

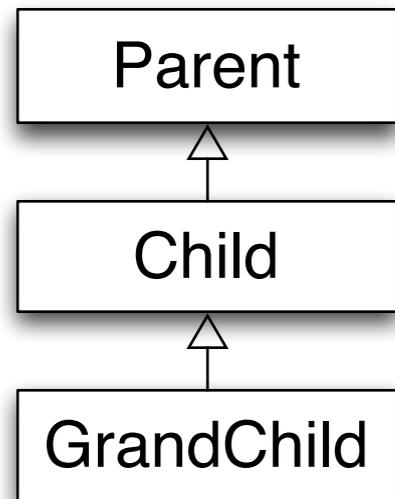
CS 535 Object-Oriented Programming & Design
Fall Semester, 2010
Doc 6 Polymorphism, Object, Testing
Sept 16 2010

Copyright ©, All rights reserved. 2010 SDSU & Roger Whitney, 5500
Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (<http://www.opencontent.org/openpub/>) license defines the copyright on this document.

References

Object-Oriented Design Heuristics, p 98

Polymorphism



```
Parent>>name  
  ^'Parent'  
  
Parent>>age  
  ^50  
  
Parent>>total  
  ^self name size + self age
```

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

```
Child>>name  
  ^'Child'  
  
Child>>age  
  ^super age - 15
```

Which method is called

aPerson := ??? new.

aPerson name

aPerson age

aPerson total

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

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
```

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

$A \sim= B$

Means $(A = B)$ not

$A \sim\sim B$

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

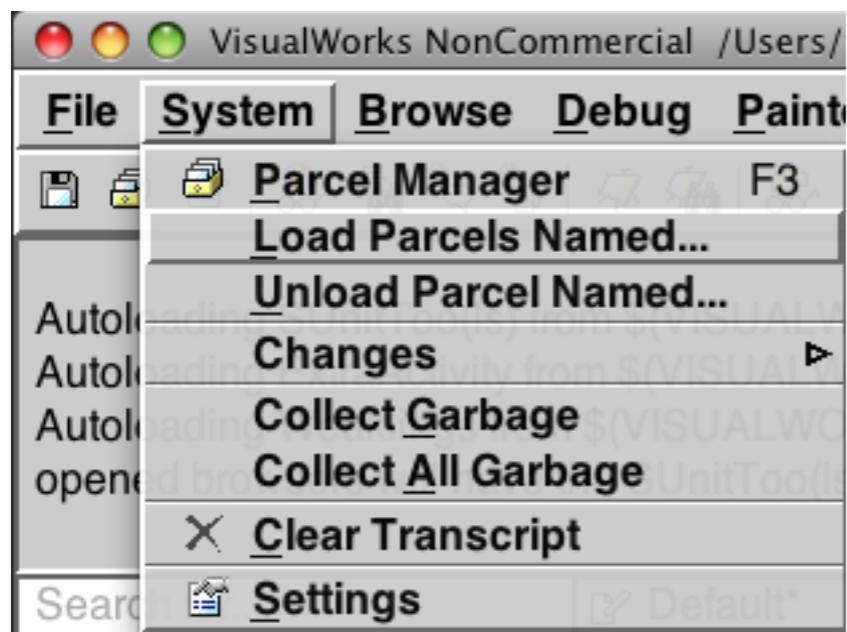
SUnitToo

Automates more actions

Loading SUnitToo

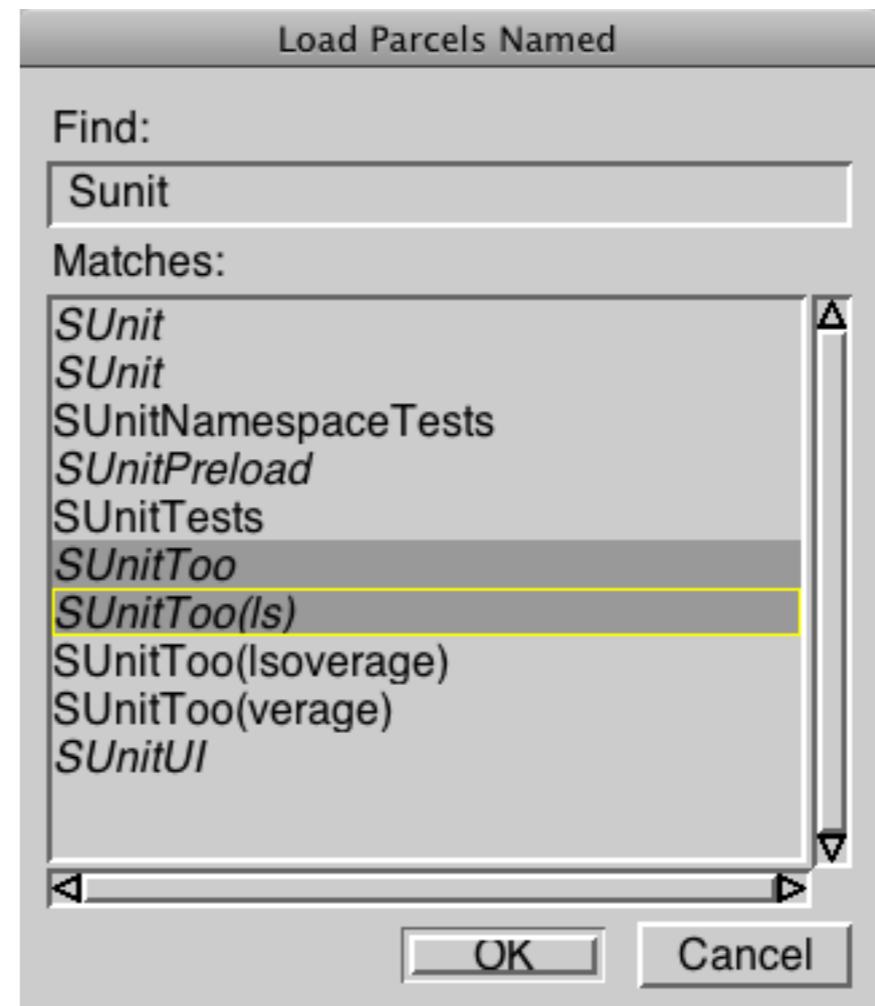
Step 1

In Launcher window



Website has a screencast of loading and using SUnitToo

Step 2



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

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.
```