On Creation, Equality and the Object Model

Michael Kölling and John Rosenberg
School of Computer Science and Software Engineering
Monash University
Email: {michael.kolling, john.rosenberg}@fcit.monash.edu.au

Subject area: language design and implementation

Abstract

In designing languages, we strive for consistency and elegance. In object-oriented languages, simple data types have always been a problem in that they do not quite fit in with the object model. In some respects they seem to be objects, in others they are not. This typically creates the need to introduce special cases into the language definition. This paper discusses the problems involved and proposes an approach to the definition of an object model that allows the elegant inclusion of simple data types.

1. INTRODUCTION

Language design is a difficult task. Designers of languages strive for simplicity, clarity and understandability of languages. One of the techniques that was long thought to bring about these noble goals is orthogonality of concepts. The idea is that a language should be based on a few independent conceptual ideas which hopefully do not interact too much and can be understood independently. Combinations of concepts then could be intuitively understood and would not need to be defined explicitly in the language definition.

Unfortunately, it has always been known that this simplistic model does not work in practice. Orthogonality is undoubtedly an important and valuable design guideline, but that is what it is: a guideline. Every language designer has experienced the frustrating problem of constructs interacting in unforseen ways, forcing the need to introduce special rules for special cases – ugly stains on the elegant and clean design of the language, resulting in orthogonality breaking down. As Meyer put it: "... the idea of orthogonality, popularized by Algol 68, does not live up to its promises: apparently unrelated aspects will produce strange combinations, which the language specification must cover explicitly." [9, p.500]

Those constructs which need to be covered as special cases in the language specification are a nuisance. They make the language description more complicated and the language harder to master. They often carry an implicit danger of being misinterpreted if the programmer does not know that the construct is a special case. A language that can be taught using a small number of principles can be learnt more easily than one that involves learning a large number of special cases.

This paper discusses three design elements of object-oriented languages and their interactions: the object model, especially the representation of simple data types in the object model, creation of objects, and the question of equality. These are areas in which orthogonality and unification of concepts break down in many object-oriented languages. We will identify the source of the problems and propose a new construct that eliminates the need for special cases in language design in these areas.

2. THE OBJECT MODEL

The aspect of the object model of languages we want to discuss here is the way in which simple data types (such as integer, boolean, etc.) are integrated in the model.

There are two different approaches:

1. Simple data types are not classes; their values are not objects. They are different kinds of entities which are handled differently and follow their own rules.
2. Simple data types are classes and their values are objects.

The first approach is taken, for instance, by C++ [10] and Java [1]. In this approach, two different concepts are introduced: the concept of objects and the concept of simple (scalar) data. There are several arguments for this approach: efficiency, syntax, object creation and identity issues.

Efficiency is one of the most commonly used arguments, but at the same time one of the weakest. We have to distinguish the abstract model from implementation issues. It is certainly inefficient to implement integers or characters in the same manner as general objects are implemented, but that is not really an argument against using an *object model* for those scalar types. It is merely a question of implementation optimisation. A language can easily be defined in a way that integrates simple types into the object model and still allows an implementation that treats those types in a more efficient manner.

The next argument is the syntactic one. Many operations are commonly used for those simple types for which a syntax is very widely known. For example, addition for integer numbers is usually written as

```
3 + 4
```

If the numbers were objects, then maybe the syntax for their operations should follow the general object invocation syntax, e.g.

```
3.add (4)
```

This syntax can then be criticised as unintuitive, inconvenient and confusing.

This argument, too, is a rather weak one. It could be argued that syntax is not such an important issue for languages. After all, a programmer has to learn the syntax for a new language anyway, and so why should it be more difficult to learn new syntax for a known construct? Leaving this issue aside, there are some simple solutions to this problem in general use. Several languages (e.g. C++, Smalltalk [3]) solve the whole problem by simply allowing the notation

```
3 + 4
```

as valid method calls.

The real issues with simple data as objects, which usually lead to the decision not to follow this approach, are problems with object creation and identity. Those issues will be discussed in detail in the following sections. Initially, however, we will examine the disadvantages of the first approach, where simple data are *not* interpreted as objects.

Treating objects and simple values as different concepts leads to different semantics (special cases) for a number of constructs. Variables, for instance, can then contain either simple values or references to objects as in Java, or additionally they may directly contain objects or references to simple values as in C++.

Let us look at the simpler case: Java. The parameter passing mechanism is defined to always pass a reference *unless* it is a simple type – then the argument is passed by value. Beginners often get confused in this respect with the string type. Because string is predefined, and because most predefined types are simple types, it is often considered to be similar to integer, bool, etc. But string values in Java are real objects and thus passed by reference - a special case among the predefined types.

Another example where this distinction causes problems is in conjunction with generic composition. All object-oriented languages support polymorphism. It is available through inheritance and, in many languages, through genericity. Making a distinction between simple data items and objects might lead to problems in combining them.

A generic list class, for instance, might accept any object type as its generic type parameter. Thus, it allows the creation of a list of persons as well as a list of buttons. But can a simple type be used where an object is expected? In other words, can we create a list of integers? If a distinction is made between simple types and objects, the answer is often no. (Java tries to overcome this problem by providing *objects* representing the simple data types *in addition to* the simple types themselves – a further special case to deal with and an unwanted effect when combining concepts.)

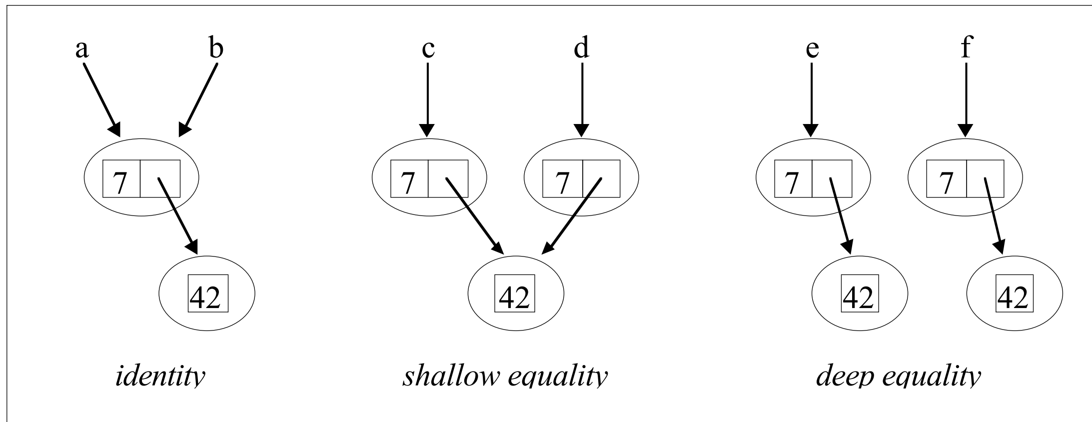

Figure 1: Objects and equality

The second approach, treating simple data as objects, avoids these problems. We can stipulate that all variables hold references to objects. Thus an integer variable holds a reference to an integer object. (Remember: this is the model; the implementation may be different.) The fact that all dynamic structures can be explained with a unified model is very attractive in the design of a language. Parameter passing, polymorphic structures, genericity – all of these constructs could be introduced without the need for special cases. Smalltalk is the best known language to follow this approach. Other languages which implement this model include Dee [4] and Blue [7], languages which, unlike Smalltalk, combine the clean object model with static, strong typing.

Looking at the arguments so far, it is evident that the unified model is preferable, if other problems can be avoided or satisfactorily solved. And there are indeed some problems which need to be addressed. The next sections discuss these problems of the unified model.

3. CREATION

One of the problems with viewing simple types as objects is the question of creation: when and how are those objects created? When we want to add the two numbers 7 and 4, for instance, we do not want to be forced to issue creation instructions first, to create those numbers, before we can add them. The languages that view simple data as objects have different answers to this question. In Dee, a data literal (the symbols “7” or “true”, for instance) is defined to be a *constructor* for the data object. The expression

```
4 + 7
```

therefore has the following semantics:

- an integer object with value 4 is created
- an integer object with value 7 is created
- the + method of the 4-object is called with the 7-object as parameter

Thus another special case is introduced to cope with this issue, this time for object creation. Objects can be created by calling a constructor method of the form

object.method, by using the “new” operator or, for some objects types only, by naming a literal. But there is even another special case within this one: An object is created only the first time the literal is named. After that, the literal “7” refers to the same object that was previously created [5, p.10].

Smalltalk takes a different approach: the literal “7” is defined as a constant reference to the 7-object. The object simply exists in the Smalltalk universe. The question as to when and how it was created is never addressed in [3]. This approach works quite well: identifiers always refer to objects and literals are just a special form of predefined identifiers which refer to some existing objects. The special case here is confined to the fact that these objects never have to be created by the user, but already exist in the environment.

4. EQUALITY

The question as to what equality means must be addressed by every object-oriented language. There are at least three different possibilities which are in use in various languages: *identity*, *shallow equality* and *deep equality*. The expression $a = b$ can mean either one of these in different contexts in different languages.

Identity means that $a = b$ is true if a and b refer to the same object. It would not be true if a and b referred to different objects, even if all the attributes of a and b were the same. In figure 1, a and b are identical. Identity is also called *reference equality*.

Shallow equality means that $a = b$ is true if all the attributes of the objects to which a and b refer are identical. Shallow equality easily can be implemented by a bitwise comparison of the memory space that represents the two objects. Identity implies shallow equality. In figure 1, c and d are “shallowly equal”, but e and f are not.

Deep equality means that $a = b$ is true if each attribute in a and b is either identical or deeply equal. e and f in figure 1 are “deeply equal”[1]. Deep equality is implied by both identity and shallow equality. In this sense deep equality is the weakest form of equality and identity is the strongest. These three types of equality are often used because they are convenient to implement: all three equality checks can easily be generated by a compiler (in the case of deep equality, the compiler might need to use tag bits to prevent infinite loops if a structure to be compared has circular references). But there is another problem: none of these might be appropriate.

The first kind, identity, is useful in many situations, and available in every object-oriented language. It is not enough, however, especially in languages that support immediate objects (objects that are not stored by reference, but are stored in a variable directly). In such languages equality cannot be defined as identity, since every assignment for those

---

[1] In this figure, integers are represented as immediate values, not as objects. The two occurrences of “7” are considered identical. In languages in which simple types are represented by objects, both would contain a reference to the 7-object.

objects involves a copy operation. No two immediate objects would ever be equal. Regardless of whether immediate objects are supported, shallow and deep equality also typically do not always provide the comparison that is required, at least not for non-trivial objects.

In non-trivial systems, equality of objects is often defined as something between deep and shallow equality. To check whether we want to regard two objects as equal in a certain context, we might require some attributes to be compared by identity and others by deep equality, while some attributes may be allowed to be different altogether. In a library system, for instance, two books might well be considered equal if they have the same title, author and ISBN-number, although their library specific item-number might be different [4]. We call this *semantic equality*. In [4], Grogono differentiates *intensional* and *extensional* equality for this purpose. Two objects are intensionally equal if they have the same representation and extensionally equal if they have the same abstract value. Intensional equality corresponds to deep equality and extensional equality is equivalent to semantic equality.

The problem with semantic equality is that it cannot be automatically generated by the compiler. A mechanism must be provided for the user to define semantic equality.

Now we can come back to the language: what do we mean when we write $a = b$?

One of the main considerations is that, at least for simple types like numbers, we want equality after assignment. Consider:

```
a := b
if a = b then
   ...
end if

a := 42
b := 42
if a = b then
   ...
end if
```

In this example, we would hope that *a* is in fact equal to *b* in both *if* statements. Anything else would be highly confusing. This means that the definition of the equal-operator is heavily influenced by the definition of assignment and the object model.

In Smalltalk, since integers are objects like all other objects, they can be cloned like every other object. An effect of this is that equality cannot be taken as identity (since otherwise semantically equivalent numbers might not be regarded as the same).[2]

In languages like Smalltalk, equality is typically defined to be shallow equality. This is fine for simple types, because they do not have a complex structure. We stated earlier,

---

[2] Some Smalltalk systems implement this differently: the clone operation then does not, in fact, clone the integer object. This does not significantly change our argument: the special case, which we are pointing out here, is then shifted from the equality operation to the clone operation.

however, that for complex objects shallow equality is typically meaningless. Identity and semantic equality are needed here.

The effect is that equality can mean different things in different contexts.

In Smalltalk, the operator == defines identity. The operator = has the same value by default, but is — also by default — redefined for all simple types to be shallow equality (which is equivalent to semantic equality for simple types). Thus, by default, the operator = means semantic equality for simple types and reference equality for everything else (but can be redefined to be semantic equality by the implementor of a class).

The drawback of this definition is that the same operator represents different functionality in different contexts so as to behave sensibly for both simple types and complex objects.

The language Dee is slightly different: the = operator means shallow equality for simple types and semantic equality for user-defined classes (with the responsibility for implementing the = operation lying with the user — no default is provided). Since shallow equality corresponds to semantic equality for simple types, it is more consistent than Smalltalk. The price it pays, though, is that the = operator is by default undefined for user-defined types. Dee provides a routine called “same”, which may be inherited from a predefined “Any” class, to check for identity.

We have deliberately restricted the discussion here to languages that have attempted to achieve a degree of simplicity and uniformity by not supporting immediate object variables — variables where the object is stored directly, instead of a reference to it. In languages providing immediate variables (such as C++ and Eiffel), the whole issue becomes considerably more complex and confusing. This can be seen in C++ by the importance of the “orthodox canonical class form” [2, p.38], a language idiom that tells the user to redefine the compiler-provided versions of construction and assignment to avoid the associated incorrect default semantics. In Eiffel, the language definition has to provide tables with all possible combinations of immediate (“expanded” in Eiffel terminology) and reference variables for assignments and equality comparisons, and specifies the semantics for every case separately [9, p.317 and p.336] — the worst case scenario of special cases in a language.

We will not attempt to include these constructs in our discussion. Instead, we direct our attention back to those languages which show some hope of achieving simplicity and uniformity — languages that do not support immediate variables.

5. INTERACTIONS

We have seen that interactions between the object model, object creation and equality cause problems that force us to define special cases for one or more language constructs.

If simple types are defined to be different from objects, we duplicate data storage mechanisms, have distinctions in the semantics of parameter passing and difficulties in finding a unified form to handle polymorphism over both types of data.

If simple types are defined as objects, special cases must be introduced in the object creation mechanism to ensure the convenient use of simple types and in the definition of equality.

Both solutions are not ideal because they introduce definitions that seem arbitrary at the logical model level. The next section introduces an alternative that deals with these issues in a more consistent and unified manner.

6. MANIFEST CLASSES

We start by defining a new concept for the language: *manifest classes*. We distinguish these from "normal" classes by referring to those as *constructor classes*. These two kinds of classes differ in the way their objects come into existence. Constructor classes are those with which we are familiar from most object-oriented languages. Initially, no objects of these classes exist. They can be created by executing a *construct* instruction which creates the object and executes a constructor routine defined in the class. Manifest classes on the other hand define their objects by enumeration. All possible objects of these classes are automatically created at system startup, and the class defines identifiers by which these objects can be referenced. So instead of providing a construction method for objects, manifest classes provide the objects themselves. No additional objects can be created at runtime.

The classes "Integer" and "Boolean", for example, are manifest classes. All their objects automatically exist, and references are provided to allow access. The symbol "3", for example, is a constant reference to the object representing the integer three.

This approach has, of course, similarities with the solutions in Smalltalk and Dee. The difference, however, is that now the distinction is made at the logical level. The object model explicitly recognises these two different types of class, and differences between numbers and complex objects can be understood at the logical level. They cease to be anomalies or special cases at a technical level. This reduces the danger of misunderstandings on the side of the programmer.

One effect of the definition that no objects of manifest classes can be created at runtime is that they cannot be cloned. This ensures, for instance, that only one object representing the integer 42 can exist. Equality for manifest classes can thus be defined as identity (since it is guaranteed to be equivalent to semantic equality).

This distinction, especially the difference between this approach and that taken in Smalltalk, may seem minor. There are, however, differences that could be beneficial. First of all, since the difference that exists in practice has been elevated to the logical model and can be explained in the appropriate terms, there is no longer any need to descend to the levels of implementation or machine representation to explain why, for instance, using the

wrong equality operator might lead to surprising results (as, for instance, the == operator applied to numbers in Smalltalk might do).

An important aspect of this model is the separation of concepts (an improvement in orthogonality): implicit creation is separated from predefined types, leaving the possibility of the combination of implicit creation and user-defined types. This is discussed further in the next section.

The language Blue (described in [8]) implements the manifest class concept. Blue uses a unified object model, where all data are considered objects. Through the introduction of manifest classes and the exclusive use of reference variables, we can define the = operator to always represent identity. This works as expected for simple types since manifest classes cannot be created or cloned at runtime. For complex objects, identity is the desired equality in many cases (since all objects are referenced indirectly and assignment does not duplicate the object). In those cases where semantic equality is different from identity, an "equal" routine may be defined in the class. This routine is in no way special – it is a normal interface routine of a user-defined class. Two objects can then be compared as in the following code example:

```
if a.equal(b) then
   ...
end if
```

The advantage of this definition is consistency: The equal operator (=) has the same meaning in every context, while semantic equality, which is inherently context specific, is invoked through a user-defined name.

Note that the distinction between manifest and constructor classes is a conceptual difference, not a technical one, and that it is on a class basis and not on a variable basis. In this respect it differs greatly from the immediate storage/reference storage distinction in other languages. Our distinction influences how objects come into existence and thus is a semantic definition. It is defined in the class, which already holds semantic information about the class and objects of that class. There can never be confusion over how a variable can be accessed, whether recursive structures are possible or when objects are duplicated, as there often is with the duplicate storage model. All objects are still accessed in a uniform manner.

7. ENUMERATIONS

Manifest classes have another positive effect: through user-defined manifest classes the programmer can provide enumerations. In many languages, enumerations do not quite fit into the object model. Some languages have therefore decided not to include enumerations at all. In C++, one of the few object-oriented languages that supports enumerations, they are defined by a construct completely separate from classes and objects.

Enumeration classes in Blue (user-defined manifest classes) define the objects by enumerating them in the class definition. At runtime, these enumeration objects do not

need to be created - they automatically exist through the definition in the class. If, for instance, an enumeration class *Colour* defines three values *red, green, blue*, then three objects are created and the identifiers red, green and blue are constant references to these objects. In Blue enumeration classes inherit some routines (such as *pred*, *succ*, *ord*) from an abstract superclass *enumeration*. Currently no further routines can be defined by the user for enumeration classes. Whether a language could be defined where user-defined manifest classes can have user-defined methods warrants further investigation.

8. STRINGS

An interesting case to investigate is the handling of the class “String” in this context. Programmers coming from older languages tend to group strings with complex objects, not with simple types (this is especially true for string objects that can grow and shrink dynamically). This grouping is based on the knowledge of how those data types are stored: since a string of this kind cannot be stored on a stack, it is considered a complex object. The grouping is based purely on technical considerations and has no justification in the logical model.

In Blue, strings are, in fact, manifest classes. The reason lies in the way we think about equality.

String literals, such as `"This is a string"` can be interpreted in two different ways: they can be defined as constructors for string objects (Dee uses this approach) or as constant references to string objects (i.e. manifest objects as in Blue).

The main argument for viewing them as manifest objects is that we tend to regard two strings made up of the same characters as being the same. Consider:

```
name := "Leah"
if name = "Leah" then
   ...
end if
```

Intuitively, we would assume the condition in the *if* statement to evaluate to true. In our mind, there is only one string `"Leah"` – we typically do not regard two of those strings as different. (This is fundamentally different from other kinds of objects: we all know that, for instance, two people, even though they have the same first name and the same last name, can still be two different people!)

For this reason, the class "String" is a manifest class; all strings logically exist, and string literals are references to those string objects. An effect of this definition is that strings must be immutable. The Blue string class defines no procedures that change the string. If for example, the characters in a string are to be converted to upper case, Blue uses a function, as shown in this example:

```
command := command.toUpper
```

The routine "toUpper" is a function that returns a reference to a different string. Equally, all other string manipulation routines, which are typically defined as procedures in other languages, are defined as functions in Blue.

This does not prohibit the compiler from performing optimisations to eliminate un-necessary string copies.

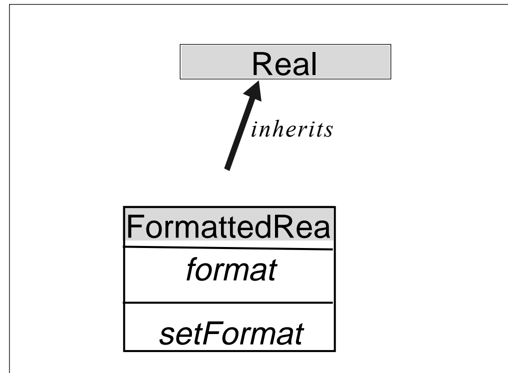

Figure 2: Inheritance example

9. INHERITANCE

So far, everything sounds good. We have removed some significant inconsistencies and introduced a new way to define enumerations. But to leave it at that would not tell the whole story. To complete the picture, we have to discuss a remaining problem with this approach: inheritance.

In introducing manifest classes, we have, in fact, introduced (or not removed) another special case: users cannot inherit from manifest classes. We claim that a simple data item is a (manifest) object. There does not seem to be a good reason to prohibit a programmer from inheriting from its class. Yet in Blue, inheriting from manifest classes is not possible.

This problem is not specific to our definition of manifest classes. It is a problem that affects all solutions trying to incorporate simple types into an object model. We use the manifest class model in our discussion, but all arguments equally apply to other approaches.

To discuss inheritance, we use an example from [6]: the classes “Real” and “FormattedReal”. Assume we have a builtin class “Real” and we want to derive a class “FormattedReal” that adds a format field and a routine “setFormat” to the Real class (figure 2). This could be used to specify output formatting. Assume further that we have a variable “fr” of this class. We can write “fr.setFormat” to access the routine or “fr.format” to access the field (if the language and the class allows us access to this data item). But how do we access the value?

The answer is that there is no access routine for the value — the object itself represents the value. Consider the integer object “5”. The class “Integer” does not define a field that

holds the value 5. The class does not, in fact, define any internal data. The object represents the value by its identity. It is the 5-object. Its value is not defined by some internal storage, but simply by its relationship to other objects: when the "add" method is called with the 2-object as a parameter, it returns the 7-object. The 5-object represents the value 5 by identity.

Accordingly the 5.0-object represents the real value by identity, and the same should be true for "FormattedReal" objects. Consider the following variables:

```
var
  x : FormattedReal
  y : FormattedReal
```

To do calculations, we should be able to write "x+y", not something like "x.value+y.value".

This, however, does not fit in with our model of manifest classes at all. In order to understand the problem, we first consider a non-manifest example. Assume a class "Student" inherits from class "Person". When we have a student object (say "John Smith"), we can also regard that object as a person (we assume that inheritance establishes a subtype relationship). The important thing is that, although a student "John Smith" exists and a person "John Smith" exists, we do not create two objects. "John Smith" is created once, and the object represents at the same time the student and the person (depending on the context). This identity is crucial for our programs to work.

Now consider "Real" and "FormattedReal". Since they are manifest classes, all objects are implicitly created. But if that is so, a 5.0-object will be created for "Real" and another 5.0-object for "FormattedReal". Rather than having one object with two views, we have two objects. We cannot allow this — it fundamentally violates the generally accepted concept of inheritance.

We could try to adjust our definition of manifest classes. We could say that, where subclasses exist, all objects are created of the subclass, not of the superclass. In that sense, all reals would be formatted reals. A user just does not have access to the format part if the object is accessed through a variable of type "Real". This is consistent with our idea of inheritance, but it does not work either. If, for example, two subclasses inherit from "Real", we still have to create reals twice and have the same identity problem.

The conclusion of this is: manifest classes and inheritance simply do not go together.

It is important to emphasise that this is not a problem of our definition of manifest classes – it is an inherent problem of simple types in general. If simple types are not treated as objects, then there is no question of inheritance at all. If they are, then they form some special case.

It is also important to emphasise that the reason these two concepts do not work together is not a mere technical difficulty. Our conclusion is that inheritance of simple types *does not have a sensible meaning*. The reason is that inheritance is concerned with attributes

(fields and routines) whereas simple types use identity as a determining factor. *Identity cannot be inherited*.

A consolation in this realisation that some kind of special case is needed is that the concept of manifest types give us a means to express this rule at a conceptual level. Instead of pretending that simple objects are objects like all others, and then introducing an arbitrary exception, we can add the (semantic) rule: Manifest classes cannot be inherited.

10. CONCLUSION

Simple types do not fit in well with the object model of object-oriented languages. It is favourable to regard simple types as objects in some respect, while they are different from objects in other contexts. This problem typically leads to the necessity to define special cases in areas of the language definition.

We have introduced the concept of manifest classes. This concept moves the necessary distinction to the logical level and removes the need for special cases for single constructs. It also allows the seamless inclusion of enumerations into the object model. We have also argued that the concept of inheritance from simple types does not have a sensible meaning in programming languages.

ACKNOWLEDGMENTS

We would like to thank Chris Exton and Prof. Peter Grogono, who have read drafts of this paper and pointed out interesting thoughts and questions that made their way into the paper. Specifically, Chris Exton mentioned the connection of the need for the orthodox canonical class form in C++ with the topics discussed in this paper. Professor Grogono, apart from many other comments, pointed out the inheritance problem.

REFERENCES

1. Arnold, K. and J. Gosling, *The Java™ Programming Language,* Addison Wesley, 1996.
2. Coplien, J., *Advanced C++ Programming Styles and Idioms,* Addison-Wesley, 1994.
3. Goldberg, A. and D. Robson, *Smalltalk-80: The Language,* Addison-Wesley, 1989.
4. Grogono, P., *Issues in the Design of an Object-Oriented Programming Language.* Structured Programming, 1991. 12(1): 1-15.
5. Grogono, P., *PC-Dee: Syntax and Semantics,* Dept. of Computer Science, Concordia University, Report February 1994a.
6. Grogono, P., *personal communication,* 1998.

7. Kölling, M. and J. Rosenberg, *Blue - A Language for Teaching Object-Oriented Programming*, in Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, 190-194, ACM, March 1996.

8. Kölling, M. and J. Rosenberg, *Blue - Language Specification, Version 1.0,* Department of Computer Science and Software Engineering, Monash University, Technical Report TR97-13, November 1997.

9. Meyer, B., *Eiffel: The Language,* Prentice Hall, 1992.

10. Stroustrup, B., *The C++ Programming Language,* second edition, Addison Wesley, 1991.