Transactions on Information and Communications Technologies vol 7, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

Pragmatism over purism? An incremental approach to the teaching of object-oriented programming.

D. Parsons

Systems Engineering Division, Southampton Institute of Higher Education, East Park Terrace, Southampton, UK

Abstract

The aim of this paper is to explore two contrasting approaches to the teaching of object-oriented programming, the 'purist' and the 'pragmatic', and to describe in detail how a pragmatic approach may be formalised and implemented.

1 To be pure or not to be pure?

There are two schools of thought in the arena of object-oriented languages. One is that to be object-oriented, a language must be 'pure'; Perhaps the most vocal exponent of this view is Bertrand Meyer (inventor of the Eiffel language). The alternative approach is one of pragmatism; That object-orientation is a tool like any other and that the purity of this tool is not the issue - its usefulness is the sole criteria by which it should be judged. The key advocate of this approach is the inventor of C++, Bjarne Stroustrup. The two languages of these protagonists (Eiffel and C++) reflect this differing ideology. Eiffel is a true object-oriented language, while C++ is a hybrid of C with various orthogonal extensions to allow object-oriented programming. Meyer is quoted as saying that 'compatibility with existing software is not an excuse for polluting the language. The language can be C...or it can be object-oriented, which is completely different' [1]. In contrast, Stroustrup has said 'It is not right to be pure. It is right to serve your own and other's needs...anyway I have a problem with the word 'pure'. because it makes me think of stormtroopers' [2].

2 Building on third generation skills

Those attempting to teach the object-oriented paradigm to students must address the question of which approach is more educationally valid, the pure or the pragmatic? This paper suggests that a pragmatic approach to teaching object-oriented programming with C++ has a number of advantages, particularly when dealing with students who have been immersed in third generation languages. We can teach that object-orientation is a revolution, and abandon most of our students' existing knowledge, or we can teach that it is evolutionary, and that there is a gentler path from the known to the unknown. This paper is based on an attempt to teach object-oriented programming in C++ using an incremental approach, which meant identifying the discrete components of the method and establishing the relationships between them. This conceptual organisation allowed a developmental path to be followed which led students from the known procedural paradigm, via simple object-oriented concepts such as encapsulation, to the more subtle aspects such as polymorphism, multiple inheritance and container classes.

3 What makes a program 'object-oriented'?

It is often said that for a language to be considered object-oriented it must support encapsulation, inheritance and (run-time) polymorphism. It might be assumed therefore that for a program to be object-oriented it should embrace all of these concepts. However, there are many example applications where the emphasis can be on objects of classes with limited inheritance and little or no polymorphism, yet the principles of object-orientation are still suitably demonstrated. It is easy to emphasise the roles of class hierarchies and dynamic binding with menageries of noise generating animals, simulations of various flying machines in an airspace or collections of graphical doodles, but how many applications in practice follow these models? In many contexts the key elements are the inter-object relationships whereby objects communicate with one another in 'using relationships' [3] or 'aggregations' and other forms of object association [4]. In these circumstances, we should be looking less at hierarchies and more at the ways in which objects pass messages to one another in order to perform high level functions ('mechanisms' [3]). In this way we can enable students to write 'object-oriented' programs at a very early stage without their needing to understand every available technique of object-orientation. Such applications place stress on modularity, scope, visibility and message parameters rather than complex hierarchies or object-specific responses to generic messages. As such they promote good practice in understanding how object-oriented systems work at the implementation level without too much conceptual luggage.

Transactions on Information and Communications Technologies vol 7, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

4 The more it changes the more it stays the same

In taking our students down the long path to object-oriented programming, we should at least make them aware that some of the skills they already have will help them on their way. Wirth [5] emphasises the links between activities in procedural programming and those in object-oriented programming. Table 1 (below) indicates areas which have commonality between the two paradigms. In each case we can start with a familiar concept and use it as a springboard for a new concept.

Table 1. Activities in procedural and object-oriented programming

Procedural	Object-oriented
data type	abstract data type / class
variable	object / instance
function / procedure	method / operation / service
function calls	message passing

This approach contrasts with programming style in 'pure' object-oriented languages in that the 'purist' tool tends to abandon the programmer in an unknown environment. The novice Smalltalk programmer for instance is forced to know everything or nothing - since everything is an object and all classes inhabit a single hierarchy there are no half measures. In contrast, the novice C++ programmer can gradually introduce objects into procedural programs, creating working programs almost from the first attempt, first object-based, then class-based and finally object-oriented. C++ is not, of course, the ideal teaching language. It does have certain advantages in the marketplace such as portability, economy, commonality with C and large available libraries of both traditional functions and classes. However, its main drawback is the opaqueness of the C syntax which underlies it. Before any C++ can be learnt, a lot of rather painful C coding must be endured. Nevertheless, C++ has already superseded some of Cs more tortuous aspects like the i-o functions of stdio.h, and a few well chosen class libraries can overcome many of its other complexities.

5 The incremental path

The main theme of this paper is that the object-oriented paradigm may be decomposed into C++ based teaching units, each of which provides a logical progression from the previous stage. Student activities providing the means for integrating these elements into a general understanding can be easily developed to encourage reuse and genericity wherever possible.

Transactions on Information and Communications Technologies vol 7, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

Clearly, the first step in any teaching of object-orientation has to be the object, and therefore the abstract data type (the class) which specifies each object. Students familiar with procedural programming will already know what a data type is, and also what a data record is. It is but a short step from understanding the relationship between a file structure and a data record (which may be seen as a data type composed of more primitive data types), to understanding the relationship between a class and an object. Again there is a conflict of ideology between authors, some emphasising the data aspects of objects, others emphasising responsibility. However, it is easier to understand data, and by extension the operations on that data, than to understand operations before discussing data. It is the abstraction of behaviour as opposed to the concreteness of data which makes this data driven approach more assimilable for the first-time object-oriented programmer.

Once students can create classes and objects, then they can learn to pass messages to those objects. This can be done in an otherwise procedural context, allowing a gentle learning curve. They can then learn the difference between dynamic and non-dynamic objects and the syntax appropriate to each. Although it is a rather unexciting aspect, it is crucial that students understand the subtleties of object scope, visibility and lifetime, and the memory management implications of these, before attempting ambitious object-oriented programs.

6 Inheritance, but not too much

Because inheritance is such an important aspect of object-orientation, it needs to be introduced as soon as classes and objects are understood. However, its role should not be overemphasised. Students tend to lay too much stress on the rigid, static structures of a class hierarchy, sometimes at the expense of understanding what happens dynamically when a program runs - a classification hierarchy has its uses, but objects are more important than classes. The Smalltalk approach whereby every class is in a single hierarchy is simply not necessary and overcomplicates any analysis and design which attempts to allow for this.

One aspect of object-orientation which is often skimmed over in texts is the containment relationships between objects. However, the various forms of aggregation (including containers) should perhaps be given equal status to the class relationships defined by a hierarchy. There are a number of reasons for this. First, the role of aggregations in providing interfaces to other objects (Booch's 'mechanisms') is an important part of the architecture of a program. Second, objects cannot change their class at run time, but they can change their roles in an aggregation. For example, a 'Checkout' object cannot become a 'Checkout With Weighing Scale' object if these are different classes. However,

Transactions on Information and Communications Technologies vol 7, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

a 'Checkout' object can dynamically aggregate a 'Weighing Scale' object at run time. In this respect, aggregation is more flexible than inheritance. Third, the delegation form of aggregation can overcome semantically problematical uses of inheritance where private derivation is used to implement classes in C++ which are not truly 'a kind of' their base classes. Finally, it may be a tool for portable systems by simplifying the representation of components of an object [6].

7 Polymorphic polymorphism

Perhaps the most difficult thing to teach in object-orientation is polymorphism. Polymorphism (derived from the Greek 'polumorphos' - having many forms) is problematical because there are many forms of polymorphism (hence polymorphism is polymorphic!) Although it is easy to make generalisations about the role of polymorphism ('passing responsibility for interpreting a message to the object', 'sending a generic message to a heterogeneous collection of objects' etc.) it is perhaps harder to explain all the different ways in which this may be implemented. As Cardelli and Wegner indicate in their taxonomy [7], there are various categories under which polymorphism can appear (figure 1), though not all of these are particularly object-oriented - ad hoc polymorphism does not have to be used as part of an object-oriented program.

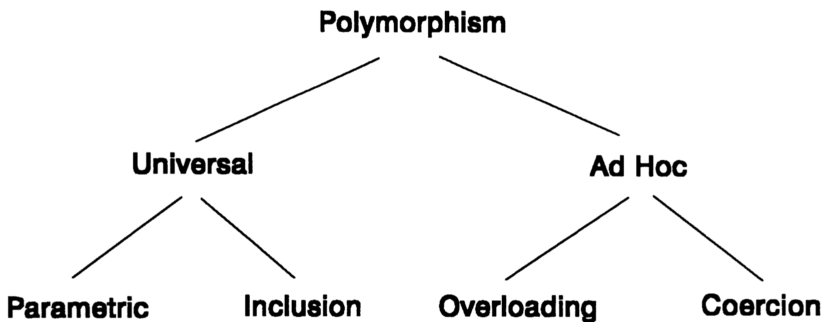

Figure 1: Cardelli and Wegner's taxonomy of polymorphic techniques.

In approaching polymorphism in C++, we need to explain a range of different aspects which can be categorised under the general headings of 'method polymorphism' (object methods exhibiting polymorphic behaviour according to the class of the object receiving the message) and 'polymorphism by parameter' (polymorphic behaviour according to the types or classes of message parameters). In terms of method polymorphism, in C++ we have

both function overloading (in a classification hierarchy this is 'inheritance' or 'inclusion' polymorphism) and operator overloading, though both of these may be regarded as forms of polymorphic object method because they are used in semantically similar ways. In addition, the crucial differences between static and dynamic binding have to be addressed for all types of object method. In polymorphism by parameter, there is an important distinction to be drawn between overloading (object methods made polymorphic by differences in parameter lists) and genericity (classes or methods made to handle objects with polymorphic behaviours). Although both of these techniques rely on parameter lists for their polymorphic behaviour, they have very different roles to play. Thus a number of different types of polymorphism can be applied to an object-oriented C++ program, not all of which fit into the generalised descriptions of what polymorphism means. Figure 2 shows a number of different areas in which polymorphism may be explored, again beginning with the more assimilable forms (simple overloading of in-class methods for example) and working through to the more subtle (dynamic binding for run-time polymorphism).

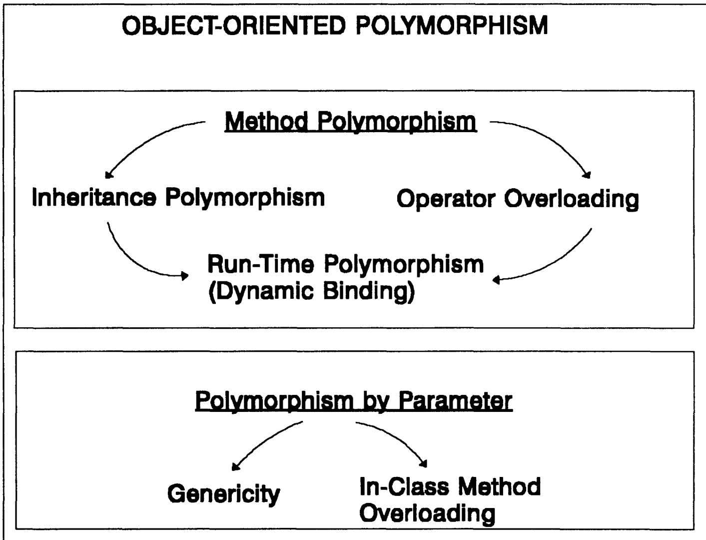

Figure 2: Approaching aspects of polymorphism with the facilities of C++.

Finally, students must learn to create and use containers in order to manage their dynamic objects, and to understand the trade off between genericity and functionality. C++ templates are a useful tool here, since they demand a full understanding of how generic messages must be intercepted by objects of unpredictable classes, giving a context for techniques such as operator overloading (particularly the relational operators) and dynamic binding (virtual functions in C++).

Beyond polymorphism, other aspects of the paradigm can be seen as merely extensions of the key elements. Multiple inheritance is useful, but only on occasion. Object persistence, though vital to any realistic application, is an implementation detail; ideally it will be a transparent process via an object-oriented database. Though students can be taught how to stream their objects to and from files, the detail is less important than the idea that objects can persist between program runs and different applications.

8 Summary

In general, the incremental approach can be summed up as follows:

1. Students should be made aware of the links between known and new techniques

2. Object-based and class-based programs are the best first step to object-orientation

3. Hybrid programs can be successful object-oriented programs, placing the stress on message passing and object lifetimes rather than on classification hierarchies and polymorphism

4. The static class relationships of inheritance are important, but should be balanced by the dynamic object relationships of aggregations and associations

5. Polymorphism has many facets, some of which are simpler to implement and understand than others. Dynamic binding is the final piece of a complex jigsaw which may be built up piece by piece from simple ad hoc beginnings such as overloaded functions.

9 Results and conclusion

How successful, then, is this approach in conveying the object-oriented paradigm to students? This can be informally evaluated to some extent by the assignment work completed by the students being taught using this incremental approach compared with work from a similar group from the previous year. With the previous group, many of the key concepts had been introduced simultaneously (albeit using C++) with graphics and class libraries. This approach was based on the common assumption that graphics objects are the simplest way to demonstrate object-orientation in practice because the hierarchies (e.g. 'line' is a kind of 'graphics object') and polymorphic responses to generic messages (such as 'draw') are allegedly obvious. In practice the students' work and individual responses suggested that more object-oriented concepts could be successfully delivered (ie actually understood

and used by the students) via the incremental approach, though it took longer to apparently show results. It seems that the greater understanding of the fundamental techniques provided by a step by step approach eventually gave the students more confidence to write truly object-oriented programs, whereas the group which began by being immersed in object-orientation in all its aspects seemed to reach something of a wall where they lacked 'ownership' of the software and found complex structures such as generic containers almost impossible to implement. Although there is a case for saying that there is no need to implement complex structures when we should be reusing those which already exist, there is also a case for saying that students should fully understand everything they do. We can train people to make certain things happen, but education is about understanding why those things happen.

In conclusion, we may make a case for a pragmatic approach to the teaching of object-oriented programming on the grounds that reuse is the ultimate aim; Reuse not only of software components but more importantly of students' experience and understanding. The incremental path reuses existing knowledge to build new classes of understanding.

References

1. Meyer, B. in Watts, W. *La résistance*, EXE magazineVol 6, Issue 11, May 1992 p.29.

2. Stroustrup, B. in Watts. W. *Yet More Bjarne*, EXE magazine, Vol 6, Issue 9, March 1992 p.32.

3. Booch, G. *Object-Oriented Analysis and Design With Applications* (2nd edition), Benjamin/Cummings, California, 1994.

4. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen W. *Object-Oriented Modeling and Design*, Prentice Hall, New Jersey, 1991.

5. Wirth, N. *OOP meets Modula-2*, EXE magazine, Vol 4, Issue 11, May 1990, pp.12-16.

6. Corbett, E. *A Framework for Application Class Design*, EXE Magazine, Vol 7, Issue 10, April 1993, pp.12-16.

7. Blair, G., Gallagher, J., Hutchison, D. & Shepherd, D. *Object Oriented Languages, Systems and Applications*, Pitman, London, 1991 p.81.