CS 535 Object-Oriented Programming & Design
Fall Semester, 2003
Doc 7 Testing, Debugging, Object
Contents

References

VisualWorks Application Developer's Guide, doc/vwadg.pdf in the VisualWorks installation. Chapter 9 & 23.

Reading

(DevGuide) Chapter 23 Coding Tools
Pages 476-478, 483-488 Code Critic & Unit Testing Sections

(DevGuide) Chapter 1 - VisualWorks Environment Inspectors section

(DevGuide) Chapter 9 - Debugging Techniques pp 197-206

(Beck) Chapter 7 - Formatting

Testing

Johnson's Law

If it is not tested it does not work

Types of tests

- Unit Tests

  Tests individual code segments

- Functional Tests

  Test functionality of an application

Why Unit Testing

If it is not tested it does not work

The more time between coding and testing

- More effort is needed to write tests
- More effort is needed to find bugs
- Fewer bugs are found
- Time is wasted working with buggy code
- Development time increases
- Quality decreases

Without unit tests

- Code integration is a nightmare
- Changing code is a nightmare

Unit Tests Must be Easy To Run

Must be able to

- Easily run many tests at once
- Allow others to run the tests
- Keep the tests for later
- Scale with more developer and project size

Test stored in a workspace

- Do not work in any sizable project
- Do not work well with multiple programmers
- Are easily lost
- Are not run very often

Testing First

First write the tests

Then write the code to be tested

Writing tests first:

- Removes temptation to skip tests
- Makes you define of the interface & functionality of the code before

SUnit

Testing framework for automating running of unit tests in Smalltalk

In SUnit

- Programmer manually writes the test
- SUnit automates the running of the test
- Simplifies finding tests that fail
-

Ports to other languages can be found at:
http://www.xProgramming.com/software.htm

Two GUI Interfaces for viewing Test Results

- TestRunner
  Already loaded in Image

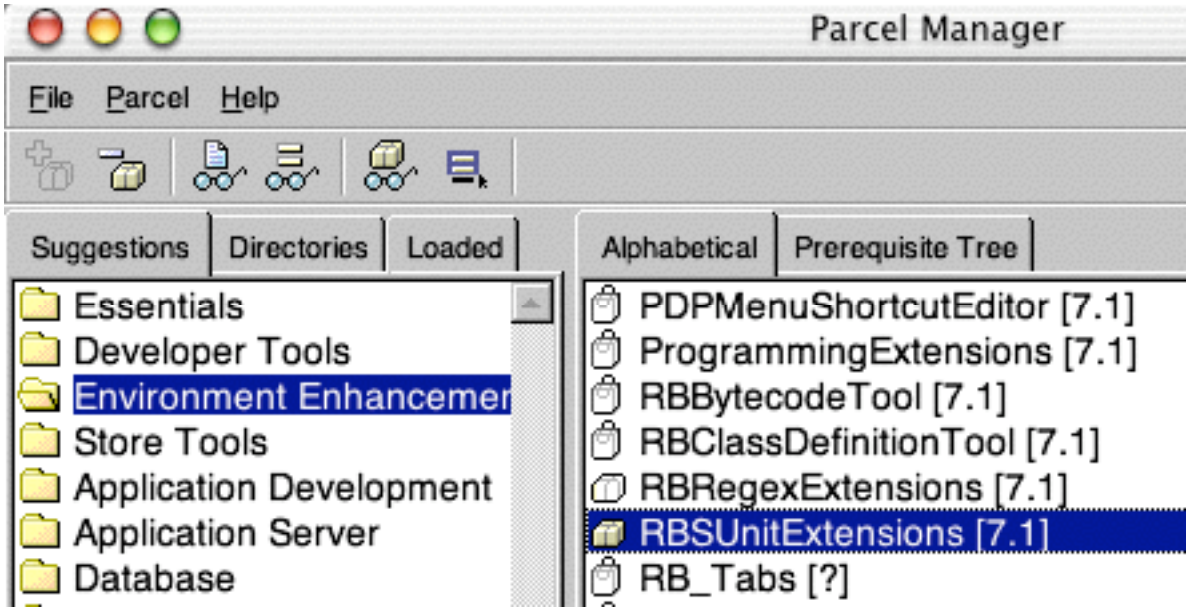
- Browser SUnit Extensions
  - Easier to run individual tests
  - Needs to be loaded

Loading Browser SUnit Extensions

Open the Parcel Manager by selecting the “Parcel Manager” item in the “System” menu of the launcher.

In the parcel manager:

- Select the “Suggestions” tab
- In the Left list pane select “Environment Enhancements”
- In the right list pane find and select "RBSUnitExtensions"
- Click on the “Load Parcels’ icon 
- The changes only affects new system browsers



How to Use SUnit

We will test the following class

```
Smalltalk defineClass: #Counter
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'count '
    classInstanceVariableNames: ''
    imports: ''
    category: 'Course-Examples'
```

Class method

```
new
    ^super new initialize
```

Instance Methods

```
count
    ^count

decrease
    self increment: 1          "Note the error"

increase
    self increment: 1

increment: anInteger
    self halt.
    count := count + anInteger

initialize
    count := 0
```

Creating and Using Unit Tests

1. Make test class a subclass of TestCase

```
Smalltalk defineClass: #TestCounter
superclass: #{XProgramming.SUnit.TestCase}
indexedType: #none
private: false
instanceVariableNames: ''
classInstanceVariableNames: ''
imports: ''
category: 'Course-Examples'
```

You do not have to remember the full name of TestCase. Just use the name TestCase. The browser will expand it to the full name for you.

2. Make test methods

The framework treats methods starting with 'test' as test methods

```
testIncrease
    | aCounter |
    aCounter := Counter new.
    self assert: aCounter count = 0.

    aCounter increase.
    self assert: aCounter count = 1.
```

TestCase methods of interest

Methods to assert conditions:

assert: aBooleanExpression

deny: aBooleanExpression

should: [aBooleanExpression]

should: [aBooleanExpression] raise: AnExceptionClass

shouldnt: [aBooleanExpression]

shouldnt: [aBooleanExpression] raise: AnExceptionClass

signalFailure: aString

setUp

Called before running each test method in the class.

Used to:

- Open files
- Open database connections
- Create objects needed for test methods

tearDown

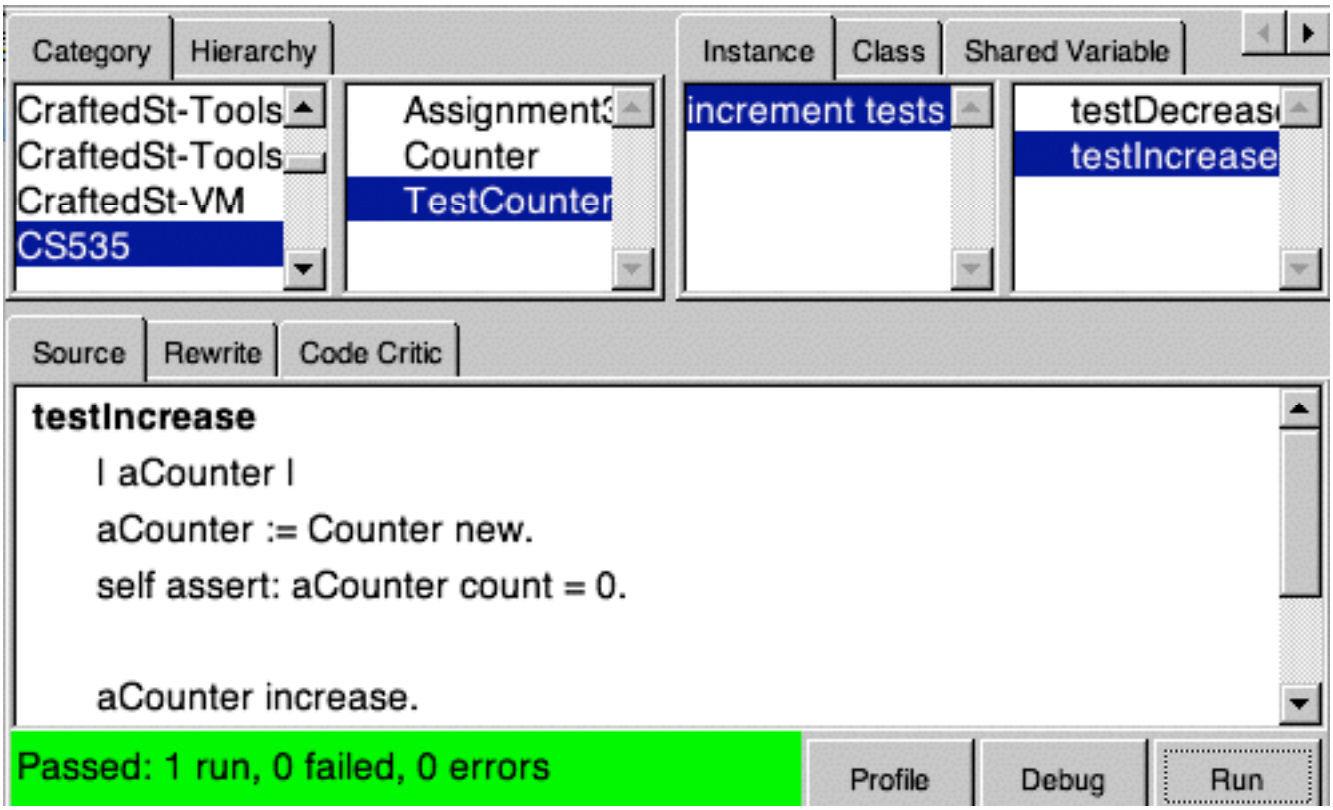
Called after running each test method in the class.

Used to:

- Close files
- Close database connections
- Nil out references to objects

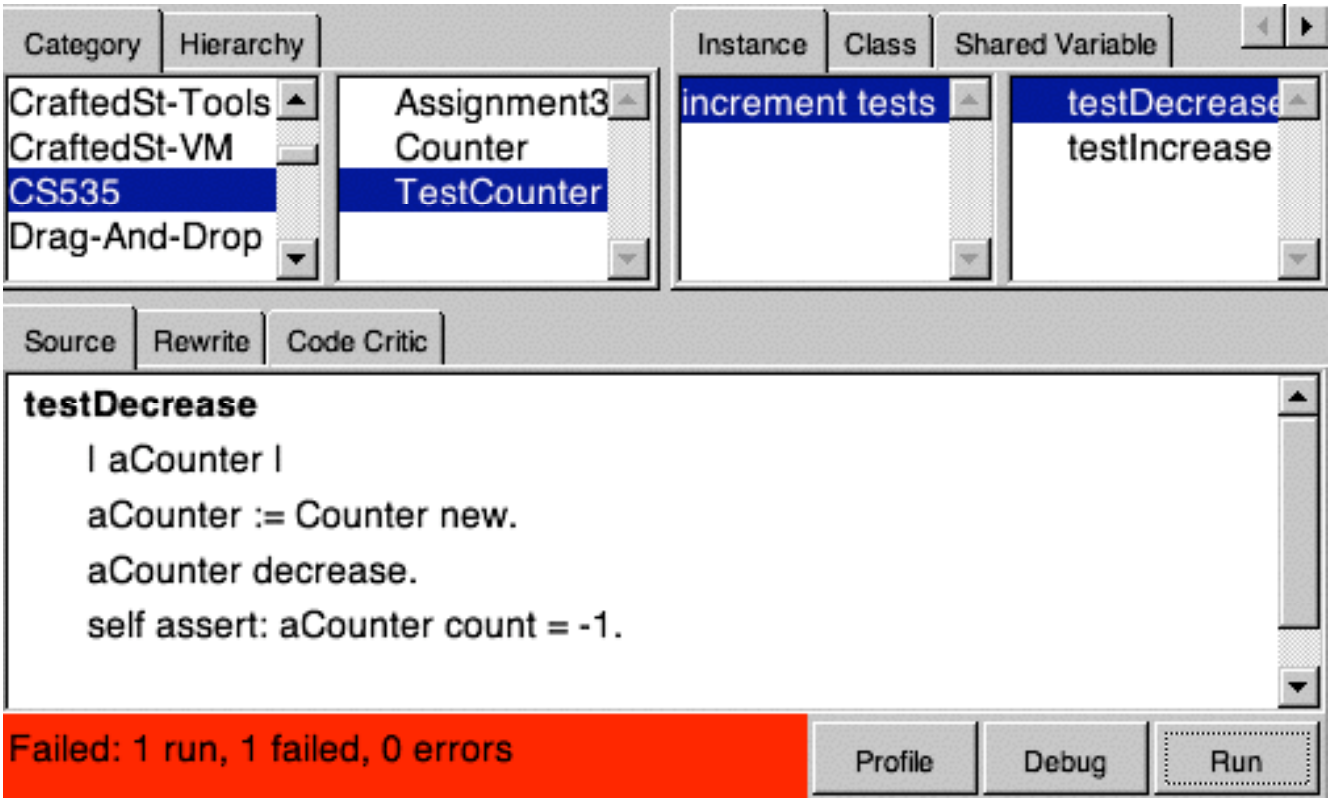
Running the Test using Browser SUnit Extensions

If you have loaded RBSUintExtensions you will see a Run button below the code pane in any method whose selector starts with “test” in a subclass of TestCase. Just click on the run button to run the test. It the test passes you will get a green bar below the code pane, if it fails you will get a red bar. If you get a red bar you can click on the “Debug” button to run the test in the debugger. When you have more tests, you can select just the class and the run button will run all the tests in the class. If the test(s) pass you will get a green bar as shown below.



A Failed Test

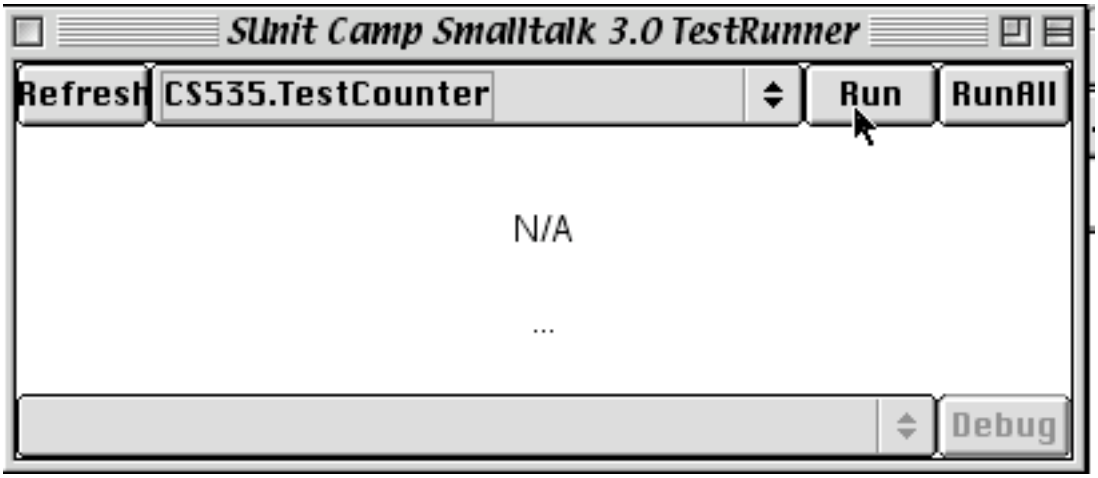
When a test fails you get a red bar.



How to Use TestRunner

- Start TestRunner

  TestRunner open



- Select test class and click on "Run"

  The list pane between Refresh and Run contains a list of all the Test classes in the system. Select the test class you wish to run. The click on the run button. If all tests work, then the middle panel will turn green. If some of them fail, then the middle panel will turn red. The list pane on the bottom of the window will list all tests that failed. Select one and click on the Debug button to open the debugger on the test method that failed.

  Note VisualWorks ships with a number of test classes in the image. When working on a project I remove them. Then all the test classes in the image are for my project. I then always click on the RunAll button to run all the tests. Sometimes when you change a method, you break code that uses that method. Running all the tests uncovers those problems early.

More Test Examples to Show Syntax

```
Smalltalk.CS535 defineClass: #TestCounter
	superclass: #{XProgramming.SUnit.TestCase}
	indexedType: #none
	private: false
	instanceVariableNames: 'counter '
	classInstanceVariableNames: ''
	imports: ''
	category: 'Course-Examples'
```

CS535.TestCounter methodsFor: 'testing'

```
setUp
	counter := Counter new.

tearDown
	counter := nil.

testDecrease
	counter decrease.
	self assert: counter count = -1.

testDecreaseWithShould
	"Just an example to show should: syntax"
	counter decrease.
	self should: [counter count = -1].

testIncrease
	self deny: counter isNil.
	counter increase.
	self assert: counter count = 1.

testZeroDivide
	"Just an example to show should:raise: syntax"
	self
		should: [1/0]
		raise: ZeroDivide.

	self
		shouldnt: [1/2]
		raise: ZeroDivide
```

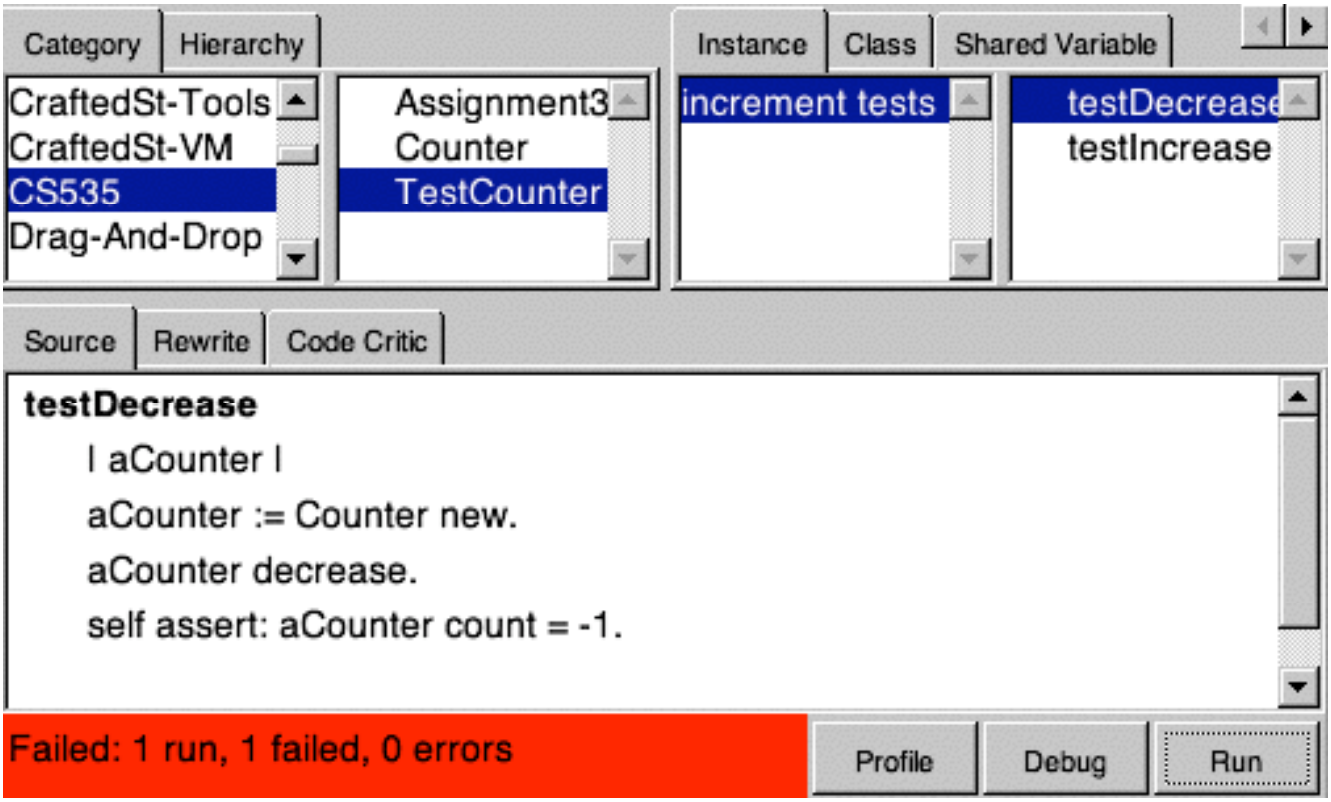
Debugging

Ways the debugg is called

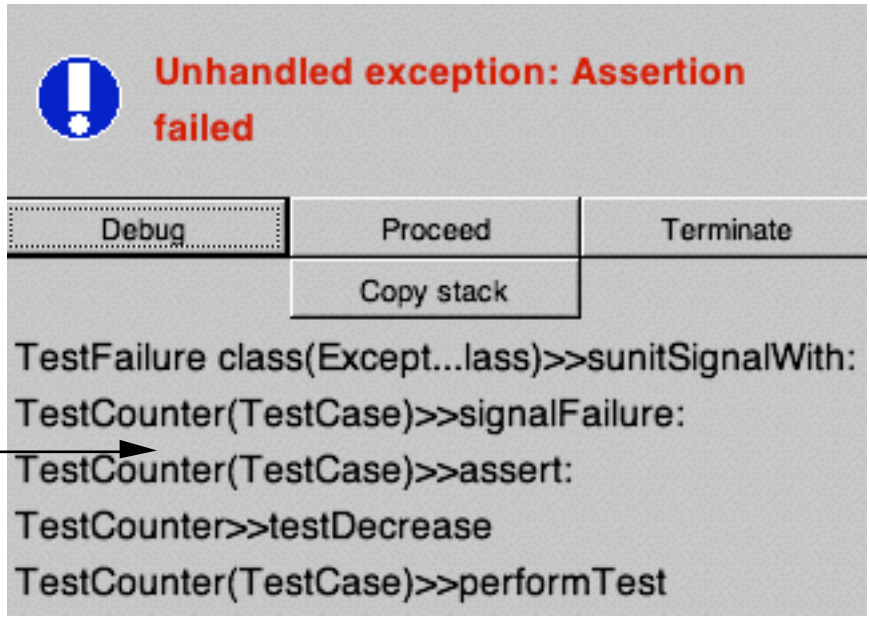
- Executing code in workspace using "Debug it"
- Executing "self halt"
- An error raised in your code
- Using probes
- Debug button in Unit Testing

To illustrate how the debugger works we will go through an example

When we run our testDecrease method it fails and shows a red bar.

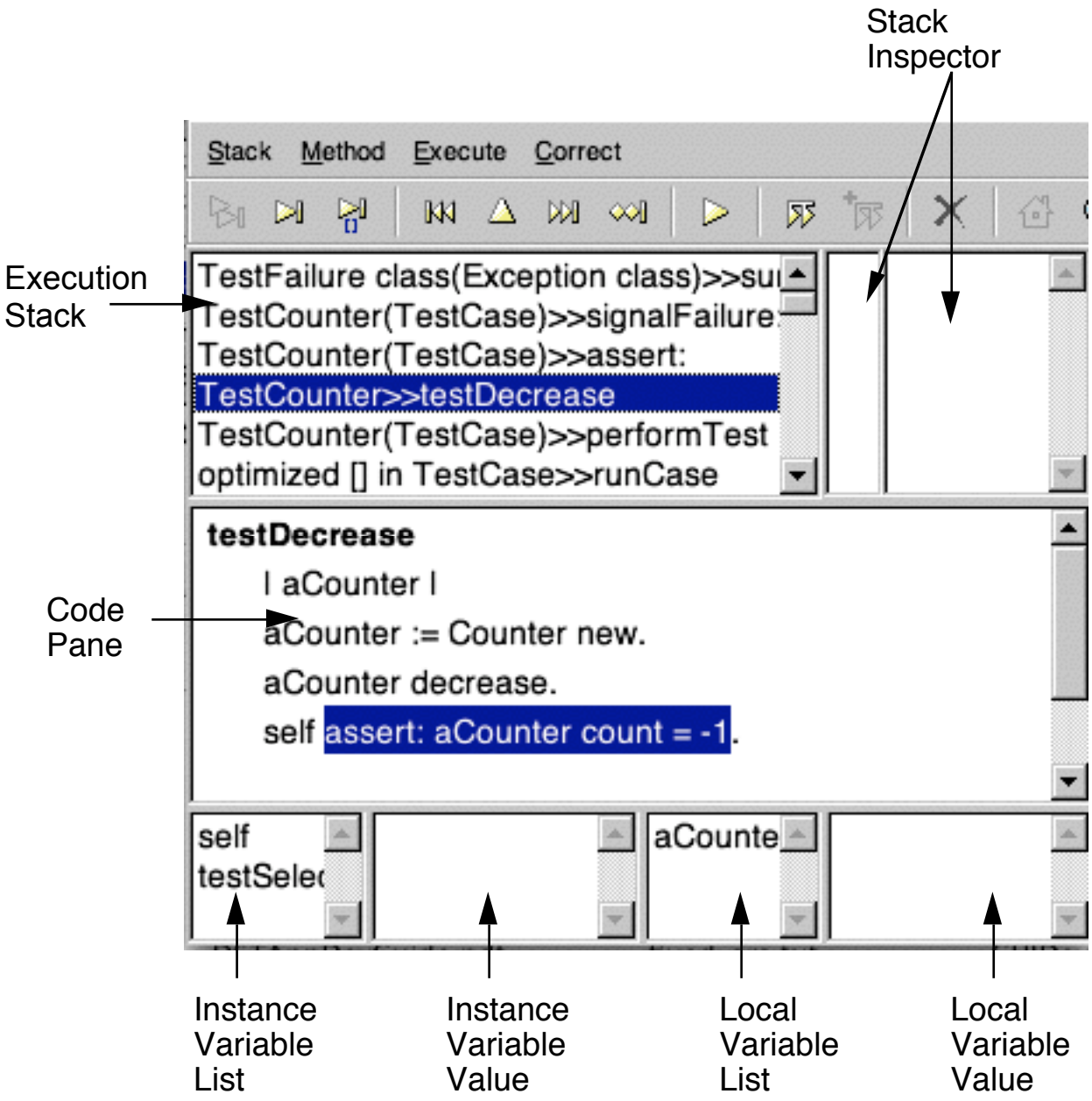


If one clicks on the Debug button you get the standard exception window.



Execution Stack

When you click on the debug button in the exception window, you get the debugger. The debugger looks like:



Debugger Components

Execution Stack Pane

Lists the methods calls that are on the execution stack. Selecting any of the listed method calls shows the state and code for the method

Code Pane

Shoes the source code of the selected method. You can make changes to the code shown. To make changes just edit the code and accept the changes. When you run the code, it starts from the selected method with the new changes

Instance Variable List Pane

This pane lists all the instance variables in the receiver of the method selected in the execution stack pane. Selecting an instance variable will show its current value in the instance variable value pane.

Instance Variable Value Pane

Displays the value of the selected instance variable. By editing this pane you can change the value of the instance variable. Just type in the new value or code that will return the new value and accept the changes.

Local Variable List Pane

This pane lists all the local variables in the method selected in the execution stack pane. Selecting a local variable will show its current value in the local variable value pane.

Local Variable Value Pane

Displays the value of the selected local variable. By editing this pane you can change the value of the local variable. Just type in the new value or code that will return the new value and accept the changes.

Step 

Executes the currently selected message in the Code pane

Step Into 

Calls the currently selected message in the code pane and shows the code for the method. Allows you to step through this method.

For more information on using the Debugger see Chapter 9 Debugging Techniques of the VisualWorks Application Developer's Guide.

Self halt Example

The statement "self halt." opens the exception window when executed, which allows one to enter the debugger.

```
Smalltalk defineClass: #Counter
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'count '
    classInstanceVariableNames: ''
    imports: ''
    category: 'Course-Examples'
```

Class method

```
new
    ^super new initialize
```

Instance Methods

```
count
    ^count

decrease
    self increment: 1          "Note the error"

increase
    self increment: 1

increment: anInteger
    self halt.
    count := count + anInteger

initialize
    count := 0
```

Sample Test code in Workspace

Execute the following code with "Do it"

```
| count |
count := Counter new.
count
    decrease;
    decrease;
    increase.
```

When the statement "self halt" is executed you will see the exception window.

Object

All 'things' in Smalltalk are objects

Objects are created from classes

The class Object is the parent class of all classes

Object class contains common methods for all objects

Determines behavior for all objects

Some Important Behavior Defined in Object

printString

Returns a string representation of the receiver

Similar to toString in Java

isNil, notNil

Tests to see if the receiver has been initialized or is still nil

```
| a |                          Result printed
Transcript
    clear;
    print: a isNil;            true
    cr;
    show: a printString;        'nil'
    cr;
    print: a class;             UndefinedObject
    cr.
a := 5.
Transcript
    print: a isNil;             false
    cr;
    show: a printString;       5
    cr;
    print: a                   5
    cr;
```

When the above code is compiled the compiler notices that a is used before it is assigned a value. The compiler will ask you if want use a before it is assigned.

printString

Since implemented in Object all objects inherit the method

printString is sent to an object when

Expression is evaluated with "Print it"

When the object is displayed in some VW windows

When sent to the Transcript via print:

Overriding Default printString Behavior

Implement printOn: not printString

printOn:'s argument is a stream

Stream Writing methods:

nextPutAll: aString
nextPut: aCharacter
print: anObject
cr
space
tab
crtab

Example - Counter

Add the following method to the Counter class

```
Counter>>printOn: aStream
    aStream
        nextPutAll: 'Counter(';
        print: count;
        nextPutAll: ')'
```

Now execute the following with a "Print it"

```
| test |
test := Counter new.
test increase.
^test
```

Now instead of printing “a Counter” you will get “Counter(1)”

Who does printOn: change printString?

printString uses printOn:

```
Object>>printString
	"Answer a String whose characters are a description of the receiver."

	| aStream |
	aStream := WriteStream on: (String new: 16).
	self printOn: aStream.
	^aStream contents
```

Equality

All objects are allocated on the heap

Variables are references (like a pointer) to objects

A == B
Returns true if the two variables point to the same location

A = B
Returns true if the two variables point to equivalent objects

In Smalltalk you want to use '=' nearly all the time

A ~= B
Means (A = B) not

A ~~ B
Means (A == B) not

Equality Example - Counter

```
| a b |
a := SimpleCircle origin: (0 @ 0) radius: 1.
b := a.
```

Now a == b and a= b are both true

```
a := SimpleCircle origin: (0 @ 0) radius: 1.
b := SimpleCircle origin: (2 @ 5) radius: 3.
```

Now a == b and a= b are both false

```
a := SimpleCircle origin: (0 @ 0) radius: 1.
b := SimpleCircle origin: (0 @ 0) radius: 1..
```

Now a == b is false

Should a = b be true or false?

Defining the Meaning of =

Object

Does not know what ='s means for your class

Default definition is to use ==

If you override = also override #hash