# Teacher's Notes

## I. Approach of Book

(See the Introduction of the book for now.)

## II. Suggestions on Teaching

I've been teaching a course like this for five years now. Below are some of my suggestions for things to do in a class like this.

### *1.1 In-Class Design*

As we get into design activities (e.g., Chapter 4 in the book), I will put up a design problem and ask students to work in small groups to do CRC Card analysis and a UML class diagram in a short period, 15-20 minutes. I ask some groups to put their class diagrams on the board or on transparencies. The results have always been very useful: Students typically have fairly well-formed (though necessarily incomplete) designs, each group taking a different tact at the problem. The discussion allows us to explore a design space, with each group defending their own decisions….

### *1.2 Live OOA/D/P*

At least once each term, I take a problem from analysis through design to program in a single 90 minute session. Of course, I never finish, but the exercise helps the students to see how all the pieces fit together…

## III. Project Suggestions

Below are suggestions for

### *1.3 Chapter 1*

1. There are other object-oriented programming languages, such as Eiffel, Modula-3, and Python (http://www.python.org). What are their stories? Were they influenced by the same things as Smalltalk, C++, and Java?

### *1.4 Chapter 2*

1. Basic data structures and algorithms can be explored after completing Chapter 2. Implementing some of these would help students to apply their knowledge of general computer science to this new language. Some examples include:

- Extend the Tree exercise to support (a) insertion where some ordering to the tree is maintained and (b) rotating and balancing of trees.
-

## IV. Answers to Exercises

### 1.5 Chapter 1

2. *Objects as cells* emphasizes the key notions of *encapsulation* and *messaging*. Encapsulation says that each object maintains its own data and contains its own behaviors. Messaging says that interactions between objects occur in well-defined mechanisms. *Software as simulation* emphasizes the key notion of an object-oriented program *modeling* the real world. *Objects as little computers* gets back to the *messaging* idea.
3. Inheritance isn’t really that critical of a feature. It does allow for extending the system easily, but in reality, most object-oriented software makes more use of *aggregation* (containing objects within objects) than inheritance. Cells do not really inherit from one another in any physical sense.
4. Biological cells do have *kinds*. Kidney cells are different than skin cells. Cells get manufactured, but not from one factory, so the basic notion of a class as an object factory fails. But a class as blueprint for a kind of object, defining its behavior and data, does exist metaphorically in biological cells.
5. Many possible solutions exist for this question. The main point is to argue from the key features in the chapter.

### 1.6 Chapter 2

1. **whileTrue**: is defined on **BlockContexts**. Integer addition is implemented is implemented in the class **Integer**.
2. The two kinds of statements that break the rule so far are assignment and return. **aVariable := 3** is not a message send. This means that it’s not possible to override assignment. **^returnValue** is not a message send, either. In some sense, it’s the end of a message send. **^** might be read as “give this back to the object who asked for me.”
3.

(a)

```
aString := 'squeak'.

nuString := ''.

aString do: [:character |

        (character isVowel)
```

```
                    ifTrue: [nuString := nuString,'-']
                    ifFalse: [nuString := nuString,character asString]].
        ^nuString


        (b)
        aSum := 0.
        aCount := 0.
        #(12 32 52 61) do:
          [:number | aSum := aSum + number.
                         aCount := aCount + 1].
        ^(aSum / aCount) asFloat
```

4. Create FriendlyMuppet class as a subclass of Muppet, then simply create the method:

```
greet
        ^’Well, howdy!’
```

5. Same as above, but with a different **greeting** (or **greet**) in each.
6. (A kinetic exercise.)
7. Surprisingly, it’s not Object. You can find it by using a System Browser to look at the Class side of Object.  You won’t find **new** there.  Now, choose *spawn protocol*. You’ll find that new is actually implemented by **Behavior**.

   How does that happen?  Choose *Hierarchy* to see the hierarchy of the class side of **Object**.  The parent of the class side of **Object** is the class **Class**, whose parent is **ClassDescription**, whose parent is **Behavior**.

```
Object ()
        Behavior ('superclass' 'methodDict' 'format' )
                ClassDescription ('instanceVariables' 'organization' )
                        Class ('subclasses' 'name' 'classPool' 'sharedPools' )


                                Object class ()
```

8. Filed out code follows:

```
'From Squeak 2.5 of August 6, 1999 on 30 August 1999 at 2:02:03 pm'!


!Array methodsFor: 'enumerating' stamp: 'mjg 8/30/1999 14:01'!
```

```
switchOn: argument
        "Provide a case-like statement:
#(      ('a' [Transcript show: 'An a was input'])
        ('b' [Transcript show: 'A b was input']))
        switchOn: 'a'

        The tricky part is that a literal array returns the block as individual
        string elements of the array.
"

        | firstCouplet blockToEvaluate |
        firstCouplet := self at: 1.
        (argument = (firstCouplet at: 1))
                ifTrue: ["Now we have to translate the rest of the block"
                                "We ask the compiler to evaluate the string
formed by concatenating the rest"
                                blockToEvaluate := Compiler evaluate:
                                        (firstCouplet copyFrom: 2 to: firstCouplet
size).
                                blockToEvaluate value.]
                ifFalse: [(self allButFirst) switchOn: argument]
! !
```

9. Filed out code follows

```
'From Squeak 2.5 of August 6, 1999 on 9 September 1999 at 4:47:20 pm'!
Object subclass: #Tree
        instanceVariableNames: 'info left right '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'TreeProject'!

!Tree methodsFor: 'testing' stamp: 'mjg 9/9/1999 16:47'!
isLeftTree
        "Is my left side a tree?"
        ^(left isKindOf: Tree)
! !
```

```
!Tree methodsFor: 'testing' stamp: 'mjg 9/9/1999 16:47'!
isRightTree
        "Is my right side a tree?"
        ^(right isKindOf: Tree)
! !


!Tree methodsFor: 'traversal' stamp: 'mjg 9/9/1999 16:43'!
inorder
        "If leftside is a tree, traverse it.
        Traverse yourself.
        If rightside is a tree, traverse it."
        | result |
        result := OrderedCollection new.
        self isLeftTree ifTrue: [result := result , left inorder].
        result add: info.
        self isRightTree ifTrue: [result := result , right inorder].
        ^result

! !


!Tree methodsFor: 'accessors' stamp: 'mjg 9/9/1999 16:42'!
info
        info isNil ifTrue: [^''] ifFalse: [^info].

! !

!Tree methodsFor: 'accessors' stamp: 'mjg 9/9/1999 16:32'!
info: something
        info _ something.
! !

!Tree methodsFor: 'accessors' stamp: 'mjg 9/9/1999 16:33'!
left
```

```
		^left
	! !

	!Tree methodsFor: 'accessors' stamp: 'mjg 9/9/1999 16:32'!
	left: aTree
		left _ aTree.
	! !

	!Tree methodsFor: 'accessors' stamp: 'mjg 9/9/1999 16:33'!
	right
		^right
	! !

	!Tree methodsFor: 'accessors' stamp: 'mjg 9/9/1999 16:33'!
	right: aTree
		right _ aTree
	! !
```

## *1.7 Chapter 3*

1. Yes, having **super new initialize** in two consecutive subclasses *will* work, however it is inefficient. **initialize** will actually get called twice. Recall that **super new** always returns an instance of the class that originally got sent the message. **initialize** will be sent to that same instance twice: Once in **Box** and once in **NamedBox**.
2. Having an object send **new** to itself inside a method named **new** is perhaps the tightest infinite loop available in Squeak.
3. Adding a Delay makes it easier to see: **30 timesRepeat: [jane turn: 12. joe turn: 10. (Delay forSeconds: 0.5) wait].**
4. In the first case, **super new** doesn't change who **self** is. **Box new** would return an instance of Box. In the case of **draw** and **undraw**, **Box draw** calls a *class* method **draw**, which doesn't exist.
5. This will be easier in Section 6 of Chapter 3, but the below method works. **TextStyle default** returns a bunch of things associated with the current text style, including an array of fonts.

```
	draw
		| p |
        super draw.
        p _ Pen new.
        p turn: 90. "Always horizontal."
        p place: position. "Go to the position without drawing a line"
        p print: (name isNil ifTrue: ['Unnamed'] ifFalse: [name]) "Deal with nil
names"
                withFont: (TextStyle default fontArray at: 1).
        "self drawNameColor: (Color black)."
```