the modification of the current object. Variable names of the current object are kept in variables, and their values are in valueAssocs. For example, if the current object is the point 2 @ 3, then the variable valueAssocs contains the value \{'x' -> 2, 'y' -> 3\}.

The instance variable objectsAndValues contains a dictionary to keep a snapshot for all objects on which a step-over is executed on.

The remaining of the stepOver: method creates the visual representation of an instance variable:

```plaintext
shape := RTBox new
size: 5;
color: [:assoc | ((objectsAndValues at: currentObject) at: assoc key) ~
(valueAssocs asDictionary at: assoc key))
ifTrue: [ Color red ] ifFalse: [ Color gray ]].
```

RTBox is a Roassal class that describes a colored box large of 5 pixels. Each box indicates a variable. The red color means that the variable has been modified after having performed the step over. A gray box indicates that the represented variable is unmodified.

The following code creates visual elements for the variables and the encapsulating element for the current object.

```plaintext
variableElements := shape elementsOn: valueAssocs.
backgroundElement := RTBox elementOn: currentObject.

view add: backgroundElement.
view addAll: variableElements.

variableElements @ RTPopup.
backgroundElement @ RTPopup.

RTNest new
on: backgroundElement nest: variableElements.

backgroundElement @ RTDraggable.
backgroundElements add: backgroundElement.

RTGridLayout on: backgroundElements.

objectsAndValues at: currentObject put: valueAssocs
asDictionary.
view signalUpdate
```

3.3 Illustration

Figure 2 gives the example of a debugging session using the visual debugger. The figure has been obtained by tracing the expression `Compiler evaluate: '10 + 20'.` Ten step-over operations have been performed, as indicated by the number of visual elements in the debugger. The class Compiler defines 9 instance variables, represented by the inner boxes. The method for which the visualization has been obtained is:
Line 3 contains an assignment on the variable `class`, as indicated by the red box, contained in the third large box (Figure 2). A tooltip, obtained by putting the mouse above the variable, indicates the new value of `class`. Line 4, containing the call of `from:class:context:notifying:`, is represented by the fourth box.

Evaluating the expression `10 + 20` performs three side effects on the object instance of `Compiler`. Without out side effect debugger, such information would not be easy to obtain.

4. Future Work

Code execution debuggers have seen little improvement in the way they communicate and interact with the user. This extended abstract informally illustrates Visual Debugger, a framework being developed to enable a new range of code execution profilers augmented with interactive visual rendering. Our future work includes a debugger that illustrate interaction of objects.

References