Smalltalk

Object-oriented programming is a fifth-generation style which emphasizes simulation of the behavior of objects. Smalltalk was the first pure object-oriented (oo) language; Java is the most popular oo language currently.

Alan Kay lead the development of Smalltalk, following his intuition in the late 60s that personal computing (which did not exist at the time) would not succeed without a more friendly programming language. Recall that Algol and other second generation languages of the time were still closely tied to mainframe computing, and thus in the domain of computer specialists. Smalltalk was the programming language for Kay's seminal idea of the laptop computer. The developmental philosophy was:

> *Simple things should be simple, complex things should be possible.*

Smalltalk evolved from Simula (Nygaard), a simulation language, and LOGO (Papert), a very simple pedagogical language used to teach programming to children 8-12 years old. The design was also heavily influenced by research in developmental psychology (Dewey, Montesorri, Piaget, Bruner), and by Sutherland's Sketchpad, the first prototype VR system (late 60s). It was also conceived as part of an integrated graphical development environment which was developed at Xerox PARC in the early 70s, and lead eventually to the MacIntosh WIMP interface (Windows, Icons, Menus, Pointing device).

These are the characteristics which were seen to compose a user-friendly language:

object-oriented simulation
graphical interface
interactive (interpreted)
programming through dialog
integrated development environment

Objects, Messages and Methods

The programming unit in oo languages is the object. Objects have both state and behavior. Behavior is triggered by sending messages to objects. Repetitive behavior is simplified by control structures. The functions which define object behavior are called methods.

It is important to realize the oo programming on a single processor machine is simply a reorganization of code at the user level. Messages are procedure invocations. Objects are data structures. Methods are functions. Consider finding the area of a rectangle:

`rectangle[area-of]`	object-oriented
`area-of[rectangle]`	functional

Swapping function and argument turns object-oriented into functional programming.

An abstract object, one with parameters rather than bound values, is a class. Instances are created by instantiating a class description with values. In functional terms, a class is a set of operators, an instance is applicative expansion of an operator. Class inheritance is normal expansion of an operator.

The data values inside an object represent the properties and relations in which that object participates. The methods inside an object simulate the behavior of the corresponding semantic object. Objects then hold the state of the computation. In general, each object acts as an autonomous agent that is responsible for its own behavior, and is not responsible for the behavior of any other object. The memory organization of a computation is organized around objects; the information usually carried in function activation frames is stored within each object.

Classes

A class definition specifies all the properties and behaviors which are common to all instances of that class. Decomposing a problem into classes is analogous to functional decomposition in functional languages.

The biological analogy is that classes are genotypes while instances are phenotypes. Classes contain the organization of an entity, those characteristics which are present in all individuals of that type. Instances contain the structure of an entity, those characteristics which differentiate one individual from another.

Classes (abstract objects) also have methods. Class methods are used to create new instances. When a `new` message is sent to a class, the class constructs a copy of itself and binds the class parameters to the instance values conveyed by the `new` message.

A class hierarchy develops when the class abstraction principle is applied to classes themselves. (This is simply saying that operators can be composed without instantiation.) The class hierarchy is an organizational technique at the interface. Unfortunately, semantic objects are not orthogonal, class decomposition (like function decomposition) can be achieved in many equally valid ways, each way being appropriate for some behaviors, and blind to other behaviors.

For example, consider the class `mammal`. A biological taxonomy places mammals in a kingdom-phylum-species hierarchy, comparing mammals to other living creatures. A pragmatic classification, on the other hand, may classify mammals by their utility, as pets, beasts of burden, pests, sources of food, etc. A geological classification may classify mammals by their ecological zone, temperate, island, arctic, etc. Each of these classification schemes is orthogonal to the others, however a particular mammal gets classified by each differently.

The original solution was multiple inheritance, objects could inherit from several classes. This idea introduces as many problems as it solves. For instance, inherited methods from two classes may be contradictory. Inheritance from many classes builds objects which are far larger than any class they may inherit from. And from an coding perspective, multiple inheritance is extremely difficult to implement, essentially doubling the size of an oo compiler. Multiple inheritance, from a functional perspective, is attempting to insert control logic into lambda

calculus, thus undermining the semantics of the model. For these reason, the Java language does not provide the option of multiple inheritance.

As an example of a class hierarchy, a partial listing of the built-in classes in Smalltalk is presented on the next page.

- Object
  - Magnitude
    - Character
    - Date
    - Time
    - Number
      - Float
      - Fraction
      - Integer
        - LargePositiveInteger
        - LargeNegativeInteger
        - SmallInteger
  - Collection
    - SequenceableCollection
      - LinkedList
      - ArrayedCollection
        - Array
        - Bitmap
        - String
      - Interval
      - OrderedCollection
    - Bag
    - MappedCollection
    - Set
  - DisplayObject
    - DisplayMedium
      - Form
        - Cursor
        - DisplayScreen
    - InfiniteForm
    - OpaqueForm
    - Path
      - Arc
      - Curve
      - Line
      - Spline
  - Behavior

In this class hierarchy we see a structured decomposition of concepts from mathematics (e.g. collections), programming (e.g. float numbers), and graphics (e.g. DisplayObject). The decomposition is rather ad hoc (e.g. bags are a mathematical extension of sets but both are listed at the same level of abstraction), and is quite interface dependent (e.g. DisplayObejcts assume a WIMP interface).

Overloading, Information Hiding and Extensibility

Overloading refers to using the same token to trigger different behaviors in different objects. Since the organizational structure isolates bindings and methods within specific objects and classes, the issues of scoping and binding regimes are not troublesome. That is, oo techniques remove the machine dependencies associated with names in procedural languages. Again from the functional perspective, this is simply that functional organization does not require variables.

Classes serve as abstract data types, therefore they provide an abstraction barrier between model and implementation. These ideas are the same as those which motivated packages, generic packages, and tasks in Ada. Classes provide enforced modularity.

Due to class inheritance, Smalltalk is flexible and extensible. The programmer can define a new class which inherits from existing classes, thus significantly reducing both programming effort and possibility of errors.

Message Sending and Protocols

In procedural languages, programs are active and data structures are passive. This reflects a hardware architecture model in which memory is used to passively store, while computational circuitry is used to modify bits and words. In oo, objects are active, they respond to communications from other objects, modify themselves, and send messages intended to communicate with and change other objects.

The set of messages a particular object responds to is called its *protocol*. It is an error to send a message to an object which does not include that type of message in its protocol. Names in Smalltalk are *not typed*, any name can be associated with any object. Instead, protocols provide strong type checking, in that all valid messages are responded to by the receiving object. Since messages are sent at runtime, Smalltalk uses *dynamic*, as opposed to static, *type checking*. Dropped messages signal run-time errors which halt computation (without crashing the system), much like an interpreted language.

All objects are identified by a pointer, or *object reference*. Since there are no functions in an oo language, the cost of storing both variables and message invocations is identical. Activation frames are no longer a relevant concept. OO techniques allow algorithms to be factored out of the program without complication.

In a single processor system, messages are still procedure invocations. The major difference from procedural approaches is that instances of objects are constructed and initialized dynamically, at run-time, rather than statically in the object code.

There are three message formats in Smalltalk. For messages with no parameters, the name of the message serves as a keyword to trigger the message functionality. For messages with one or more parameters, keywords are again used to identify parameters, as in

```
Box grow: 100 color: green scrollbar: false
```

This approach however is awkward for numerical operators, for instance

```
x plus: 2
```

sends the plus message to the object `x`, with the parameter binding `2`. Smalltalk originally began as a pure oo language, even each number was an object. In a design concession, Smalltalk was changed to treat numerical computational more conventionally. So,

```
x + 2
```

is written instead, although this code still sends the + message to the object `x`.

Top Level

Smalltalk is *meta-circular*, its main loop is written in Smalltalk:

```
true whileTrue: [Display put: user run]
```

The `true` object is sent the `whileTrue` message which is bound to the object generated by sending the `Display` object a `put` message bound to `user`. Inside `user`, an object `userTask` responds to the `run` message by reading an expression, evaluating it, and returning the result to `Display`. `Run` is defined as:

```
run =def= Keyboard read eval print
```

Note that the Smalltalk style is similar to function invocation. The operational semantics of the language is to send the leading object the first message which follows. The object returns a different object which then responds to the next remaining message, and so on.

Above, the `Keyboard` object responds to the `read` message by prompting the user interface and returning the string typed by the user. This input string then responds to the message `eval` by calling the Smalltalk interpreter for evaluation. The resultant object responds to the message `print` by printing the result to output.

Concurrency

The autonomous nature of objects makes concurrency natural for oo languages. In Smalltalk, concurrency is achieved by having an object capable of processing a set of messages. Smalltalk uses the functional concept of *mapping*. To run several concurrent tasks, `map` the object over the tasks. This in turn is converted into time-sharing threads by the os.

```
concurrent-run =def= scheduler map: [...]
```