CS 535 Object-Oriented Programming & Design
Fall Semester, 2013
Doc 6 Classes, Polymorphism, Testing
Sept 12 2013

self & super

self

Refers to the receiver of the message (current object)

Methods referenced through self are found by:

Searching the class hierarchy starting with the class of receiver

super

Refers to the receiver of the message (current object)

Methods referenced through super are found by:

Searching the class hierarchy starting the superclass of the class containing the method that references super

Why Super

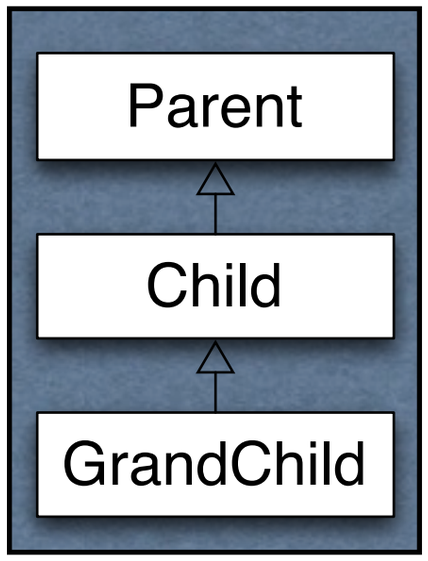
Super is used when:

The child class overrides a method
Needs to call overridden method

Common Pattern

```
ClassPointSubclass>>initialize
    super initialize.
    z := 0.
```

self and super Example




```
Parent>>name
   ^'Parent'
```

```
Child>>name
   ^'Child'

Child>>selfName
   ^self name

Child>>superName
   ^super name
```

```
GrandChild>>name
   ^'GrandChild'
```

Code	Output
| grandchild |	
grandchild := Grandchild new.	
Transcript	
show: grandchild name;	Grandchild
cr;	
show: grandchild selfName;	Grandchild
cr;	
show: grandchild superName;	Parent
cr;	

How does this work

grandchild selfName

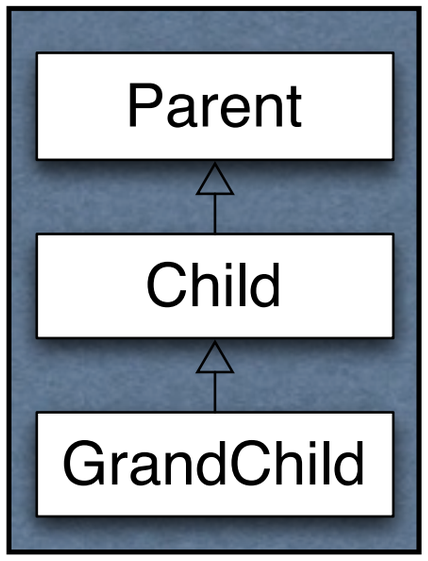
Receiver is grandchild object
Code in selfName method is ^self name
To find the method "self name" start search in Grandchild class

grandchild superName

Receiver is grandchild object
Code in superName method is ^super name
superName is implemented in Child class

To find the method "super name" start search in the superclass of Child

Why doesn't super = receiver's parent class?




```
Parent>>name
   ^'Parent'
```

```
Child>>name
   ^super name , 'Child'
```

```
| trouble |
trouble := Grandchild new.
Transcript
    show: grandchild name;
```


If super referred to the parent class of the receiver the above code would result in an infinite loop. The receiver is a GrandChild object so the parent is Child. So in Child>>name "super name" would refer to Child>>name.

Class Methods

```
ClassPoint class>>origin
	^self x: 0 y: 0

ClassPoint class>>x: xNumber y: yNumber
	^(self new)
		x: xNumber;
		y: yNumber;
		yourself
```

```
center := ClassPoint origin.
center x
"Returns o"
```

```
ClassPoint class>>new
	^super new initialize
```

new & initialize

```
ClassPoint>>initialize
    x := 0.
        y := 0.
```

```
ClassPoint class>>new
        ^super new initialize
```

ClassPoint new

SomeParentClass new initialize

aClassPointObject initialize

SomeParentClass new returns a ClassPoint object

Initialization and Inheritance

```
Smalltalk.Core defineClass: #Parent
   superclass: #{Core.Object}
   instanceVariableNames: 'foo '
```

Class Method

```
new
   ^super new initialize
```

Instance Methods

```
initialize
   foo :=6.

foo
   ^foo
```

Initialization of Subclass

How to initialize bar?

```
Smalltalk.Core defineClass: #Child
   superclass: #{Core.Parent}
   instanceVariableNames: 'bar '
```

Bad Idea 1 – Use Same pattern

```
Child class>>new
   ^super new initialize

Child>>initialize
   bar := 2.

Child>>bar
   ^bar
```

Why bad?

Does not work!

```
| test |
test := Child new.
test foo “returns nil”
```

initialize is called twice

Child class>>new is not needed

Child class inherits an identical method

Bad Idea 2 – Subclass initializes Parent Variable

```
Child>>initialize
   bar := 2.
   foo := 6.
```

Why Bad?

Child class now involved in private affairs of the Parent

Changes to the Parent instance variables require changing Child

Solution

```
Parent class>>new
   ^super new initialize

Parent>>initialize
   foo :=6.

Parent>>foo
   ^foo
```

```
Child>>initialize
   super initialize
   bar := 2.

Child>>bar
   ^bar
```

Class Methods that Create Instances

Smalltalk does not have constructors like C++/Java

Use class methods to create instances

Place these class methods in "instance creation" category

Initial State of Instances

Create objects in some well-formed state

Class creation methods should:

- Have parameters for initial values of instance variables or
- Set default values for instance variables

Provide an instance method that:

- Sets the initial values of instance variables
- Place method in "initialize" or "initialize - release" category
- Use the name setVariable1: value variable2: ...

Disabling new

Point new

Does not work

Point x: 1 y: 12

This works

Point class>>new

^self shouldNotImplement

Implementers wanted users to specify initial value of a point

Actually the method is in the parent class of Point.

Class Instance Variables

A class has one instance of a class instance variable

Each subclass has a different instance

Accessible by

- Class methods of the class
- Class methods of subclasses

Example

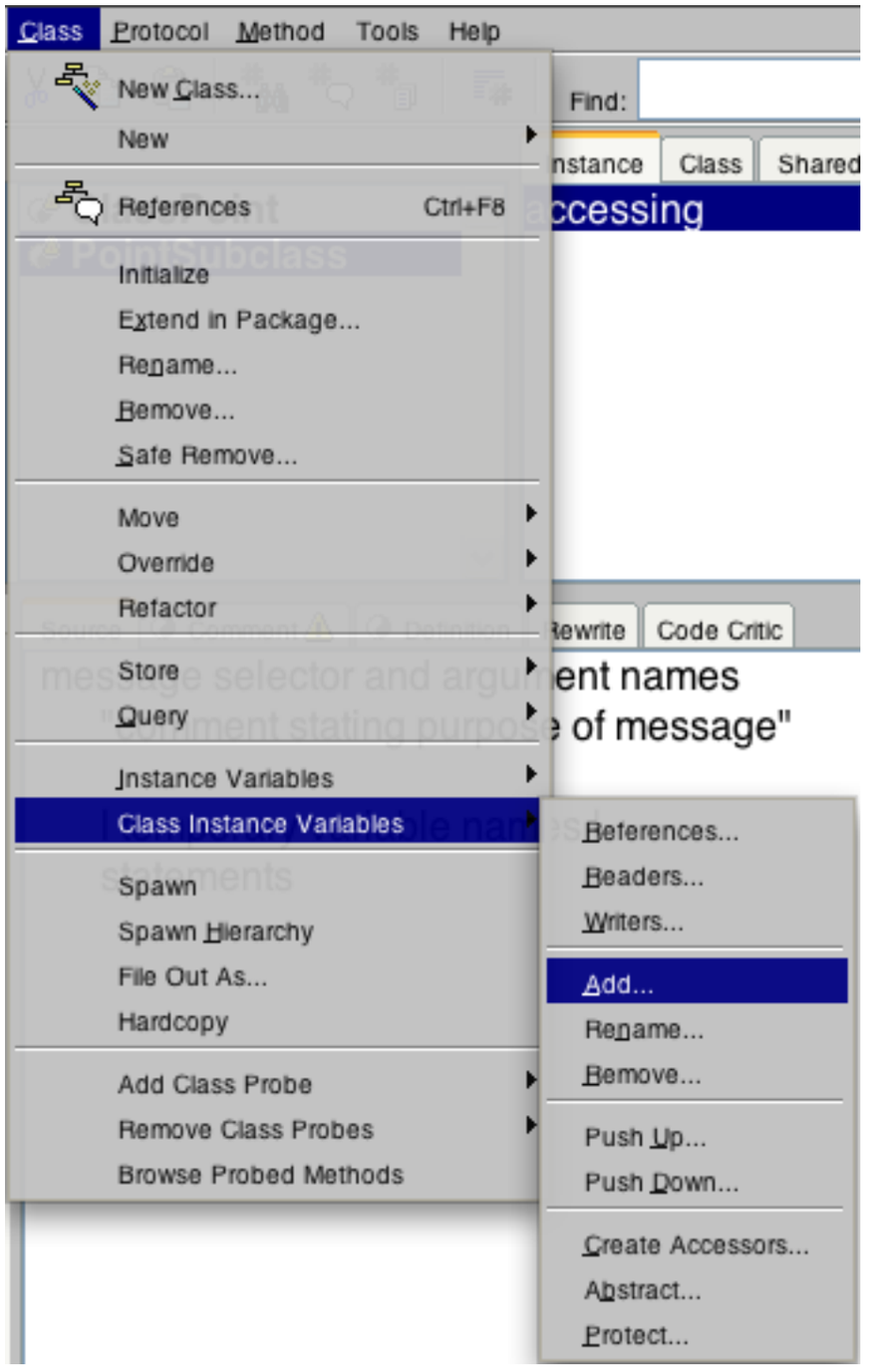
```
Smalltalk.Core defineClass: #ClassInstanceVariableExample
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: 'test '
    imports: ''
    category: 'As yet unclassified'
```

Adding/Removing Class Instance Variables

Method 1

Edit the class definition directly

Method 2



Example

```
Smalltalk.Core defineClass: #Parent
   superclass: #{Core.Object}
   classInstanceVariableNames: 'test '

Parent class>>test
   test isNil ifTrue:[ test := 0].
   test := test + 1.
   ^test
```

```
Transcript
    print: Parent test;
    cr;
    print: Parent test;
    cr;
    print: Child test;
    flush
```

```
1
2
1
```

```
Smalltalk.Core defineClass: #Child
   superclass: #{Core.Parent}
   classInstanceVariableNames: ''
```

Lazy Initialization

```
Parent class>>test
   test isNil ifTrue:[ test := 0].
   test := test + 1.
   ^test
```

More on Blocks

```
Integer>>foo
    | x block |
    x := 10.
    block := [self + x].
    ^block
```

In workspace

```
| x fooBlock result |
x := 5.
fooBlock := 3 foo.
result := fooBlock value
```

what is the value in result?

Indexed Instance Variable

Provides slots in objects for array like indexing

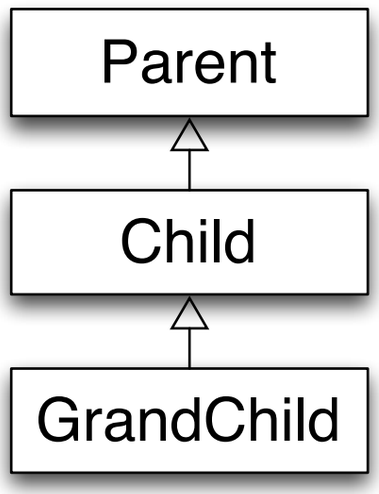
Used for Arrays

I have never added indexed instance variables

I have always used existing collection classes

Polymorphism

Polymorphism




```
Parent>>name
   ^'Parent'

Parent>>age
      ^50

Parent>>total
      ^self name size + self age
```

Which method is called

aPerson := ??? new.

aPerson name

aPerson age

aPerson total

```
GrandChild>>name
   ^'GrandChild'

GrandChild>>age
      ^super age - 18
```

```
Child>>name
   ^'Child'

Child>>age
   ^super age - 19
```

when ??? is

Parent

Child

GrandChild

Template Method

```
Parent>>total
    ^self name size + self age
```

Parent method (total) defines algorithm using methods

Subclasses implement those methods

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 (270) for all objects

Determines behavior for all objects

printString

Returns a string representation of the receiver
Similar to toString in Java

5 printString	'5'
$a printString	'$a "16r0061"'
#(1 2 3) printString	'#(1 2 3)'
a:= ClassPoint new. a printString	'a ClassPoint'

Implementing printString for ClassPoint

```
ClassPoint>>printOn: aStream
        aStream
                nextPut: $(;
                print: x ;
                nextPut: $,;
                space;
                print: y;
                nextPut: $).
```

a:= ClassPoint new. a x: 4; y: -1. a printString	'(4, -1)'

Where is printStream?

Object uses Template Method

```
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
```

printString is a template method

You just implement printOn: and printString will work

Thursday, September 12, 13

Remember "do it once and only once"? Template method is one way of achieving that. Since the standard way of creating a string representation is to create a WriteStream (don't worry about what that is), write to the stream and then return the contents of the stream we put the common code in Object and just implement the part specific to our class. We could implement the entire logic in each class, but that would not be "do it once and only once".

Useful WriteStream methods

```
ClassPoint>>printOn: aStream
    aStream
        nextPut: $(;
        print: x ;
        nextPut: $,;
        space;
        print: y;
        nextPut: $).
```

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

isInteger

'cat' isInteger	false
$5 isInteger	false
4 isInteger	true
4.5 isInteger	false

```
Object>>isInteger

    ^false
```

```
Integer>>isInteger

    ^true
```

Replace case (if) with Polymorphism

```
Object>>isInteger
    ^self class = Integer
```

verses

```
Object>>isInteger

    ^false
```

```
Integer>>isInteger

    ^true
```

Polymorphism makes change easier

What if we add a new type of Integer?

```
Object>>isInteger
    self class = Integer
        ifTrue: [^true].
    self class = CS535Integer
        ifTrue: [^true].
    ^false
```

verses

```
Object>>isInteger

    ^false
```

```
Integer>>isInteger

    ^true
```

```
CS535Integer>>isInteger

    ^true
```


When we add a new type of Integer class we just have to make sure it returns the correct result. We do not have to find and change all the if or case statements that check to see if something is an integer.

Avoid checking the type of an Object

Heuristic 5.12
Explicit case analysis on the type of an object is usually an error.
The designer should use polymorphism in most of these cases

```
Transcript show: anObject printString

        verses


anObject isInteger
    ifTrue: [Transcript show: anObject printString].
anObject isString
    ifTrue: [Transcript show: anObject].
anObject isArray
    ifTrue: [anObject do: [:element | Transcript show: element].
```

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

Defining =

If you define = also define hash

```
ClassPoint>>= anObject
        anObject isPoint ifFalse:[^false].
        ^self x = anObject x and: [self y = anObject y]

ClassPoint>>hash

        ^x hash hashMultiply bitXor: y hash
```

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

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

Three GUI Interfaces for viewing Test Results

TestRunner

- Already loaded in Image

Browser SUnit Extensions

- Easier to run individual tests
- Needs to be loaded

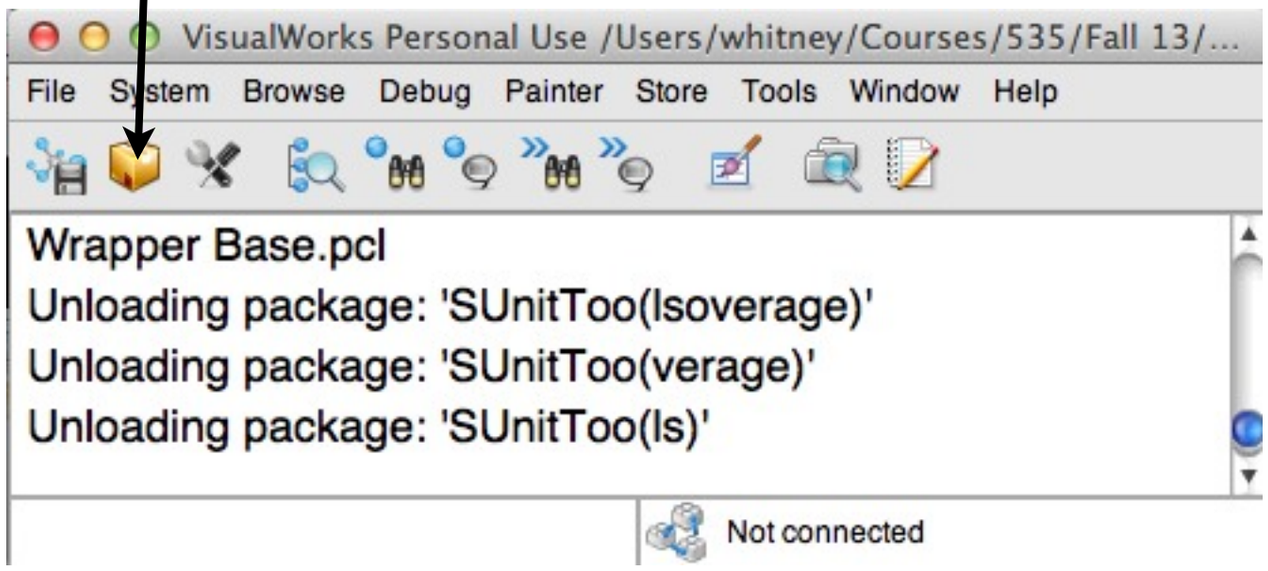
SUnitToo

- Automates more actions

Loading SUnitToo

Step 1

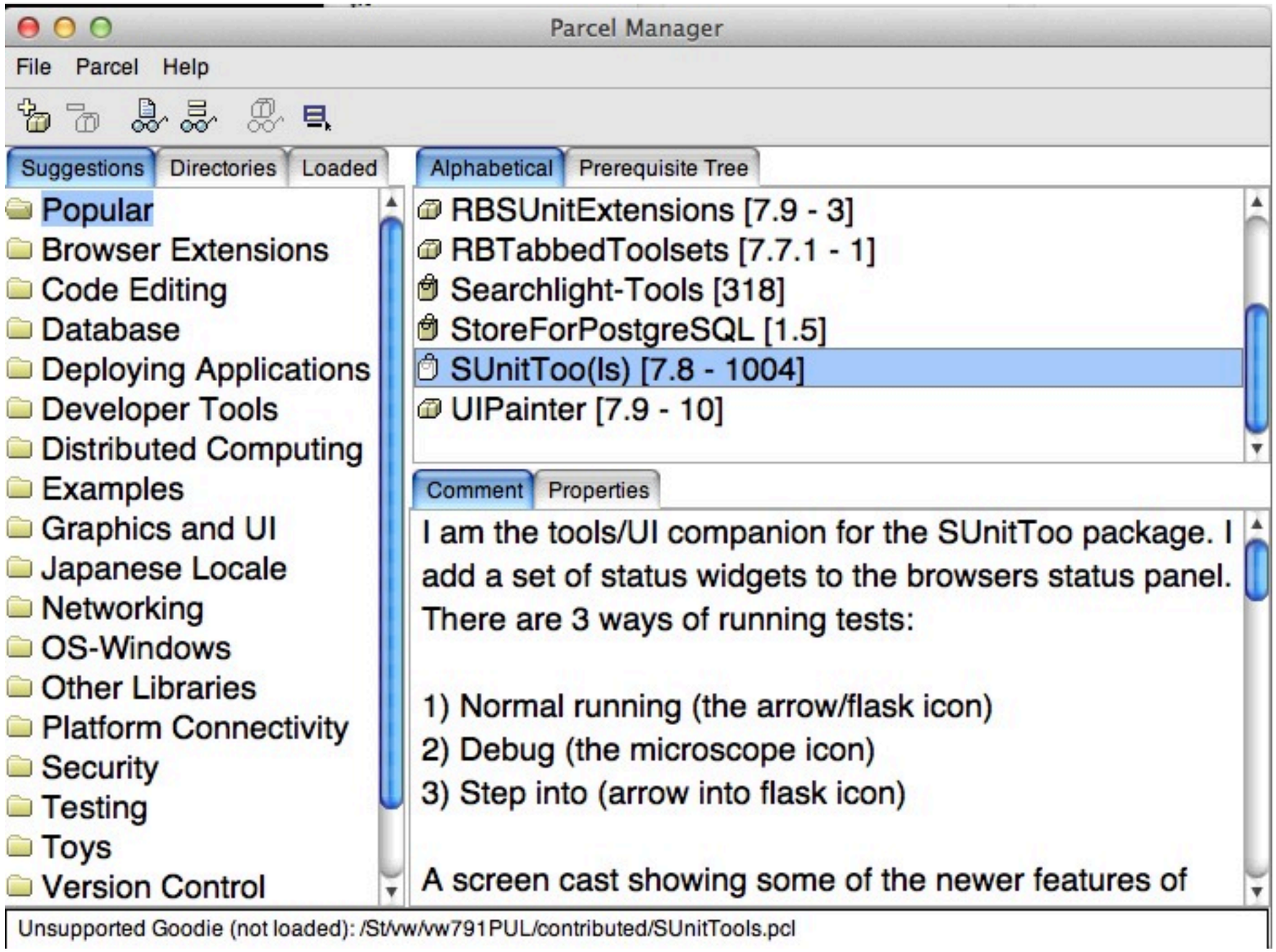
In Launcher window



Open the parcel manager

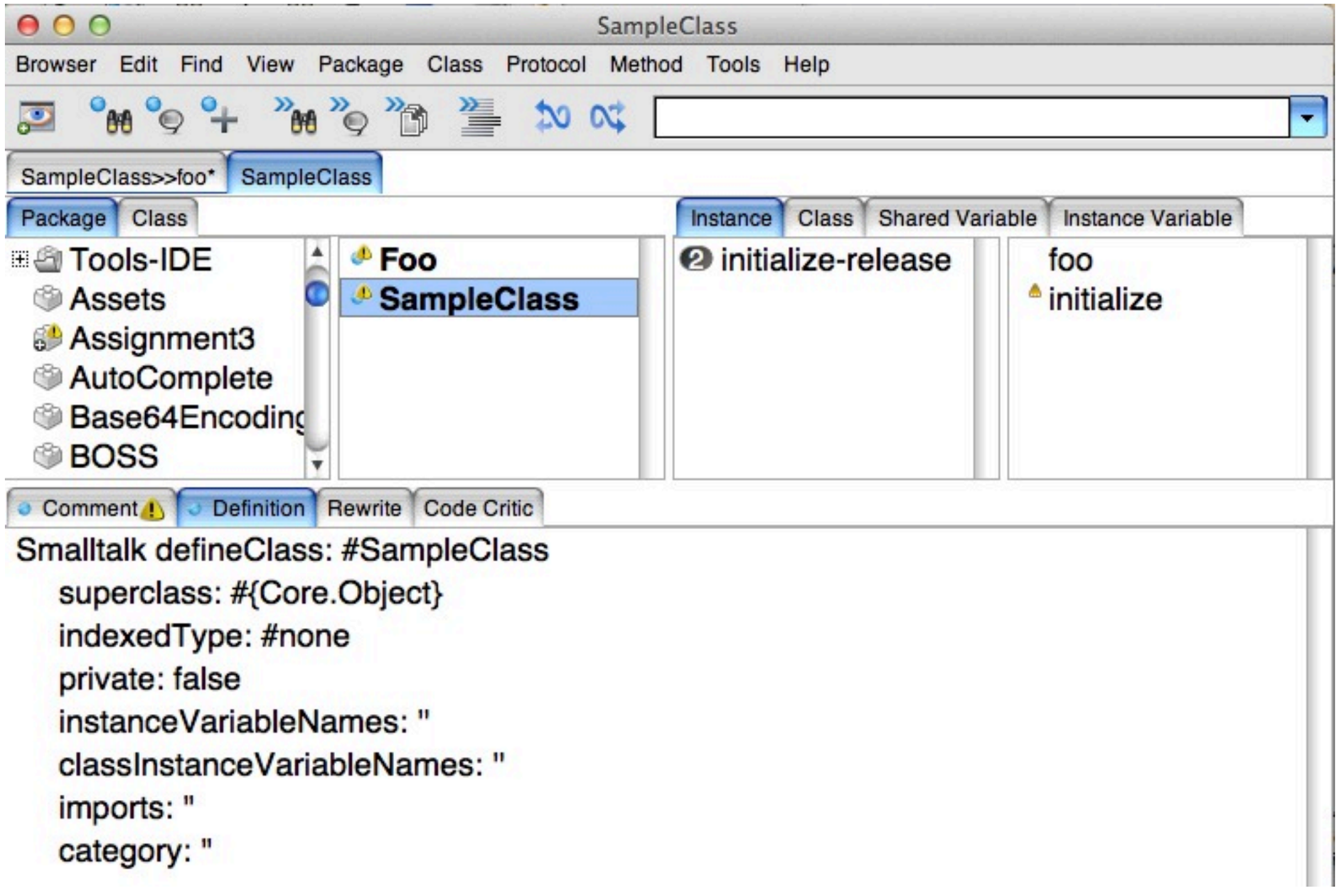
Loading SUnitToo

Step 2



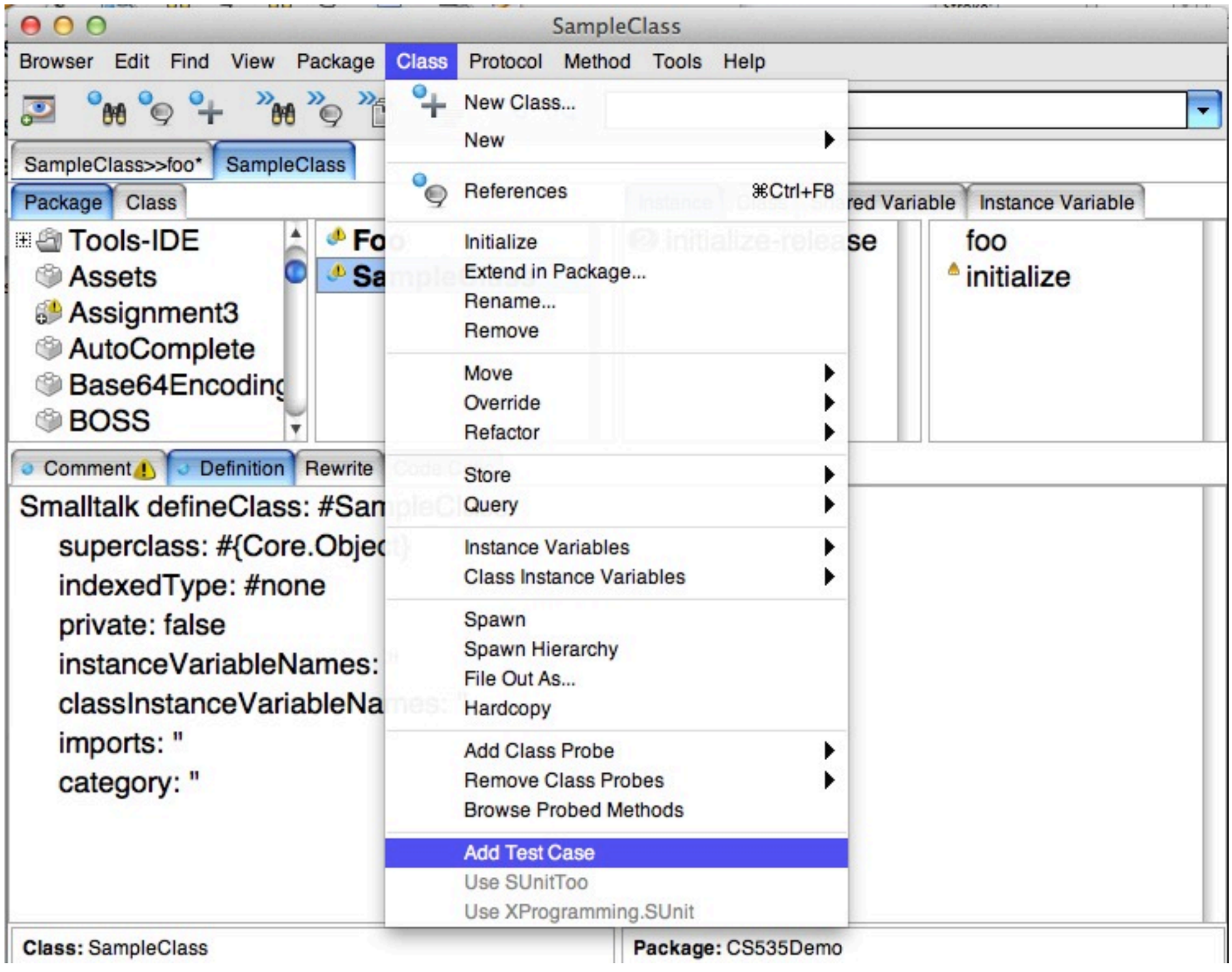
Creating a Test Class

Select the class you want to test



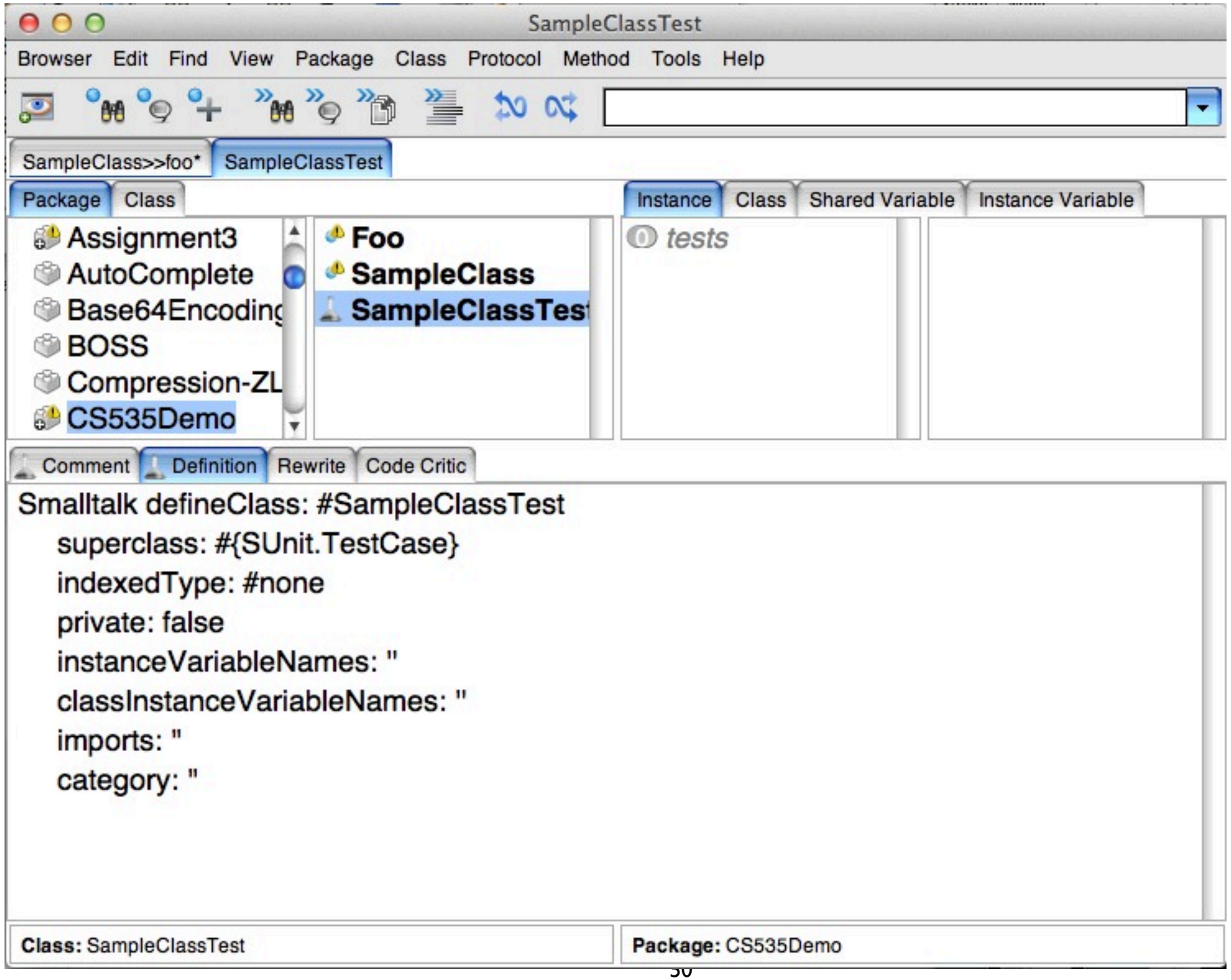
Creating a Test Class

Select "Add Test Case" from Class menu

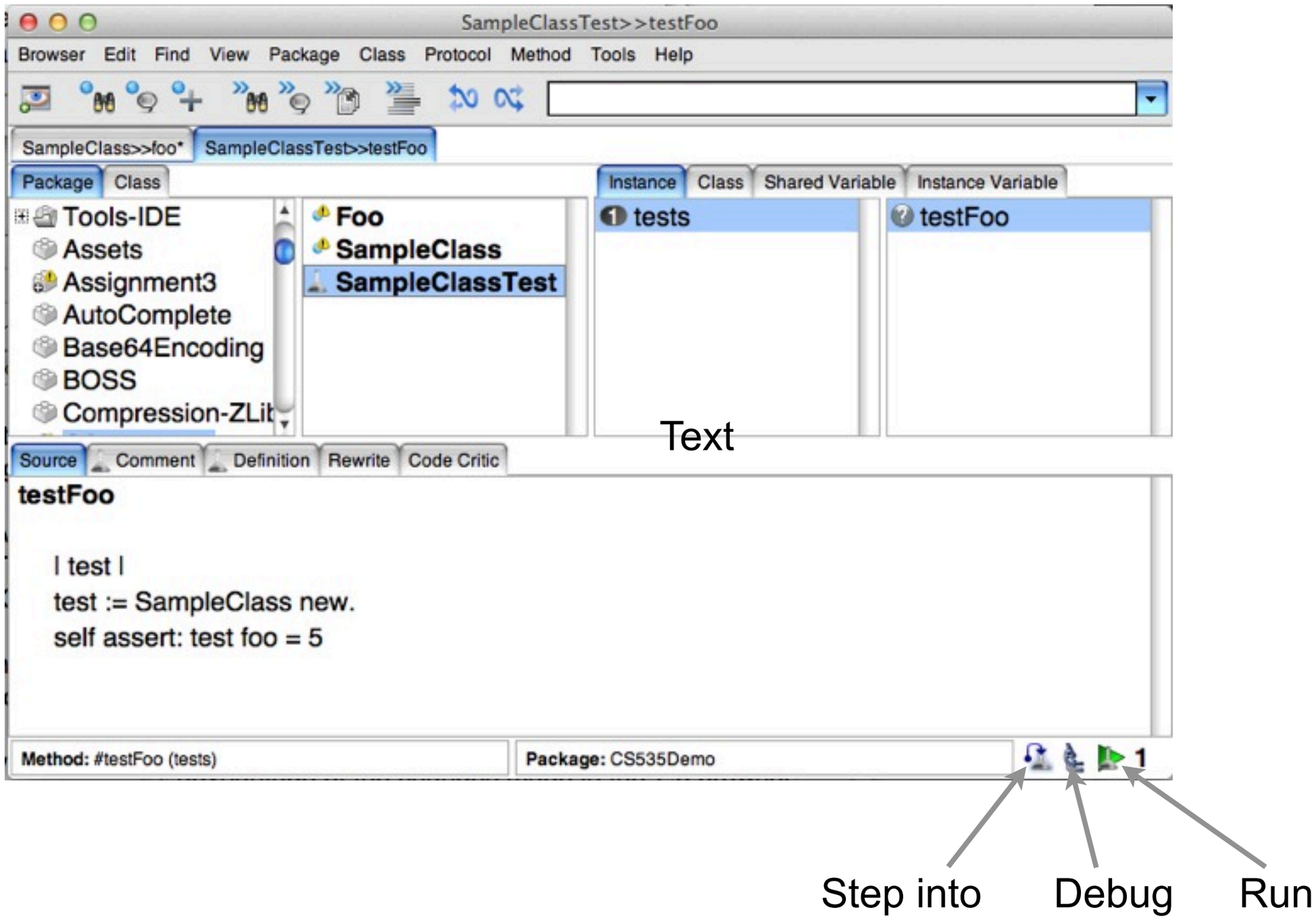


Creating a Test Class

Now can add test method to the class



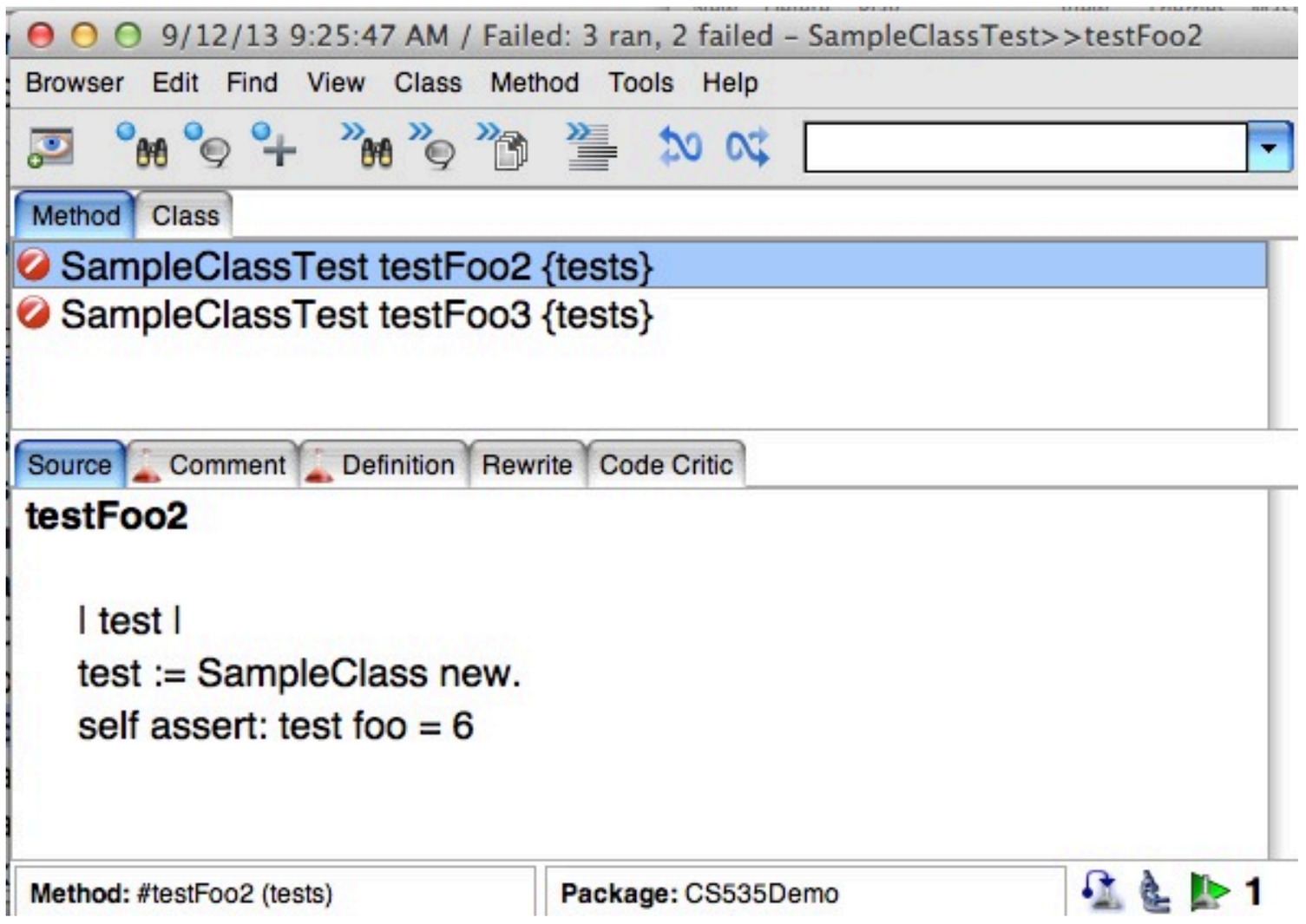
How to Run the Tests



Result of Running Test

Result of Running Multiple Tests

Window of listing failed tests



Sample Test Case

```
ClassPointTest>>testX

    | aPoint |
    aPoint := ClassPoint new.
    self
        assert: aPoint x = 0;
        assert: aPoint y = 0.
    aPoint x: 5.
    self assert: aPoint x = 5.
    self deny: aPoint x = 10.
```

ClassPointTest is subclass of SUnit.TestCase

Framework runs methods whose name start with test

This is a silly test. We don't need to test an setter method. But this is just an example of a test method.

Important Methods of TestCase

assert: aBooleanExpression
deny: aBooleanExpression
should: [aBooleanExpression]
should: [aBooleanExpression] raise: AnExceptionClass
shouldnt: [aBooleanExpression]
shouldnt: [aBooleanExpression] raise: AnExceptionClass
signalFailure: aString

Another Example

```
testZeroDivide
    self
        should: [1/0]
        raise: ZeroDivide.

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

```
self should: [2 = 1 + 1]
```

setUp & tearDown

setUp

Called before running each test method

tearDown

Called after running each test method

Used to set up and tear down items for tests

files

database connections

objects needed for test methods

Example

```
ClassPointTest>>setUp

    largePoint := ClassPoint new.
    largePoint
        x: 100;
        y: 100
```

```
ClassPointTest>>testLarge
    self assert: largePoint x = 100.
    largePoint x: 10.
    self assert: largePoint x = 10.
```