Approaching Inheritance from a "Natural" Mathematical Perspective and from a Java Driven Viewpoint: a Comparative Review

Marc Conrad[1], Tim French[1], Carsten Maple[1], and Sandra Pott[2]

[1] University of Luton, LU1 3JU, UK
marc.conrad@luton.ac.uk, tim.french@luton.ac.uk,
carsten.maple@luton.ac.uk
[2] University of York, YO10 5DD, UK
sp23@york.ac.uk

Abstract. It is well-known that few object-oriented programming languages allow objects to change their nature at run-time. There have been a number of reasons presented for this, but it appears that their is a real need for matters to change. In this paper we discuss the need for object-oriented programming languages to reflect the dynamic nature of problems, particularly those arising in a mathematical context. It is from this context that we present a framework that realistically represents the dynamic and evolving characteristic of problems and algorithms.

1 Introduction

It is acknowledged that there is widespread use of the Java and C++ languages within undergraduate teaching of joint Honours degree programmes in Computer Science and Mathematics in UK Universities. This can be confirmed by conducting a review of materials available publically online. The results of such a survey, which includes well-established institutions as well as those formed more recently, are presented in Figure 1. The dominance of such languages is hardly surprising, given the close conceptual cross-fertilization between object-orientation and certain specific mathematical concepts such as abstraction, generalization, and overloading. Mathematicians are able through the use of Java or C++ to contextualize mathematical abstractions in software. Equally, Computer Scientists are able to explore fundamental mathematical constructs through an easily accessible and natural medium of expression i.e. a simple Java program. Indeed, for these and other related reasons, the graduates of such joint honours degree programmes are much in demand from industry and commerce, where object-oriented methods and tools now predominate. On the face of it therefore we appear to have "a marriage made in heaven".

However, some concerns remain – and in particular this paper highlights how Java itself may be acting as a conceptual restraint for both mathematicians and computer scientists alike in certain specific areas of mutual concern. For example, it is well known that Java only supports a highly restrictive form

University	Language
Aston	Maple
Birmingham	Undefined/Unclear
Bristol	C++
Brunel	Generic OO
Cambridge University	Generic OO
Cardiff	Java
Coventry	Modula-2
Dundee	Java
Edinburgh	Java
Essex	Java
Hertfordshire	Undefined/Unclear
Imperial College	Undefined/Unclear
Keele	Undefined/Unclear
Kent	Java
Kings College London	Java
Kingston University	Java
Lancaster	Undefined/Unclear
Liverpool	Java
Loughborough	Undefined/Unclear
Manchester	Java
Manchester Metropolitan	Java
Oxford Brookes	C++
Oxford University	Generic OO
Queen Mary College, London	Java
Queens University Belfast	Modula-2
Sheffield Hallam	C++
Strathclyde	Undefined/Unclear
Surrey	Undefined/Unclear
Swansea	Undefined/Unclear
UCL	Java
University of Central Lancashire	Undefined/Unclear
University of Wales (Aberystwyth)	Java
University of Wales (Bangor)	Java
Westminster	Generic OO
York	Maple

Fig. 1. A snapshot survey of UCAS GG15 (BSc Joint Hons. in Mathematics and Computer Science) Entry 2003–4. Programming languages taught during 1st year. Source: published internet material in the public domain, August 2003.

of inheritance whereas mathematically other forms are commonly encountered (such as dynamic inheritance and method renaming). The aim of this paper is to examine these wider types of inheritance in some detail so as to encourage mathematicians and computer scientists alike to move their thinking beyond the limitations imposed by the standard Java programming environment itself (or in fact any other tangible programming environment).

It is of course quite possible to incorporate additional features into Java (or C++) through the use of various extensions such as the Darwin project [11]. Equally it is possible to develop customized software implementations that seek to explore mathematical structures in an overtly axiomatic manner such as com.perisic.ring [6] or Axiom [10]. We will show that the close conceptual and axiomatic links that exist between object orientation and mathematical structures may merit a closer self-examination of the choice of programming environment to be used in undergraduate Maths/Computer Science teaching. Whilst Java is undeniably industrially credible and popular, it is nevertheless useful to examine its limitations too. We go on to explore the limitations in respect of inheritance and proceed to suggest ways in which these limitations may be overcome through widening thinking processes, and/or by a process of extension through the creation of additions or tailored development environments.

2 The framework

In the context of Computer Algebra there are some packages that support the object oriented paradigm. For example Axiom [10] has type hierarchies ordered in an inheritance like structure and similarly Mupad [18] that explicitly enables the defining of child classes of existing classes as groups, fields, etc. Even mainstream mathematical software packages, such as Maple [24] and Mathematica [25], now reflect the significance of Java as a programming language by offering an interface between the package and Java applets and applications.

A radical approach to describing mathematical relationships within object oriented software can be identified in the experimental Java package com.perisic.ring [6]. It is focussed on an object oriented implementation of mathematical structures in an axiomatic manner. In the following we will take a closer look at aspects of the design ideas of this package as they serve well as a "reference model" for the discussion on non-standard inheritance relationships presented in Section 3. For details we refer the reader to the web site and documentation of [6].

The main philosophy of the com.perisic.ring package is that postulated properties of a domain are reflected as abstract methods of Java classes. For example an algebraic ring *has* by definition addition and multiplication. Of course, addition and multiplication are not known *algorithmically* for an arbitrary (unspecified) ring: they cannot be implemented.

Therefore the mathematical entity "algebraic ring" is implemented as an abstract class `Ring` with abstract methods as in

```
abstract public class Ring {
  abstract public RingElement add(RingElement,RingElement);
  abstract public RingElement mult(RingElement, RingElement);

...
}
```

Both methods have "ring elements" as arguments and return values. A `RingElement` is a class that can be (polymorphically) associated to any object (e.g. a multi-precision integer, or a vector of `RingElement` objects).

Not all methods need to be abstract. For instance a method `square` can be implemented via $a^2 = a \cdot a$ using the abstract multiplication method. More sophisticated non-abstract methods are exponentiation or evaluation of a polynomial.

Thus, an abstract Java class is able to mimic the axiomatic definition of a mathematical entity, here a "ring". The axioms *addition* and *multiplication* appear as abstract methods in `Ring` while *squaring* is not axiomatic and therefore may be implemented. Note that structural axioms such as *associativity* or *distributivity* cannot be declared as methods but rather have to be formulated as constraints.

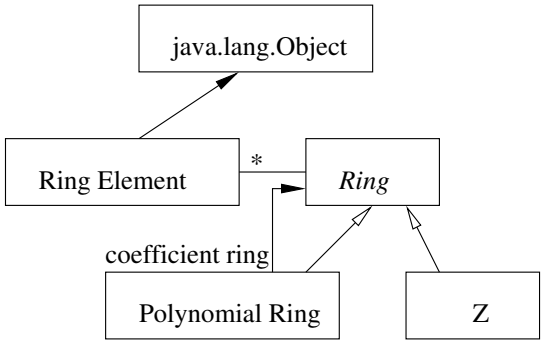

Fig. 2. A UML diagram of the implementation of polynomials over **Z** following the com.perisic.ring approach.

In Figure 2 we give an example illustrating the power of this concept by showing how multivariate polynomials arise naturally by a straightforward implementation of univariate polynomials.

The abstract class `Ring` bidirectionally links with many `RingElement` objects as described above. The class `Ring` has two child classes, namely the ring of integers **Z** and a univariate polynomial ring. Both of these rings implement addition and multiplication. For **Z** this is possible via built-in integer operations, whilst the polynomial ring implements addition and multiplication via the addition and multiplication methods of the coefficient ring. That

is, the polynomial ring is not only derived from the class `Ring` but also *uses* a `Ring` object in its role as a coefficient ring. Starting with an instantiation of $\mathbf{Z}$ we can then recursively instantiate polynomial ring objects as $\mathbf{Z}[a]$, $\mathbf{Z}[a][b]$, $\mathbf{Z}[a][b][x]$, and so forth.

In a similar way other structures can be implemented using the abstract class `Ring`. For instance the com.perisic.ring package supports quotient rings, modular rings, and universal rings. Obviously the design pattern can also be applied to groups, vectors spaces, and metric spaces, for example.

3 Non-standard features

3.1 Dynamic Inheritance

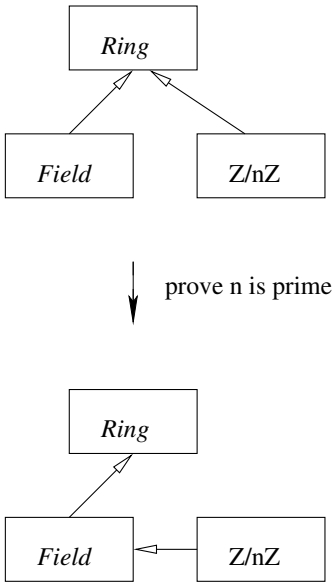

Fig. 3. When n is proven to be prime the inheritance relationship changes.

When implementing the mathematical entities "field" and "ring" as classes, it is straightforward to have them arranged in an inheritance relationship as shown in Figure 3. A field *is a* ring with additional properties (such as division). Some algorithms, for instance vector space computations, work only for fields and not rings thereby justifying an extra class "field", rather than introducing "fieldness" as a property of ring.

For any given ring object however we do not always know *a priori* – that is at compile time – if it is a field. For instance the ring $\mathbf{Z}/\mathbf{nZ}$ in Figure 3 is a field if and only n is a prime number. Although it has been recently shown that proof of

primality can be performed in polynomial time [2] it is not reasonable to assume that the developer of the class hierarchy or a compiler can predict whether or not a variable holds a prime number at a certain point of program execution of an program (actually this is a special variant of the halting problem). The most appropriate solution is to assume at the start of the program that $\mathbf{Z}/\mathbf{nZ}$ is a ring. When the primality of n has been proven the inheritance relationship should change, so that the $\mathbf{Z}/\mathbf{nZ}$ object, now an instance of a field, can for example be used in the context of vector space computations.

Of course we freely acknowledge that Dynamic Inheritance is hardly a "new" feature. A C++ implementation (or rather workaround) is already discussed in [7]. Also strongly related is the concept of *predicate classes* [4] that is rooted in Cecil [5] but in essence is language independent. In [14] a Java extension featuring Dynamic Inheritance is proposed. Dynamic Inheritance is supported in Lava as part of the Darwin project [11].

Although our focus is on mainstream languages such as Java, we briefly discuss other concepts dealing with dynamic inheritance in order to get a better idea how such features could be incorporated into Java, namely Self [22] and *Fickle* [8].

Self [22] emphasizes an object driven approach rather than a class/instance driven approach. New objects are generated by copying existing objects and modifying them. This increases the flexibility of objects since they can act more independently than in a more rigid class driven paradigm. One consequence is that an object can flexibly add and remove parent objects any time during program execution.

From a mathematical point of view it makes sense, in certain cases, for an object to be able to change its ancestors. For instance it is well known that the ring $\mathbf{Z}[i]$ is Euclidian, while most other quadratic orders $\mathbf{Z}[\sqrt{d}]$ are not. Therefore, from an object-oriented point of view, the information *Euclidian* is located in the ring object itself and not in the class. The Expert design pattern [16] suggests that the object should be able to change its parent class (from Ring to Euclidian Ring).

From the point of flexibility, languages such as Self are optimal and below we further discuss how such flexibility can be transferred into a mainstream language such as Java. Mathematical intuition, i.e. examining this issue from a formal conceptual perspective, suggests that we should not *give up* on classes too easily. In fact the ideal situation would be that $\mathbf{Z}[i]$ is a child object of a Euclidian ring whilst still being of class *quadratic order*. For example, with reference to the earlier example of modular integer rings, $\mathbf{Z}/13\mathbf{Z}$ is and should remain of type modular integer rings, i.e. an instance of a class $\mathbf{Z}/n\mathbf{Z}$, instantiated with $n = 13$. After proof of primality, its parent class will be changed to *Field*, but it will not cease to be a Modular Ring.

A possible compromise solution is to introduce a class *Finite Prime Field* that is a child class of *Modular Integer Ring* which implements a *Field* interface (we have to abandon the idea of an abstract *class* Field, as Java does not support multiple inheritance). Then the task is "simply" to *reclassify* at runtime the

$\mathbf{Z}/13\mathbf{Z}$ object from *Modular Integer Ring* to *Finite Prime Field*. Reclassification is a concept both discussed and directly implemented in *Fickle*.

The general idea is this: An object is related to a *Root Class* (in this case the class *Ring*). Then it can be reclassified to each child class (called *State Classes*) of this Root class. A special operator *!!* in *Fickle* reclassifies an object from one State Class to another when both belong to the same Root Class. Please see [8] for further details.

The translation of *Fickle* into Java as described in [1] is conceptually interesting. Here, each *Fickle* object is translated in Java to a pair $< id, imp >$ where *id* is the *identity* of an object and *imp* is the *implementation* of the object. Whilst the *identity* remains unchanged for a particular object, the *implementation* has the class information and can thus change. From this concept we can deduce the feasibility of dynamic inheritance in Java. From a pragmatic viewpoint the resulting Java code is complex, difficult to follow, and counterintuitive, to say the least. For instance a method implying a possible change of class has to be translated into a pair (a static and a member method). Calls of member methods in *Fickle* are translated into calls of static methods with an object as an argument (this is quite similar to C workarounds for object oriented programming).

Nevertheless the *Fickle* idea can serve as a roadmap for an implementation of reclassification in Java directly. An object that is to be reclassified is represented as a pair of an identity and an object of the current class. That means, from a user perspective that the only additions to the Java language are:

- a keyword, say `dynamic`, in connection with the `new` keyword, indicating that a new object can be reclassified;
- a method `reclassify` indicating a reclassification.

Figure 4 shows a code example illustrating the proposed syntax of Java reclassification.

The priorities for the Java Developer Team for J2SE 1.5 are not in dynamic inheritance. However references to references that would allow an easier implementation of the *Fickle* concept seems at least to be recognized as a desired feature (and is already partly available in the java.lang.ref package) [3].

3.2 Dynamic Generalization

Assume for the moment that Euclid were a contemporary mathematician who had just discovered the Euclidian division (also known as division with remainder), and that he wants to add Euclidian division into existing mathematical software that features a ring/field implementation as on the left hand side in Figure 5. The proper place for a Euclidian Ring – a ring with Euclidian division – is between the ring and field. Not every Ring is Euclidian and every field is trivially a Euclidian ring [15].

As this fictional example shows mathematical progress and invention does not only confine to deriving child classes from parent classes but also encompasses the invention or discovering of properties that do not apply to all entities (here:

```
public class MatrixRing {
    public Ring ringOfEntries;

    public MatrixRing( Ring theRingOfEntries) {
         ringOfEntries = theRingOfEntries;
         }

    public Solution []  solve(Equations [] eqs ) {
         if( ringOfEntries.getModule().isPrime() ) {
          // reclassify
          ringOfEntries.reclassify(
                Class.forName("com.perisic.FinitePrimeField"));
          // Use a library method to make a standard Gauss elimination
          Solution [] MatrixOverFieldUtilities.Gauss(eqs);
          }
         else {
          //  [...] do something more sophisticated, tailored for
          // Z/nZ (e.g. applying the Chinese Remainder Theorem
          // with componentwise solution).
          }
         }

public static void main (String [] args ) {

    // [...] enter module, we do not know if it is a prime

    Ring  ZModNZ = new dynamic ModularIntegerRing(module);
    MatrixRing MR = new MatrixRing( ZModNZ);

    Equations [] eqs = ... // generate a system of linear equations

    Solution [] result = MR.solve(eqs);

    // [...] print results
    }
}
```

Fig. 4. Example of the proposed syntax of Java reclassification. For determing the solution space of a system of linear equations over a field a standard Gauss elimination over fields can be used. We assume that this is provided by a class MatrixOverFieldUtilities. We further assume suitable definitions of the classes Ring, ModularIntegerRing, Equations and Solution. The keyword `dynamic` and the method `reclassify` are the proposed extensions of the Java language.

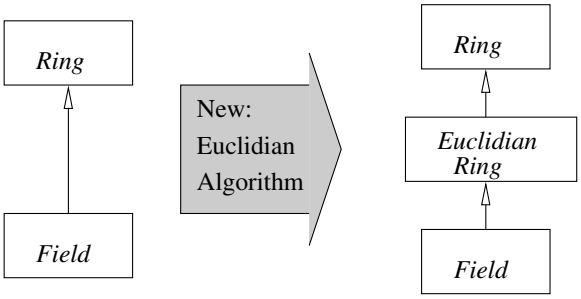

Fig. 5. The "new" Euclidian ring has its place in the middle of the hierarchy

Rings) and may be trivial for (but apply to) a subset of these entities (here: Fields). In object oriented terms that means that in mathematics it is natural for new classes to appear as a *generalization* of an existing class, or even as both a specialization and a generalization of existing classes. This process is known as *interclassing*. For a motivation of interclassing from a software engineering point of view see [9].

Although the imaginary scenario that Euclid invented Euclidian division after the invention of the concept of a ring and field may be somewhat artificial, defining new structures in the context of an existing hierarchy is a standard technique in contemporary mathematics. A typical example taken from Functional Analysis are the Triebel-Lizorkin spaces, which were introduced in the 1970's as simultaneous generalizations of a number of well-known classes of function spaces, e.g. L^p spaces, Hardy spaces, the space of functions of bounded mean oscillation (BMO), Lipschitz spaces and Sobolev spaces (see e.g. [23]). A Triebel-Lizorkin space is a specialization of a Banach function space. A more recent example are the so-called real Q-spaces, which are a simultaneous generalization of the space BMO and certain other Banach function spaces [12].

This "interclassing" in Mathematics is often motivated by the desire to create a unifying framework for several known classes of mathematical objects in a certain context (as in the first of the two examples mentioned above), or to bring existing mathematical techniques to new applications (in the second example).

Note that the problem of interclassing is substantially different from the problem of run-time reclassification described earlier in section 3.1. Here, we start with a class hierarchy that may be arranged in a package and that may not even contain any source code. We want to extend this class hierarchy by adding a class on a well defined position in an inheritance tree. Even if the source code is available it may be not desirable to change this code, especially if the class library is well established and the addition of the new class has *experimental* character, or is only relevant for a specialized application area.

Outside of a mathematical context the idea of *interclassing* is already discussed in [21]. In [9] an implementation is described using the OFL model. However, in terms of pragmatic usage OFL is inadequate as it requires *de facto* to

learn OFL as an additional language, namely the understanding of the correct use of hyper-generic parameters. Also, in using hyper-generic parameters the developer of a library already unnecessarily restricts possible extensions.

For these and other reasons we propose here a mechanism for interclassing at *run-time* an existing library that is conceptually simpler and is also proven to be workable in practice in the application area of computer games [19]. Incorporating this mechanism into Java requires the addition of three "easy" to understand keywords and two methods.

We will illustrate this by extending the earlier Euclidian Ring example: Imagine an inheritance hierarchy with a Ring as a parent class and two rings, say field and polynomial ring over real numbers as child classes, and we further assume that these classes are part of a library and cannot be changed simply by adding source code. However we *can* assume that the developer of the class hierarchy wants to allow interclassing, assuming that a mathematical developer knows that new structures are likely to be added.

Adding a new class *Euclidian Ring* in this hierarchy will be between the Ring class (as parent) and both the Field and Polynomial Ring (as child classes). Thus, at least in principle, both Field and Polynomial Ring, obtain additional behaviors. The obvious problem is, *how do the Field and Polynomial Ring "know" about this additional features?*. The solution is *shadowing* of objects and/or classes. Shadowing of an object means that each message sent to this object is "filtered" through a set of shadows. If the message is already understood in the shadow, it is executed in the shadow. If the message is not understood, then it is passed to the shadowed object. In addition, the shadow itself is able to send a message directly to the shadowed object. Shadowing is a language feature of LPC [19], a language designed for implementing MUDs (namely LPMuds). The shadow mechanism for objects is documented in [20]. As LPC is classless (using rather a prototype approach with cloneable objects as in Self), there is obviously no mechanism for class shadowing.

For our proposed Java extension we assume that shadowing a class means that a shadow is thrown (automatically) onto each object instantiated from this class. Also, LPC does not allow to shadow an inheritance relationship. Again, there is conceptionally no problem to allow this in our solution (compare e.g. with Self [22] where each parent slot is in fact a special kind of a data slot).

To add this feature to Java we propose the following concepts: First, a keyword `shadowable` gives a class the property that it can be shadowed. Internally each shadowable class and object maintains a list of its shadows.

The keyword `shadow` declares a class to be used as a shadow, and especially enables this class to use the keyword `shadowOwner`. The keyword `shadowOwner` refers to the shadowed object (similarly the keywords `this` and `super` refer to well defined, context-depended objects).

The member method `shadowedBy( Object ob);` actually shadows an object, whereas the static method `shadowedBy(Class c);` shadows a class. Both methods are available in any shadowable object/class (similar to `clone()` that is available in each cloneable object).

Figure 6 and 7 illustrate the usage of these keywords. Note that this syntax allows great flexibility whilst maintaining simplicity and usability of the language.

```
package example;

public abstract class Ring {
          // Ring methods here as in com.perisic.Ring
         }

// shadowable new keyword allows shadowing
public abstract shadowable class Field extends Ring {
         // Field specific methods here, e.g. inversion
         }

// A polynomial ring of one variable over a real numbers.
public  shadowable class PolynomialRingOverReals extends Ring {
         // Polynomial ring arithmetic implemented here
         }
```

Fig. 6. Declaration of a Ring/Field hierarchy using the new keyword `shadowable`

There seems to be no straightforward workaround for dynamic generalization at run-time other than changing the source code and recompiling. The main problem that any workaround faces is the inherent difficulty of informing an object (in this case a Field) to accept a message (in this case Euclidian Division) that hasn't been originally defined in a member method.

3.3 Overriding with Renaming

A group is a set with an operation and certain properties (the existence and uniqueness of a neutral element, associativity, etc). In general the operation is denoted by the symbol $\circ$ as in $c = a \circ b$. In a concrete situation there is often a standard notation for the group operation. The most familiar are $+$ for addition in an additive group and $*$ or $\times$ for multiplication in a multiplicative group.

For instance Figure 8 shows the multiplicative group $GL(2, \mathbf{Q})$ of invertible 2×2 matrices as a child class of an abstract group with an operation. The natural way to implement the group operation includes renaming the operation multiplication.

Renaming is hardly a new feature in object oriented contexts. In Eiffel renaming is the preferred method of choice to avoid ambiguity in multiple inheritance relationships [17]. Renaming exists also as standard feature in Python [13]. In contrast to the problems discussed earlier in section 3.1 and 3.2 this is not a sophisticated problem from a software engineering point of view. However adding

```
public abstract class EuclidianRing extends Ring {
     // Returns a pair (q,r) such as a * q + r = b.
     abstract RingElt [] EuclidianDivision(RingElt a, RingElt b);
     }

// The shadow for the field
public shadow FieldShadow extends EuclidianRing {
     RingElt [] EuclidianDivision(RingElt a, RingElt b) {
              RingElt [] result = new RingElt[2];
              RingElt result[1] = shadowOwner.zero(); // r = 0
              RingElt result[0] = shadowOwner.div(b,a) // q = b/a
              return result;
              }
// The shadow for the polynomial ring
public shadow PolynomialRingShadow extends EuclidianRing  {
     RingElt [] EuclidianDivision(RingElt a, RingElt b) {
             // Code for Euclidian division here
              }

public class TestStub {
      public static void main (String [] args ) {
              // shadowing all objects instantiated from this class
              example.Field.shadowedBy(Class.forName("FieldShadow"));
              // shadowing one object only.
              P = new PolynomialRingOverReals();
              P.shadowedBy( new PolynomialRingShadow());

              // ....
              }
      }
```

Fig. 7. An extension of the hierarchy of Figure 6 using an Euclidian Ring. The new method here is Euclidian division and made available for child classes

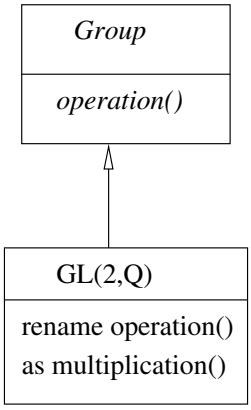

Fig. 8. $GL(2, \mathbf{Q})$ overrides operation() and renames it as multiplication().

this concept to Java would be an easy step to improve usability of Java within a mathematical context.

4 Conclusion

From the preceding discussion it is clear that a consideration of inheritance purely from a Java implementation perspective is a potentially conceptually self-limiting form of intellectual enquiry, though unquestionably highly pragmatic and vocationally useful. It is of course possible that mainstream languages will eventually evolve so as to directly support dynamic inheritance, generalization, and renaming or otherwise provide more explicit support for more abstract mathematical structures such as groups, rings, or vector spaces. However in the short-term it is perhaps more realistic to envisage that those seeking to explore the close conceptual cross-fertilization between pure mathematics and the object oriented paradigm will seek to supplement the standard Java "diet" through the creation of tailor made packages that adopt an overtly axiomatic vision. In any event it is clear that by only considering the types of inheritance that a particular programming language actually happens to support, students will fail to appreciate the wider (conceptual) picture and indeed arguably fail to gain their full maturity as mathematicians. Equally, those who consider their main interest to be in software design and development may have much to gain intellectually by expanding their knowledge of topics such as inheritance by exploring beyond the features that happen to be supported by any particular programming language.

References

1. D. Acnona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, E. Zucca. *A type preserving translation of Fickle into Java* Electronic Notes in Theoretical Computer Science 62 (2001). Available at: http://www.elsevier.nl/locate/entcs/volume62.html

2. Maindra Agrawal, Neeraj Kayal, Nitin Saxena. *PRIMES is in P*, http://www.cse.iitk.ac.in/news/primality.html, August 2002.
3. J. Bloch, N. Gafter, E. Ort (Moderator). *New Java Language Features in J2SE 1.5*, JavaLive Transcript, July 2003. Available from: http://developer.java.sun.com/developer/community/chat/JavaLive/2003/jl0729.html
4. C. Chambers. *Predicate classes*, in: *Proceedings of the ECOOP'93*, volume 707 of *Lecture Notes in Computer Science*, pages 268–296, Kaiserslautern, Germany, July 1993.
5. C. Chambers. *The Cecil Language: Specification & Rationale*, avialable at: http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html.
6. Marc Conrad. *com.perisic.ring – A Java package for multivariate polynomials*, http://ring.perisic.com.
7. James Coplien. *Advanced C++ programming styles and idioms*, Addison-Wesley 1992.
8. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini and P. Giannini. *Fickle: Dynamic object re-classification*, in: *ECOOP'01*, LNCS **2072** (2001), pp. 130–149.
9. Pierre Crescenzo, Philippe Lahire. *Using Both Specialisation and Generalisation in a Programming Language: Why and How?* Advances in Object-Oriented Information Sytems, OOIS 2002 Workshops, Montpellier, pages 64–73, September 2002.
10. Tim Daly et. al. *Axiom Computer Algebra System*, http://savannah.nongnu.org/projects/axiom.
11. *The Darwin Project*, http://javalab.iai.uni-bonn.de/research/darwin.
12. M. Essén, S Janson, L. Peng and J. Xiao, *Q-spaces of several real variables*, Indiana University Mathematics Journal, vol 49, no 2(2000), 575 – 615
13. Jeremy Hylton. *Introduction to Object-Oriented Programming in Python (Outline),* http://www.python.org/j̃eremy/tutorial/outline.html, Januar 2000.
14. Günter Kniesel. *Darwin & Lava - Object-based Dynamic Inheritance ... in Java*, Poster presentation at ECOOP 2002.
15. Serge Lang. *Algebra*, third ed., Addison-Wesley, 1993.
16. C. Larman. *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design*, Prentice Hall, 2001.
17. Betrand Meyer. *Overloading vs. Object Methodology*, Journal of Object-Oriented Programming, October/November 2001.
18. The MuPAD Research Group. *MuPAD – The Open Computer Algebra System,* http://www.mupad.de.
19. Lars Pensjö. *LPC*. Documentation available at: http://www.lysator.liu.se/mud/lpc.html
20. *Documentation of the Shadow function*, in: http://www.lysator.liu.se/mud/MudOS-doc/efuns/system/shadow.html
21. P. Rapicault, A. Napoli. *Evolution d'une hirarchie de classes par interclassement.* In: LMO'2001, Hermes Sc. Pub. *"L'objet"*, vol. 7 - no. 1–2/2001.
22. The Self Group. *Self*, http://research.sun.com/research/self
23. H. Triebel, *Theory of Function Spaces*, Monographs in Mathematics, vol 78, Birkhäuser Verlag Basel, 1983
24. Waterloo Maple Inc. *Maple* http://www.maplesoft.com
25. Wolfram Research. *Mathematica*, http://www.wolfram.com.