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 practicers identifying and addressing execution anomalies [1].

Traditional debuggers communicate the interrupted application state to a practicer 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 Pharo[1]. 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.
```

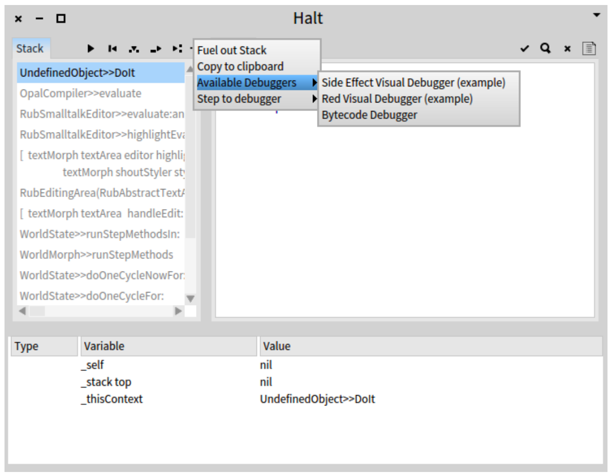
Subsequently, the two classes have to be hooked together and registered to be globally accessible. We then define the following three methods on the class side of `VSDSideEffectDebugger`:

[1] `http://pharo.org`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

, .
Copyright © 2014 ACM . . . $15.00.
http://dx.doi.org/10.1145/Reprinted from , [Unknown Proceedings], , pp. 1–3.

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

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

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



```
UndefinedObject>>DoIt
OpalCompiler>>evaluate
RubSmalltalkEditor>>evaluate:an
RubSmalltalkEditor>>highlightEva
[ textMorph textArea editor highli
        textMorph shoutStyler st
RubEditingArea(RubAbstractTextA
[ textMorph textArea  handleEdit:
WorldState>>runStepMethodsIn:
WorldMorph>>runStepMethods
WorldState>>doOneCycleNowFor:
WorldState>>doOneCycleFor:
```

Type	Variable	Value
	_self	nil
	_stack top	nil
	_thisContext	UndefinedObject>>DoIt

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 variables valueAssocs shape variableElements
     backgroundElement |
  super stepOver: aContext.

  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 a 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:

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

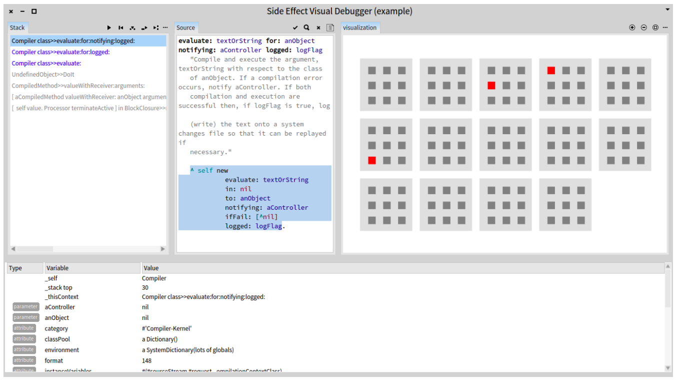
`RTBox` is a Roassal class that describes a colored box large of 5 pixels. Each box indicates a variable. The red color means that the variable has been modified ofter having performed the step over. A gray box indicates that the represented variable is unmodified.

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

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

  RTGridLayout on: variableElements.

  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:



Compiler class>>evaluate:for:notifying:logged:
Compiler class>>evaluate:for:logged:
Compiler class>>evaluate:
UndefinedObject>>DoIt
CompiledMethod>>valueWithReceiver:arguments:
[aCompiledMethod valueWithReceiver: anObject argumen
[self value. Processor terminateActive] in BlockClosure>>

```
evaluate: textOrString for: anObject
notifying: aController logged: logFlag
   "Compile and execute the argument,
textOrString with respect to the class
   of anObject. If a compilation error
occurs, notify aController. If both
   compilation and execution are
successful then, if logFlag is true, log

   (write) the text onto a system
changes file so that it can be replayed
if
   necessary."

   ^ self new
           evaluate: textOrString
           in: nil
           to: anObject
           notifying: aController
           ifFail: [^nil]
           logged: logFlag.
```

Type	Variable	Value
	_self	Compiler
	_stack top	30
	_thisContext	Compiler class>>evaluate:for:notifying:logged:
parameter	aController	nil
parameter	anObject	nil
attribute	category	#'Compiler-Kernel'
attribute	classPool	a Dictionary()
attribute	environment	a SystemDictionary(lots of globals)
attribute	format	148

Figure 2: Debugging session

```
1 Compiler >> evaluate: textOrStream in: aContext to: aReceiver
      notifying: aRequestor ifFail: failBlock logged: logFlag
2   | methodNode method value toLog itsSelection
      itsSelectionString |
3   class := aContext == nil ifTrue: [aReceiver class ] ifFalse: [
      aContext method methodClass].
4   self from: textOrStream class: class context: aContext notifying:
      aRequestor.
5   ...
```

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

[1] A. Zeller, Why Programs Fail: A Guide to Systematic Debugging, Morgan Kaufmann, 2005.

[2] J. Sillito, G. C. Murphy, K. De Volder, Questions programmers ask during software evolution tasks, in: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, SIGSOFT '06/FSE-14, ACM, New York, NY, USA, 2006, pp. 23–34. doi:10.1145/1181775.1181779.
URL http://people.cs.ubc.ca/~murphy/papers/other/asking-answering-fse06.pdf

[3] A. Chiş, T. Gîrba, O. Nierstrasz, The Moldable Debugger: A framework for developing domain-specific debuggers, in: B. Combemale, D. Pearce, O. Barais, J. J. Vinju (Eds.), Software Language Engineering, Vol. 8706 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 102–121. doi:10.1007/978-3-319-11245-9_6.
URL http://scg.unibe.ch/archive/papers/Chis14b-MoldableDebugger.pdf

[4] A. Bergel, D. Cassou, S. Ducasse, J. Laval, Deep Into Pharo, Square Bracket Associates, 2013.
URL http://rmod.lille.inria.fr/pbe2/