References

Object-Oriented Design Heuristics, p 98
Polymorphism

Which method is called

aPerson := ??? new.
aPerson name
aPerson age
aPerson total

when ??? is
Parent
Child
GrandChild

Parent>>name
  ^'Parent'

Parent>>age
  ^50

Parent>>total
  ^self name size + self age

Child>>name
  ^'Child'

Child>>age
  ^super age - 15

GrandChild>>name
  ^'GrandChild'

GrandChild>>age
  ^super age - 18

GrandChild>>name
  ^'GrandChild'

GrandChild>>age
  ^super age - 18
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

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 printString</td>
<td>'5'</td>
</tr>
<tr>
<td>$a printString</td>
<td>'$a &quot;16r0061&quot;'</td>
</tr>
<tr>
<td>#( 1 2 3) printString</td>
<td>'#(1 2 3)'</td>
</tr>
<tr>
<td>a:= ClassPoint new. a printString</td>
<td>'a ClassPoint'</td>
</tr>
</tbody>
</table>
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

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

<table>
<thead>
<tr>
<th>Value</th>
<th>isInteger</th>
</tr>
</thead>
<tbody>
<tr>
<td>'cat'</td>
<td>false</td>
</tr>
<tr>
<td>$5</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>4.5</td>
<td>false</td>
</tr>
</tbody>
</table>

#### Code Examples

**Object>>isInteger**

```plaintext
^false
```

**Integer>>isInteger**

```plaintext
^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?

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

<table>
<thead>
<tr>
<th>Unit Tests</th>
<th>Functional Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests individual code segments</td>
<td>Test functionality of an application</td>
</tr>
</tbody>
</table>
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

![Image of Load Parcels Named window]
Sample Test Case

ClassPointTest>>testX

<table>
<thead>
<tr>
<th>aPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>aPoint := ClassPoint new.</td>
</tr>
<tr>
<td>self</td>
</tr>
<tr>
<td>assert: aPoint x = 0;</td>
</tr>
<tr>
<td>assert: aPoint y = 0.</td>
</tr>
<tr>
<td>aPoint x: 5.</td>
</tr>
<tr>
<td>self assert: aPoint x = 5.</td>
</tr>
<tr>
<td>self deny: aPoint x = 10.</td>
</tr>
</tbody>
</table>

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
    shouldn't: [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.