

Towards better programming - a taxonomy and model of object oriented mechanisms

Basil Worrall and Judith Bishop

University of Pretoria, Department of Computer Science, South Africa,
bworrall, jbishop@cs.up.ac.za
<http://cs.up.ac.za/~jbishop/>

Abstract. The concepts and mechanisms of abstraction and polymorphism are usually the foundation on which programmers build their skills in an object oriented language. Along the way the programmer adopts certain mechanisms that are used in preference to others, which he understands and feels most comfortable using. This practice often mitigates against considering other mechanisms offered by the language, and against switching to a different more appropriate language. In this paper we propose a taxonomy of object-oriented mechanisms, and develop a practical model which presents the mechanisms in a uniform and simple way via a standard set of programming elements and diagrams. From the standpoint of abstraction and polymorphism, we consider inheritance, interfaces, delegates and generics. Together with comparative tables and summaries, the model can assist a programmer to use higher order abstraction and ultimately lead to better programming practices.

1 Introduction

The concepts and mechanisms of abstraction and polymorphism are usually the foundation on which programmers learning object-oriented programming build their knowledge and skills in a specific language. Along the way the programmer adopts certain mechanisms that are used in preference to others. The preference is for mechanisms that the programmer understands best and feels most comfortable using. This practice often leads to a closed-minded attitude when considering other mechanisms offered by their favourite language. It also mitigates against switching to another language. Observing extreme degrees of fanaticism, such programmers may consider perfectly valid alternatives as unorthodox, not properly supported, bad programming practice, or inefficient. In reality, such attitudes lead to program code that is tangled and over-complicated by the programmer's attempts to avoid considering all mechanisms appropriate for the task at hand.

A prime example of such an attitude is found in programmers favouring C++, yet avoiding the use of templates. Their argument against templates will usually be stated in terms of the difficulty experienced in reading of the code, a lack of a complete understanding of the semantics, or even complaints that the error messages generated by the compiler are incomprehensible. Each of

these does have some degree of truth. The notation used for templates in C++ would seem alien when compared to the standard language syntax, making the code seem unreadable. The semantics of templates are tricky to understand at first, because it is unclear what the compiler produces when it encounters parameterized code. Furthermore, many C++ compilers do produce somewhat obscure messages regarding the source of errors in the code (although this is not restricted to template code). However, exactly the same arguments can be invoked when considering the alternative program which would require, say, a different version of a list type for each data type to be collected, with inheritance hierarchies four or five levels deep and type-casts at every other statement. In all likelihood, not only would such a program suffer from the symptoms mentioned above, it would also have less than optimal execution time and would be difficult to debug at run time.

The ethic in object-oriented software engineering has always been to produce reusable code that is both optimal (in terms of memory footprint and execution speed) and easily maintained. It is our belief that the optimality and maintainability of code resources are jeopardized by a strong desire among programmers to remain in a “comfort zone,” i.e. a design and programming style that denies the use of arguably better methods because of a perceived lack of understanding. We propose that a programmer both be familiar with and frequently use all forms of abstraction and polymorphism. Often taking the road less traveled leads to more open program design and better abstraction models for the problems being solved.

In this paper we propose and develop a taxonomy for object oriented mechanisms (Section 2) and develop it into a uniform model which explains the elements of the mechanism, and presents simple examples of how it operates (Section 3). Section 4 summarises the model in terms of the languages of the day - C++, Java and C#. Section 5 concludes.

2 A Taxonomy of OOPS Concepts

2.1 Background

Two important characteristics of object-oriented programming languages are abstraction and encapsulation. *Abstraction* in a language handles complexity by hiding implementation details [9][12]. Thus a programmer may extract and use the features of a system that contribute towards the solution of a specific problem. *Encapsulation* allows a language construct to consist of both attributes and operations, which combined describe the construct’s behaviour. These constructs form a unit of abstraction, and started out being referred to as abstract data types.

Abstraction and encapsulation are not enough for an object-oriented language. It also needs instantiation of objects at run-time, polymorphism and generalization and specialization mechanisms. Some of the concepts that these attributes map to in many object-oriented languages programming languages

are types, classes, interfaces, inheritance, delegates and templates. A good discussion of the various concepts is found in [9] and [11]. We define the terms type and class here, and relate them to similar terms used in the literature, then take the remaining terms into the taxonomy.

2.2 Type

In typed languages, each variable needs to have a specified type. The type is used to describe the structure of the value or values contained in that variable. Furthermore, the type is used to ensure correctness of a program by enabling type checking (or typing) at compile time. Typing is the determination of the type of expressions, variables and values to ensure that a program runs without errors. There are three kinds of typing namely *static*, *strong* and *dynamic* typing. Object-oriented languages require strong and/or dynamic typing to enable inheritance and polymorphism concepts we discuss later. Strong typing means that all types are established at compile time, and a program will never fail due to a type incompatibility error. Dynamic typing means that types are established at run time. Therefore a program may abort due to a type error.

2.3 Class

Ellmer defines the term class as fulfilling two notions based on the Aristotelian theory of concepts, namely *intension* and *extension* [9]. Intensionally, a class is a blueprint for all instances of itself. Instances of a class are called *objects* and they are the subject of discourse in all object-oriented programming languages. We use the term instance to denote the physical nature of an object. An object in a language has

- storage for state and structure,
- identity to distinguish it from another object, and
- interaction capabilities to enable it to communicate with other objects.

By intension, therefore, we say that classes define the attributes and behaviour of objects. Extensionally, a class can be seen as maintaining its extent (all its instances) and providing a means to generate new instances of itself (constructors). The term *abstract data type* (or ADT) as used in object-oriented literature, is synonymous with class.

The difference between class and type is succinctly explained in [9]. “Types specify the structure and semantics of values while classes are also concerned with implementation aspects.” An example may help to see the difference. A list may be implemented in C++ using references (pointers) - `PtrList` - or arrays - `ArrayList`. Both implementations provide the methods `add` and `remove`. `PtrList` and `ArrayList` are different classes (their implementations differ), yet they are the same type (their behaviour is equivalent).

Our objective in this paper is to present various abstraction and implementation mechanisms that can be used interchangeably in modern object-oriented

languages, and to examine the various advantages and disadvantages of each. With our observations in this regard in mind, we present a model which can be used to select the appropriate mechanism for a given problem.

2.4 Taxonomy of Concepts

Kristensen and Østerbye, in [14], define abstraction as the process of forming concepts based on observing phenomena. They present a model for abstraction that includes the elements *classification*, *aggregation* and *specialization*. *Classification* is done either through the process of observing a phenomena and forming a concept (termed classification), or through mapping a concept to a phenomenon (termed exemplification). Aggregation and specialization are mappings between different concepts. *Aggregation* occurs when a concept is constructed from more simple concepts. Its opposite is *decomposition*, when a concept is broken up into multiple simple concepts. *Specialization* occurs when a concept is formed by taking a closely related concept and adding properties to it. The opposite process is *generalization*. Figure 1 shows the hierarchical arrangement of the abstraction elements.

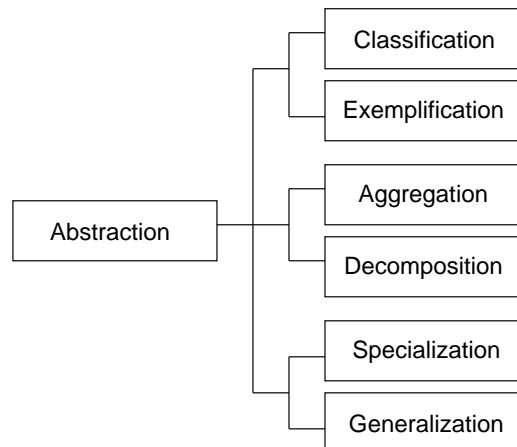


Fig. 1. Abstraction elements

The elements of abstraction have their counterparts in programming languages. Classification is the instantiation of a class, while exemplification is the treatment of an object as an instance of a class. Aggregation is the construction of a class from other classes. Specialization and generalization are seen in the inheritance mechanisms offered by object-oriented languages.

Abstraction works at the class level. In other words, the elements of abstraction deal with the use and structure of a class. The abstraction concepts

classification and specialization lend themselves to use at the behavioural level of classes. This usage is known as *polymorphism*. Polymorphism is the ability of a language to assign more than one type to the values and variables in a program. Cardelli and Wegner classify four kinds of polymorphism in [6].

1. *Parametric polymorphism* is obtained when a method works uniformly on a range of types. These are also called generic methods.
2. *Inclusion polymorphism* is obtained when a class may be a specialization or a generalization of another class.
3. *Overloading polymorphism* is obtained when an operation provided by a class may be performed by different methods.
4. *Coercion polymorphism* is obtained when a parameter to a method is converted to another type. In some cases, the coercion is implicit (i.e. the compiler will ensure that the object is converted to the appropriate type), in other cases we need explicit coercion (also known as type-casting) to ensure correct behaviour

3 A model for OO Concepts

Thus far our discussion of abstraction and polymorphism has been from the conceptual perspective. In this section we shall present the concepts of inheritance, interfaces, delegates and generics (templates), all of which are mechanisms used in most modern object-oriented languages (C++, Java, C#) and which implement abstraction and polymorphism. The contribution of this section is the development of a uniform model by which the mechanisms can be explained and therefore compared. The summary that follows, when read with this discussion, should encourage programmers to employ more of the higher order mechanisms where they are appropriate.

3.1 Elements of the model

The model is based on diagrams which draw from a fixed set of program elements. The common names of the elements are intended to aid a programmer in moving from the more well known mechanisms (inheritance, interfaces) to the seemingly higher-order ones (delegates and templates). The elements are:

- D is a driver class from which abstraction and polymorphism will be initiated
- A is a collection (array, list, hashtable) which is potentially polymorphic
- C_1 , C_2 and C_3 are classes which are intended to be related in some way
- M is a method which defined in the classes and called from D
- I is an interface which will specify M 's signature
- M_1 , M_2 , M_3 and M_4 are also methods, closely related in behaviour
- G is a delegate
- T is a placeholder for a type in a generic class

3.2 Inheritance

A class may be declared as a specialization of one or more classes. C# and both support single class specialization (also known as single inheritance). C++ [8] is one of the few languages that supports multiple inheritance. Multiple inheritance is seldom practiced as it introduces possible conflicts in a hierarchy of classes. Ellmer describes a number of conflict resolution procedures that can be used in various situations [9].

Figure 2 illustrates the model for single inheritance. In the diagram C_1 is the *base* class that classes C_2 and C_3 specialize. C_2 and C_3 may *override* the method M declared in C_1 with their own definitions. In other words they may provide a specialization of the behaviour represented by $C_1.M$ for application in their own type. A precondition for overriding is that M in the inheriting class must have the same *signature* as the original M . The signature of the method is comprised of its return type, its name and the number and types of its parameters. If M in the inheriting class has the same name and return type, but different parameters, it is said to be *overloading* M in the original class.

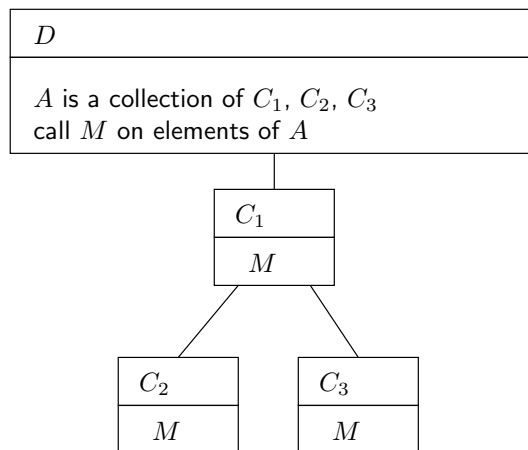


Fig. 2. Model for inheritance

D defines a collection A which contains objects of C_1 , C_2 and C_3 . If we assume that the collection type of A was defined only for objects of the type represented by C_1 , we see the application of inheritance in an object-oriented language. From the perspective of specialization abstraction, C_2 and C_3 are type-compatible with C_1 since we have only added to the structure and attributes of the base class. They still share the base structure of C_1 . From the inclusion polymorphism perspective we could have added methods besides M of C_1 to C_2 and C_3 , overridden M , or both. If we added more methods, we know that M can still be called, and the use of objects of C_2 and C_3 in A is still valid (accessing

and modifying an object contained in A , without coercion, is constrained by the fact that this object is of C_1). On the other hand, if we overrode M in C_1 or C_2 , we have a more complicated situation.

The reason for the complication is that there are diverse interpretations of the term inheritance, and much literature in the field of object-orientation is dedicated to describing each interpretation and implementation. Ewings thesis [10] provides an excellent explanation for the diversity in inheritance mechanisms, and lists eighteen realizations of inheritance encountered in the literature. Our interest here is in strict specialization (as opposed to inheritance where no behavioural modification is made), and we present the following three mechanisms present in modern object oriented languages.

1. **Abstract Inheritance** Any class can be called *abstract*, but its real use is when it does not provide an implementation for one or more methods it declares. Such an omission has the effect of forcing specialisation down to other classes if any objects are to be instantiated. However, the abstract class itself, can still be used as in declarations.

In our example, if C_1 is abstract, then it can be used as the type on which A is declared, and creation of objects for A will have to be done from specializations of C_1 , namely C_2 and C_3 . Thus the declaration of A on an abstract class encourages polymorphism. In other words, we are forced to inherit from an abstract class if we wish to use any of the implementations it does provide. However, an object can be seen as an instance of an abstract class if it is type cast (coerced) to this class from a specializing class. It is also possible to define collections based on abstract types. For example, if we had made M (and consequently C_1) abstract in the figure, declaring A to be an array of C_1 would not have resulted in an error. We would simply have been restricted to inserting instances of C_2 and C_3 into A . With the following definition of C_1

```
abstract class C1 {
    abstract public void M();
}
```

we can declare

```
C1 A = new C1[10];
```

and after C_2 and C_3 objects have been placed in A , still write a loop over C_1 objects, as in:

```
foreach (C1 c in A) {
    c.M();
}
```

2. **Virtual Inheritance** This mechanism sees a class inheriting from another well-defined class with fully implemented behaviour. By the concept of specialization, this means that an inheriting class either adds to or replaces

the methods. When we replace the methods we need to consider the degree of modifiability. There are two degrees of modifiability of behaviour [9], *arbitrary redefinition* and *constrained redefinition*. If a language supports arbitrary redefinition, any and all methods may be modified by the specializing class. In constrained redefinition, a class may dictate those methods it grants the rights to specialization in inheriting classes.

Virtual inheritance has the benefit of behaviour predictability. This means that the methods of the specializing class will be used even when its objects are cast to another class higher in the hierarchy. The identity of the method to call during execution is made available through *late binding*, another name for virtual inheritance. Java has arbitrary redefinition semantics and only allows virtual inheritance [2]. Methods in C++ classes are subject to constrained modification, and the user can specify whether they are virtually inherited or not. C# only allows those methods declared to be virtual to be overridden in specializing classes.

Suppose C1 and C2 are defined with virtual inheritance in C#.

```
class C1 {
    virtual public void M() {
        Console.WriteLine("This is C1.M()");
    }
}

class C2 : C1 {
    override public void M() {
        Console.WriteLine("This is C2.M()");
    }
}
```

then if A is declared of type C1, and an object of class C2 is created in A[5], then the foreach loop shown previously will call the M in C2, as expected for the object A[5]. We shall see how this behaviour may vary when discussing the next technique.

3. **Non-virtual Inheritance** Non-virtual inheritance has multiple redefinition semantics. Setting this technique apart from virtual inheritance is the fact that casting an object to one of its parents in the hierarchy results in the parents methods being invoked instead of the object's. Herein lies the variability of the behaviour of such an object. Using the same example, suppose I and C2 are declared as follows, with a foreach loop to call M on each element of the collection, as in:

```
interface I {
    public void M();
}

class C2 : C1 {
    new public void M() {
```

```

        Console.WriteLine("This is C2.M()");
    }
}

foreach (C1 c in A) {
    c.M();
}

```

A[5] (and every element of A) is cast to C1, so that in fact calling A[5] calls C1's M. The method call was statically bound to C1 at compile time. We could also afterwards force a call to C2's M as in

```
((C2)A[5]).M();
```

We force a method to undergo static binding by marking it with the *new* keyword in C#. Java does not have this form. With C++ it is the default.

3.3 Interfaces

Interfaces are similar to abstract classes, in that they do not provide implementations for their methods. The difference is that only initialized attributes (sometimes only constants) may be specified in an interface, whereas abstract classes allow any attributes. Figure 3 shows the interface model.

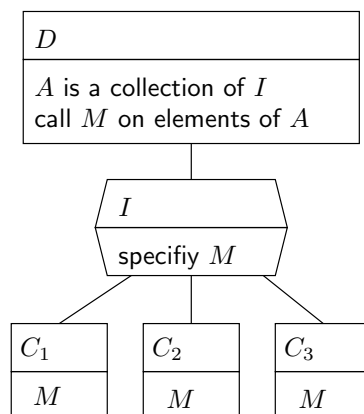


Fig. 3. Model for interfaces

In the diagram *I* is the interface which specifies the method *M*. *C*₁, *C*₂ and *C*₃ are classes that *implement* the interface. Implementation is similar to abstract inheritance, as the inheriting class is forced to implement the methods specified in the interface. Therefore *C*₁, *C*₂ and *C*₃ must each provide an implementation of *M*. The implementations of *M* in each class need to have the same signature

as M in the interface. A class may implement multiple interfaces, which sets implementation aside from abstract inheritance as a form of abstraction. We shall see an example of this distinction later.

Each entry in A is considered an instance of I . The class D therefore has access to all methods and attributes defined in I without knowing the full structure and behaviour of the objects of the various classes. Concentrating again on C_2 , it may be defined in C# as

```
class C2 : I {
    public void M() {
        Console.WriteLine("This is C2.M()");
    }
}
```

In this example, each C class implements I. Note that the syntax for inheritance and implementation is the same. This is not the case in Java, where the keyword *implements* is used to list the interfaces. Unfortunately C++ does not directly support interfaces. It is possible to emulate the interface mechanism by creating an abstract (or pure virtual) class that does not provide implementations for any of its methods, and inheriting from that. Using the same foreach loop as before, when A[5] is accessed, C2's M will be called.

An option that Java has is anonymous classes, which can be used with interfaces (and abstract classes) when only one instance of the class implementing the interface is to be created.

3.4 Delegates

This mechanism provides a different form of overloading polymorphism by removing the restriction that overloaded methods must share the same name. However, delegates do ensure that the signatures of methods that are overloaded match. We are not referring here to the delegation as used in classless (prototype) languages such as Self [17]. A discussion of the difference can be found in [16]. In this paper delegation is used in the sense of *forwarding semantics*.

In delegation, the driver class D declares a delegate as a method G, thereby setting the delegate's signature. An object M of this delegate is created and set to null. Now methods from within the class or from other classes may register themselves with the delegate object M in order that they are invoked when the delegate object is invoked.

Figure 4 shows a model for delegates. It shows that as opposed to our previous examples, M_1 , M_2 , M_3 and M_4 may each have a unique name, but still be connected to the delegate object M_G . Depending on the implementation, when D invokes M_G , either the last method registered with it will be invoked, or all the methods registered with it will be invoked in order.

Delegates are provided in C++ and C#. The implementation in C++ is function pointers, and allows only one method to be registered with a pointer at a time. C# delegates behave like types that have an implicit collection structure

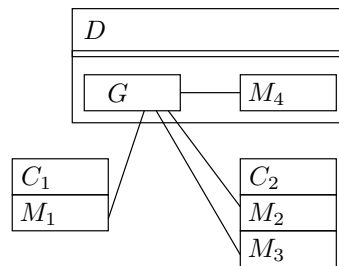


Fig. 4. Model for delegates

that allows candidate methods to be added and removed from it during run time. It is necessary to create an “instance” of the delegate before methods can be registered with it. The code below shows how this is done. Delegation as a syntactical construct is not available in Java - however, it is possible to simulate it using the reflection classes.

Continuing with our example, C2 can now be declared with an individually-named M2, as in

```

class C2 {
    public void M2() {
        Console.WriteLine("This is C2.M2()");
    }
}
  
```

The delegate declaration and instantiation is:

```

public class D {
    delegate void G();
    G M = null;
}
  
```

Registration proceeds as in

```

M += new G(c2.M2);
  
```

and a call to M() will then invoke C2's M2. Notice that with delegates, the collection A is no longer relevant.

3.5 Generics

Generics are object-oriented languages' support for parametric polymorphism. Viroli defines parametric polymorphism, saying “it [parametric polymorphism] allows to abstract a piece of code from one or more types, making it reusable in many different contexts.” [18] Popular uses of generics are in collection classes, where the actual behaviour of the class is independent of the object types stored in the collection. The terminology of generics differs from language to language.

For example Ada has generics [1][3], C++ has templates [8] and ML has polymorphic functions [15]. Generics are so popular among developers that both Sun Microsystems and Microsoft have agreed to add them to their flagship languages in the near future [5][7]. Early versions of the intended languages have already been developed; more information can be found in [13] and [4].

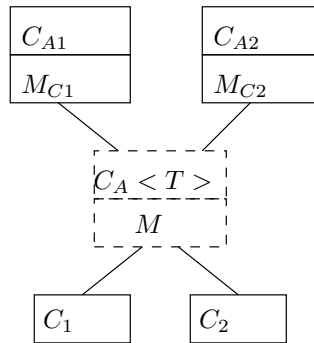


Fig. 5. Model for generics

We present a model for templates in Figure 5. Classes C_1 and C_2 represent types that may be substituted for T in generic class $C_A < T >$ (represented by a dashed box). T is the parameter for class C_A , and is the type that method M in this class will operate on. A represents the type of the class, for instance list or hash table. C_{A1} contains a collection A defined on C_1 types, and can call M on C_1 elements of A . C_{A2} is similar to C_{A1} , except that A is defined on C_2 types.

The following example is in Generic C# [7], and presents an example of a generic stack, following the model in Figure 5.

```

class Stack<T> {
    private T[] store;
    private int size;

    public Stack() {
        store = new T[10];
        size = 0;
    }

    public void Push(T x) {...}
    public T Pop() {...}
}

class UseIntStack {
    Stack<int> stack = new Stack<int>();
}
  
```

```

    public UseIntStack() {
        stack.Push(42);
        int x = stack.Pop();
    }
}

class UseStringStack {
    Stack<string> stack = new Stack<string>();
    public UseStringStack() {
        stack.Push("Hello, World!");
        Console.WriteLine(stack.Pop());
    }
}

```

int and string correspond to C_1 and C_2 in the model. Through the generic class Stack, the two classes UseIntStack and UseStringStack (corresponding to C_{A1} and C_{A2}) are created. As with all the other examples, the full code is on the website (see Section 5)

4 Application of the model

Frequently one mechanism for abstraction or polymorphism may be used to achieve the same effect as another. It is necessary to have alternatives, especially when a particular language does not provide an implementation. We round off our discussion of these constructs by presenting the advantages and disadvantages of each. Table 1 summarizes this information.

In the previous section mention was made of certain peculiarities in the implementation of the four mechanisms in a particular language. Table 2 summarises the support that C++, C# and Java have for the four mechanisms. Where appropriate we have included comments about the syntax used in a language.

5 Conclusion

For those programmers trained in the Java era, since the mid nineties, object-orientation is synonymous with inheritance and interfaces. With the advent of C#, delegates have become a reality, especially in GUI programming which relies strongly on that mechanism. C++ has had templates for many years but they are not widely used, and even less widely written. The intention to put generics in Java and C# is to be recommended, but it remains to be seen whether the mechanism will be taken up by programmers. The model contributed by this paper, presents a means for ordinary programmers (including students) to come to grips with the semantics and syntax of the higher order mechanisms. When read together with the tables, the programmer has more chance of making an informed choice of what to use when, and or being less intimidated by the

Table 1. A comparison of inheritance, interfaces, delegates and generics

	Advantages	Disadvantages
Inheritance	<ul style="list-style-type: none"> • Promotes behaviour reuse • Allows structural and behavioural specialization • Scalable 	<ul style="list-style-type: none"> • Virtual inheritance is slow • Abstract type can be broken (e.g. a stack is not an array)
Interfaces	<ul style="list-style-type: none"> • Promotes thorough abstraction • Allows different views on an object • Lightweight • Scalable • Allows behaviour specialization 	<ul style="list-style-type: none"> • No reuse • Slow (virtual inheritance semantics) • Limited structural specialization
Delegates	<ul style="list-style-type: none"> • Loose coupling in code • Allows multiple naming 	<ul style="list-style-type: none"> • Logic may be difficult to follow while reading code
Generics	<ul style="list-style-type: none"> • Run-time efficient • Promotes reuse • Type safe • Allows behaviour specialization 	<ul style="list-style-type: none"> • May be inappropriately applied • No structural specialization • Not scalable

prospect of moving on from the more well-known constructs. To our knowledge, no such model has been proposed since the early nineties [11].

The work is being extended with a comprehensive suite of realistic sample programs which will illustrate all the mechanisms described in the model. Experimental tests with programmers are also being contemplated, to evaluate further the efficacy of the work. Full details of the programs from which extracts were taken can be found on <http://www.cs.up.ac.za/polelo/model>.

6 Acknowledgements

We acknowledge the financial assistance of the National Research Foundation under Grant 2053401. We thank John Muller and Cobus Smit for stimulating discussions in the early stages of this paper. JMB thanks NTB for his unfailing support in enabling this work to proceed.

References

1. Ada. Reference manual for the Ada programming language. GPO 008-000-00354-8, 1980.
2. Arnold K, Gosling J. The Java Programming Language, 2nd ed., Addison-Wesley, 1997.
3. Bishop JM. The Effect of Data Abstraction on Loop Programming Techniques. In IEEE Transactions on Software Engineering 16(4), pp 389-402, April 1990.
4. Bracha G, Odersky M, Stoutamire D, Wadler P. Making the future safe for the past: Adding Genericity to the Java Programming Language. In OOPSLA'98, pp 183-200. Vancouver, BC, Canada, October 1998.

Table 2. Summary of mechanisms

	C++	C#	Java
Abstract inheritance	Use pure virtual methods	Provided for	Provided for
Virtual inheritance	Use virtual modifier on methods	Use virtual modifier on methods	Default
Non-virtual inheritance	Default	Use new modifier on methods	Not provided for
Interfaces	Use pure virtual classes	Provided for	Provided for
Delegates	Provided through function pointers, only one method per delegate	Provided for. One or more methods per delegate	Not yet provided for
Generics	Provided through templates	Not yet provided for	Not yet provided for

5. Bracha G. JSR 14: Add Generic Types To The Java Programming Language. <http://web1.jcp.org/en/jsr/detail?id=14>. Last visited 23 January 2003.
6. Cardelli L, Wegner P. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys* 17(4), pp 471-522, December 1985.
7. C-Sharp. The C# Programming Language - Future Features. <http://www.gotdotnet.com/team/csharp/learn/Future>.
Microsoft Word Format Whitepaper (VCS Language Changes.doc). Last visited 23 January 2003.
8. Ellis M, Stroustrup B. *The Annotated C++*. Addison-Wesley, 1990.
9. Ellmer, E. *Basic Principles and Concepts of Object-Orientation*. PhD Thesis, University of Vienna, Institute of Applied Computer Science and Information Systems. November 1993.
10. Ewing G. *Class Inheritance: The Mechanism and Its Uses*. Honours project, Department of Software Development, Monash University, Australia. October 1994.
11. Grogono P. *Issues in the Design of an Object Oriented Programming Language*. Structured Programming, January 1991.
12. Jamil, HM. *Semantics of Behavioral Inheritance in Deductive Object-Oriented Databases*. PhD Thesis, Department of Computer Science, Concordia University, Québec, Canada. November 1996.
13. Kennedy A, Syme D. *Design and Implementation of Generics for the .NET Common Language Runtime*. Presented at PLDI 2001.
14. Kristensen BB, Østerbye K. *Conceptual Modeling and Programming Languages*. In *ACM SIGPLAN Notices* 29(9), pp 81-90, 1994.
15. Milner R. *The Standard ML core language*. *POPL*, pp 132-145, 1997.
16. Ostermann K, Mezini M. *Object-Oriented Composition Untangled*. In *Proceedings of OOPSLA'01*, *ACM SIGPLAN Notices* 36(11), pp 283-299, 2001.
17. Ungar D, Smith R. *Self: The power of simplicity*. *OOPSLA'87*, *ACM SIGPLAN Notices* 22(12), pp 227-242, 1987.
18. Viroli M. *Parametric Polymorphism in Java: an Efficient Implementation for Parametric Methods*. *Proceedings 2001 ACM Symposium on Applied Computing*, Las Vegas, Nevada, US. ACM Press, pp 610-619, 2001.