Educational Object Oriented Programming Language Design: (A Survey On Blue Language)

Hojjat Sheikhattar
Hojjat@users.sourceforge.net

Some design principles (some of them are barrowed from Blue language paper):

1. **Pure Object Oriented:** It means no non-class base code find in the language. It also indicates that, we are trying to omit any type which is not a class from the language.
2. **No Conceptual Redundancy:** It states that there should be one [it is better to say only one] well-defined mechanism to express each concept.
3. **Clean Concept:** It means that the concept – which is wanted to teach- be represented in the language in a way that directly reflects the theoretical model and is not compromised by secondary issues.
4. **Readability:** The language must itself help (in other words force!) the developer to write a readable code.

Requirements for an object oriented teaching language:

(extracted from “REQUIREMENTS FOR A FIRST YEAR OBJECT-ORIENTED TEACHING LANGUAGE”
Michael Klling, Bett Koch and John Rosenberg
Basser Department of Computer Science
University of Sydney, Australia)

1. The language should support clean, simple and well-defined concepts. This applies especially to the type system, which will have a major influence on the structure of the language. The basic concepts of object-oriented programming, such as information hiding, inheritance, type parameterisation and dynamic dispatch, should be supported in a consistent and easily understandable manner.

2. The language should exhibit "pure" object-orientation in the sense that object-oriented constructs are not an additional option amongst other possible structures, but are the basic abstraction used in programming.

3. It should avoid concepts that are likely to result in erroneous programs. In particular it should have a safe, statically checked (as far as possible) type system, no explicit pointers and no undetectable uninitialised variables.

4. The language should not include constructs that concern machine internals and have no semantic value. This includes, most importantly, dynamic storage

allocation. As a result the system must provide automatic garbage collection.
5. It should have a well defined, easily understandable execution model.

6. The language should have an easily readable, consistent syntax. Consistency and understandability are enhanced by ensuring that the same syntax is used for semantically similar constructs and that different syntax is used for other constructs.

7. The language itself should be small, clear and avoid redundancy in language constructs.

8. It should, as far as possible, ease the transition to other widely used languages, such as C.

9. It should provide support for correctness assurance, such as assertions, debug instructions, and pre and post conditions.

10. Finally, the language should have an easy-to-use development environment, including a debugger, so that the students can concentrate on learning programming concepts rather than the environment itself.

Some Good Ideas which is used in Blue Language (extracted form Blue Language Spec.) :

1. Manifest & Dynamic Classes:

Manifest classes are classes where all objects are known statically. The objects pre-exist with the definition of the class and do not have to be created. The manifest classes are Integer, Real, Boolean, String and Enumeration classes. The first four of those are predefined and all values are known to the Blue compiler. Enumeration classes are user defined. The definition of such a class consists of an enumeration of all existing objects of that class, simultaneously creating a named reference to each object.
The literal '2', for instance, is a reference to the unique integer object with the value 2. The code segment
a := 2
b := 2
assigns references *to the same object* to a and b. Only one integer object with the value *2* exists. This does not create two distinct objects.
All literals are constant references to objects of manifest classes.
Dynamic classes are classes where, rather than listing all objects, a creation method for objects is specified. Dynamic classes are arrays and user defined general classes.
With the definition of a dynamic class, no object is created automatically. The user has to execute an explicit *create* operation to create objects of these types. Care must be taken not to confuse this with pointer and non-pointer types in other languages. In Blue, **all variables hold references to object**s. An integer variable holds, when assigned a value, the *reference* to that integer

object. Thus the object model is simple: only references to objects exist. The difference between manifest and dynamic objects affects only the time and method of creation of the objects of the class, not the mechanism by which they are referenced.

2. Aliases:

All types in Blue are classes and all data are objects. This general rule simplifies the language design. There are, however, a number of data types for which it is convenient to use syntax other than the Blue object call to perform one of their operations. The reason for this can be:

• Another syntax is commonly used and is therefore more intuitive (e.g. *3 + 5* for integer addition, rather than *3.add (5)*).

• Another syntax simplifies use of elementary constructs which should be used by beginners before the underlying language concepts need to be understood, e.g.

print ("result=", 42)

instead of

output.write ("result=".concat (42.toString))

• Another syntax is more convenient (usually because it is shorter, see above).

For these reasons, several operations on the predefined classes are supported by special syntax. This special syntax is allowed in addition to the standard object-call syntax generally available for all classes, and is referred to as *aliase*s.

The prime reason for the introduction of aliases is to make the reading and writing of simple programs performing elementary tasks easy. Aliases provide an easy, intuitive syntax for the most common operations and considerably increase the ease with which Blue can be used by beginners. Aliases will be learnt as statements or expressions in their own right by beginners, making it unnecessary to understand all underlying concepts right from the start. The expert programmer or compiler implementor, however, will appreciate the unifying concept for all data types.

Note that aliases are a pure *syntactic* addition which does not add any functionality to the language. They do not affect the semantics or the theoretical language description of Blue (although they are part of the language), and are purely intended to increase readability and intuitivity of statements.

Some common aliases and their resolutions are:

alias	*resolution*
3+6	3.add (6)
b1 or b2	b1.or (b2)
a[i]	a.getElem (i)
str (num)	num.toString
str (a, b, ...)	a.toString.concat (b.toString.concat (...))
print (a, b)	output.write (str (a.b))

The list of existing aliases is short and fixed – programmers cannot define additional aliases.