Unblocking the Debugger

Joe Winchester and Mark Jones

Debuggers are evolving from tools that trap and help diagnose errors into an integral part of the development environment. Their most useful task is arguably the ability to step through code, line by line, allowing the programmer to follow message flow. The debugger supplied with IBM's Smalltalk/Visual Age falls short of this purpose where the code is embedded within blocks. To locate this code often requires traversing the internals of seemingly irrelevant methods. In this article we describe an enhancement to the IBM Smalltalk/Visual Age debugger that allows code within blocks to be debugged more easily.

Our approach to solving the problem first involved understanding the mechanics of the existing debugger, and then observing and automating how a programmer manually debugs his or her way through methods that contain code as blocks.

THE WAY THINGS ARE

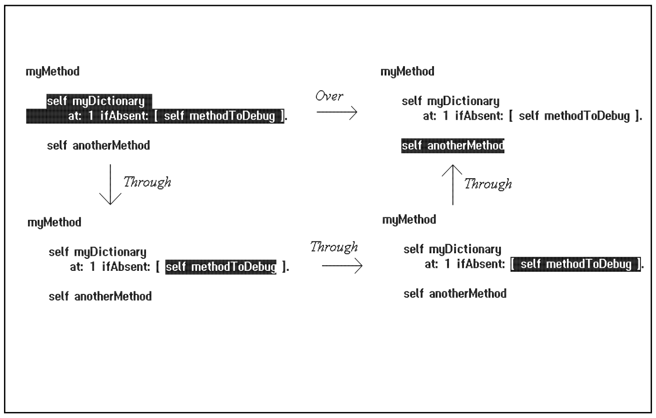
The supplied debugger comes with four buttons: *into, over, return*, and *resume*. Pressing into allows the next message about to be sent to be debugged, and pressing *over* allows the message to be skipped past. When the message contains a block as an argument or receiver, pressing *over* skips past the message and past the code contained within the block. There are some exceptions to this, for example: **Integer>>to:** *anInteger* **do:** [ ], and **Boolean>>ifTrue:** *[ ]*. Pressing *over* on these messages takes the programmer to the code within the block. This happens because the compiler inlines the code to implement messages directly in the sender for optimization purposes, and no actual message sent occurs at runtime. With all other messages that contain blocks as receivers or arguments, debugging the code within the block requires pressing into and then traversing the method internals to find the relevant value or value: message that will evaluate the block. To do this with **Dictionary>>at:** aKey **ifAbsent:** [ ] requires pressing *into* once, *over* seven times, and a final *into* to reach the block. If the message being debugged is a loop, e.g. **Collection>>detect:** *[ ]* **ifNone:** *[ ]*, repeated debugging of the method internals is required to visit the code within the block for each iteration of the loop. The problem here is that you should be concentrating on your code, instead of worrying about reading and understanding the internals of methods that are basically language constructs.

THE WAY THINGS SHOULD BE

We found ourselves wishing for behavior that would act the way *over* does for the inlined messages, i.e. it would always take the programmer straight to the code within the block. This functionality we decided to name *through* and implement with a new button on the debugger. Figure 1 shows how *over* takes the programmer past the code within the **Dictionary>>at:** *aKey* **ifAbsent: [ ]** block. Pressing *through* would go into the code within the block and back to the method body without having to traverse irrelevant method internals.

MIMIC REAL LIFE

As with any good OO solution, the answer lies in observing the real world and mimicking it in code. Therefore, let us examine the scenario "How does a programmer debug the method **Dictionary>>at:** *aKey* **ifAbsent:** *[ ]* ." If the pro-


```
myMethod

    self myDictionary
        at: 1 ifAbsent: [ self methodToDebug ].

    self anotherMethod
```
```
myMethod

    self myDictionary
        at: 1 ifAbsent: [ self methodToDebug ].

    self anotherMethod
```
```
myMethod

    self myDictionary
        at: 1 ifAbsent: [ self methodToDebug ].

    self anotherMethod
```
```
myMethod

    self myDictionary
        at: 1 ifAbsent: [ self methodToDebug ].

    self anotherMethod
```

Figure 1. Depicting how *over* takes the programmer past the code within the Dictionary>>at: aKey if Absent: [ ] block.

grammer wishes to reach the code within the block, they step into the method. An inexperienced programmer might then continue to step into each successive message until the block's embedded code is reached. A more experienced programmer knows to only examine messages that are likely candidates to evaluate the block. In both cases, the programmer succeeds once the original method that contains their source becomes active again.

We shall attempt to automate the behavior of the rookie Smalltalker. The key to our solution will come from an understanding of the mechanics behind the debugger and attempting to leverage this to implement the request "Step into a message with a block and continue to step into each successive message send until the code within the block is about to be executed."

DEBUGGER BASICS

Source code written in Smalltalk is compiled into instances of **CompiledMethod,** each of which contains a stream of bytecodes. Each bytecode is an intermediate representation of an instruction that is interpreted by the virtual machine. Instances of class **Process** execute code, and the distinguished instance of **ProcessorScheduler** is responsible for controlling process execution.[1,2,3]

The supplied debugger runs in its own process separate from the method being debugged. It uses the method,

ProcessorScheduler>>debugResume: *aProcess* **when:** *aConditionBlock* **do:** *aContinueBlock*,

to cause the process being debugged to execute until *aConditionBlock* evaluates true. Once this has occurred *aContinueBlock* is evaluated. While *aConditionBlock* is being tested the debugger must wait until *aContinueBlock* is evaluated. This is achieved with a semaphore, which the process that the debugger executes waits on, and is signaled by the *ContinueBlock*. To ensure that *aConditionBlock* is tested between each message send, the messages **breakEveryByteCode:** *true* and **useByteCodeMask:** *true* to the process being debugged. Each process has a number of frames that can be thought of as a stack of **CompiledMethod** instances.

THE ENHANCED DEBUGGER

To begin adding our new behavior we will subclass the supplied debugger and provide a new button *"Through."*

```
EtDebugger subclass: EnhancedDebugger
    instanceVariableNames: 'throughButtonWidget'
    classVariableNames: ''
    poolDictionaries: ''
```

To automate our rookie's behavior we will use **ProcessorScheduler>>debugResume:** *aProcess* **when:** *aConditionBlock* **do:** *aContinueBlock* and construct *aConditionBlock* that evaluates true when the method being debugged is active once more. A process can derive its active method using its start frame:

```
Process>>#activeMethod

    ^self methodAtFrame: self startFrame
```

It is equally important to check that it is still possible to return to the method in which *through* is being processed. This is because, during execution, a return may have been processed, causing the method to be exited.

The test can be accomplished by determining if the method is still in the frame stack.

```
Process>>#cannotReturnTo: aMethod


    "Loop around the frames and get the method at
 each.  Return false
    if any of these are the method argument"

    self startFrame to: self numberOfFrames - 1 do: [ :index |
        ( self methodAtFrame: index ) == aMethod
            ifTrue: [ ^false ] ].
    ^true
```

We are ready to implement a method on the enhanced debugger to process *through*.

```
EnhancedDebugger>>#processStepThrough

    | sem currentProcess currentMethod |

    "don't continue if selected process is not resumable"
    self isSelectedProcessResumable ifFalse: [ ^self ].

    "don't continue if a source change has been specified"
    self changeRequest ifFalse: [ ^self ].

    "ensure there is a selected method"
    self selectTopFrameIfNone.
    currentProcess := self selectedProcess.
    currentMethod := self selectedMethod.

    "Break every byte code of the process and resume execution
 until the method is active once more or is not in the method
 stack"
    currentProcess
        breakEveryBytecode: true ;
        useBytecodeMask: true.
    sem := Semaphore new.
    Processor
        debugResume: currentProcess
        when: [
            currentProcess activeMethod == currentMethod or: [
            currentProcess cannotReturnTo: currentMethod ] ]
        do: [ :hasStackOverflowOccured |
            hasStackOverflowOccured
                ifTrue: [ self removeProcess: currentProcess ].
            sem signal ].
    sem wait.

    "refresh debugger"
    self isOneProcessSelected ifTrue: [ self refreshAfterStep ].
```

THROUGH VERSUS OVER

This implementation automates how the rookie programmer debugs methods with *source* as block arguments. If *through* is used instead of *over* to debug a method containing a block with inlined *source*, the debugger takes the programmer straight into the code within the block. Once inside the block *over* can be used to step past each message send. However, it becomes easy to break out and into the method internals we are attempting to bypass, by pressing *over* after the last message inside the block where *through* should have been pressed. With methods that loop such as **Collection>>detect: [ ] ifNone: [ ],** switching between *through* and over to debug each iteration became an unnecessary distraction, the very thing we are trying to remove. The through button is needed to perform only our new processing when it is required, and to process *over* otherwise. This way the programmer can repeatedly press *through* to visit every message send within their source, with the *over* button being the exception required to genuinely skip past the source within a block.

A BETTER SOLUTION

We shall improve our solution by repeating the earlier principle of observing real life and mimicking it in code. Understanding as to whether *through* or *over* is required (when using the debugger) comes from looking at the highlighted portion of the source.

The three situations when *through*, rather than *over*, is required are:

- Sending a message in which the receiver is a block containing inline source;
- Sending a message in which any one of the arguments is a block containing inline source; and
- Leaving a block on completion of the inline source.

A segment of highlighted source is equivalent to a parse node. Different classes of parse node represent different code constructs. Thus, we will ask each parse node **#isThroughRequired** and specialize to recognize the three situations.

The superclass of all parse nodes is **EsParseNode.** By default through is not required.

```
EsParseNode>>#isThroughRequired

    ^false
```

The node representing the first two of our code constructs is **EsMessageExpression.** The decision is deferred to the receiver node to determine if it is a block and to the message pattern node to determine if any argument is a block.

```
EsMessageExpression>>#isThroughRequired

    ^self receiver isThroughRequired
        or: [ self messagePattern isThroughRequired ]
```

The block node is **EsBlock.**

```
EsBlock>>#isThroughRequired

    ^true
```

The message pattern node for a keyword message is **EsKeywordPattern.** The decision is deferred to each argument node to determine if any are blocks.

```
EsKeywordPattern>>#isThroughRequired

    self arguments
        detect: [ :anArgument |
            anArgument isThroughRequired ]
        ifNone: [ ^false ].
    ^true
```

The third of our situations is already covered by the block node **EsBlock.**

BACK TO THE ENHANCED DEBUGGER

The next step is for the debugger to determine the parse node for the next message to be sent within the currently selected method.

```
EnhancedDebugger>>#currentParseNode

    ^self parseTree == nil ifFalse: [
        self parseTree
        nodeWhichContainsPC: (self currentPC: self
selectedFrameIndex )
        hasDropped: self selectedProcess hasDropped ]
```

Finally, we are ready to implement a method on the debugger that will perform *through* processing when required and *over* at all other times. This is the method that will be called by the *through* button, allowing the user to have a single button that permits them to visit every message sent within their source.

```
EnhancedDebugger>>#processThrough

    | aNode |

    aNode := self currentParseNode.

    "process as through if selected method is the active method"
    (self selectedFrameIndex ==
      self selectedProcess startFrame

    "and, if node requires through"
    and: [ aNode notNil and: [ aNode isThroughRequired ]])
        ifTrue: [ self processStepThrough ]
        ifFalse: [ self processStepOver ]
```

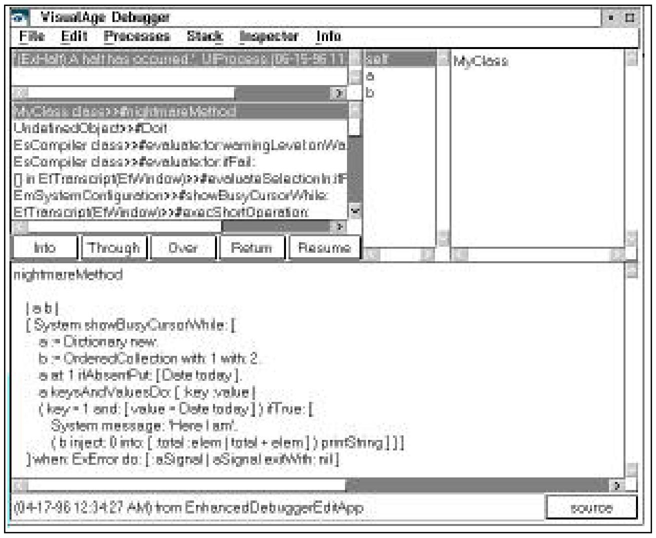
CHANGING THE USER INTERFACE

The new '*through*' button is placed between the existing '*into*' and '*over*' buttons. To achieve this the method #createWorkRegion is specialized as follows:

```
EnhancedDebugger>>#createWorkRegion

    super createWorkRegion.

    throughButtonWidget := self
```


VisualAge Debugger
File Edit Processes Stack Inspector Info
Into Through Over Return Resume
source

```
(ExHalt) A halt has occurred.' UIProcess (06-15-96 11
MyClass class>>#nightmareMethod
UndefinedObject>>#DoIt
EsCompiler class>>#evaluate:for:warningLevel:onWa
EsCompiler class>>#evaluate:for:ifFail:
[] in EtTranscript(EtWindow)>>#evaluateSelectionIn:ifF
EmSystemConfiguration>>#showBusyCursorWhile:
EtTranscript(EtWindow)>>#execShortOperation:
```

```
self
a
b
```

```
MyClass
```

```
nightmareMethod

  | a b |
  [ System showBusyCursorWhile: [
    a := Dictionary new.
    b := OrderedCollection with: 1 with: 2.
    a at: 1 ifAbsentPut: [ Date today ].
    a keysAndValuesDo: [ :key :value |
    ( key = 1 and: [ value = Date today ] ) ifTrue: [
      System message: 'Here I am'.
      ( b inject: 0 into: [ :total :elem | total + elem ] ) printString ] ] ]
  ] when: ExError do: [ :aSignal | aSignal exitWith: nil ]
```

```
(04-17-96 12:34:27 AM) from EnhancedDebuggerEditApp
```

Figure 2. Represents a method that shows the effectiveness of the *through* button.

```
        newButtonWidget: 'Through'
        selected: #processThrough.
    self stepIntoButtonWidget setValuesBlock: [ :w |
        w rightWidget: throughButtonWidget ].

    throughButtonWidget setValuesBlock: [ :w |
        w
            topAttachment: XmATTACHOPPOSITEWIDGET;
            topWidget: self stepToReturnButtonWidget;
            leftAttachment: XmATTACHPOSITION;
            leftPosition: 11;
            rightAttachment: XmATTACHWIDGET;
            rightWidget: self stepOverButtonWidget;
            bottomAttachment: XmATTACHWIDGET;
            bottomWidget: self textWidget parent].

    self stepOverButtonWidget setValuesBlock: [ :w |
        w leftPosition: 22 ].

    self stepToReturnButtonWidget setValuesBlock:
        [ :w |w leftPosition: 33 ].

    self resumeButtonWidget setValuesBlock: [ :w |
        w leftPosition: 44 ].
```

To switch over to the enhanced debugger, the following method must be evaluated on the Transcript.

```
    System startUpClass debuggerClass: EnhancedDebugger
```

Evaluating with **EtDebugger** will reset to the original supplied debugger.

LIMITATIONS

The implementation now combines the way the rookie programmer debugs methods containing source inlined as blocks, with the decision process made by the more experienced programmer who knows when such debugging is required.

However, there are two situations to be aware of:

- Exceptions raised while *through* processing in the receiving block of methods such as **Block>>#when:do:** do not lead to stepping *through* the exception handling block code. This situation occurs because *through* processing is not required to debug all code constructs within a block. In these cases the exception is raised during the supplied *over* processing. Therefore, the exception handling block is not stepped *through*. Due to a feature of the supplied implementation of *over* if an exception is raised, the debugger reactivates prematurely resulting in a mismatch between the method being displayed in the debugger and the active method.

- A performance problem exists if many messages are sent between the point when through processing starts and the block that caused *through* processing to be required is evaluated. This occurs because *through* processing involves executing the intervening code one bytecode at a time, and this has a performance overhead. Methods where this is noticeable are rare, and are generally outside the standard block messages included with the core IBM Smalltalk. A hypothetical method of the type that would be affected is **DataBaseFile>>readAt:** ***aKey*** **ifNoRecordFound:** ***[ ]*****.** Here, many messages will potentially be sent before the block is evaluated.

CONCLUSIONS

The enhanced debugger fulfills all the initial goals, namely to enable the developer to focus on tracing the execution of the code in which they are interested. To see how effective it can be try to debug the method shown in Figure 2, with and without the use of the *through* button. All of the code required to implement the debugger is included in this article, and can also be downloaded from the Visual Age CompuServe member-supplied forum, and from ftp://ftp.smalltalk.com/pub/ibm.smalltalk/win_debug.zip. Having used the *through* button for some time we now find it to be an invaluable aid to debugging. We hope you will find it as useful, and we welcome all feedback.

The authors would like to thank Tim Morrison of Unity Software and David Cotton for their imput in preparing this article, as well as Doug Shaker of The Smalltalk Store for putting the source code on his ftp server. S

References

1. Goldberg, Adele, and Robson, David, Smalltalk-80: The Language, Addison Wesley ISBN 0-201-13688-0.
2. IBM Smalltalk Programmer's Reference, SC34-4493-02.
3. Budd, Timothy, A Little Smalltalk, Addison-Wesley ISBN 0-201-10698-1, 1987.

Joe Winchester and Mark Jones are consultants working for Computec International, Costa Mesa, CA. They are currently working on building application frameworks using Visual Age for a healthcare company. They can be contacted at 103276, 233@Compuserve.com.