Exploring the synergies between the Object-Oriented paradigm and Mathematics: a Java led approach

MARC CONRAD
Department of Computing and Information Systems University of Luton
Park Square, Luton, LU1 3JU, United Kingdom
Email: Marc.Conrad@luton.ac.uk

TIM FRENCH
Department of Computing and Information Systems University of Luton
Park Square, Luton, LU1 3JU, United Kingdom
Email: Tim.French@luton.ac.uk

Abstract

Whilst the object oriented paradigm and its instantiation within programming languages such as Java has become a ubiquitous part of both the commercial and educational landscapes, its usage as a visualisation technique within Mathematics undergraduate programmes of study has perhaps been somewhat underestimated. By regarding the object oriented paradigm as a medium for conceptual exploration (rather than merely as a tool) the aim is to show how the close conceptual links between object orientation and certain mathematical structures such as rings and groups can be more fully realised, using a ready-made public-domain Java package.

1. Introduction

The object oriented (OO) paradigm has become synonymous with IT systems that exhibit reusability, modularity and scalability. Indeed, many within the IT industry regard OO methods of construction as being the *de facto* choice for information system deployment of all types and sizes (cf. the Object Management Group [1], Sun's Java [2], or Microsoft's C# [3]). Current UK University undergraduate programmes in Computer Science (as well as in the related disciplines of Computing, Informatics, and Software Engineering), increasingly reflect this ubiquity. Thus, students enrolled on programmes containing a significant component of Computer Science typically encounter OO via exposure to methods such as the UML [4] and/or, through exposure to programming languages that support the OO paradigm, such as Java, J2ME, C++, C#, or Python. Such students also typically encounter OO in the context of studying Object Oriented Relational Database Management Systems, within E-enterprise n-tier architectures, or indeed within any number of the diverse topic areas that comprise a "modern" undergraduate Computing curriculum. Figure 1 shows in summative form, how modern curricula in the UK are informed by the OO paradigm at 1st year Undergraduate level. The on-line survey specifically addressed students studying Computer Science/Mathematics undergraduate degree programmes and it is this group that forms the principal target audience for the approach advocated here.

Given the ubiquity of the OO paradigm, within both pedagogic and vocational settings, it would seem foolish not to seek to fully exploit the potential of OO as an educational medium for the visualisation of abstraction. With this in mind, we go on to suggest ways in which the OO paradigm can be used to actively support and

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

Figure 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.

reinforce student conceptualisation and visualisation of some specific "pure" mathematical structures such as groups, rings, vector spaces, etc.

It is perhaps reasonable to assume that our intended target audience (e.g. 2nd year BSc students studying Computer Science & Mathematics) will have already previously

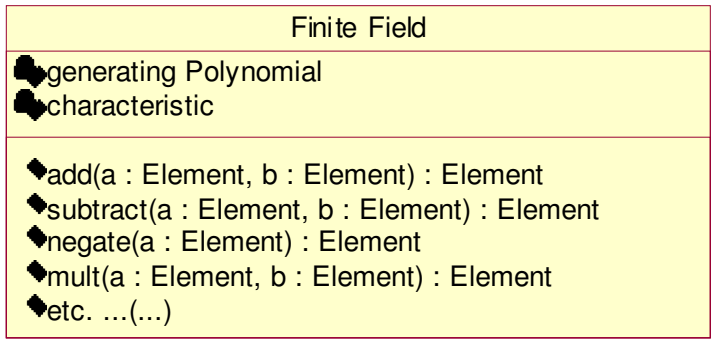

Figure 2: A class Finite field with the generating polynomial and the characteristic as attributes and the arithmetic as methods. We assume a polymorphic class Element

gained some exposure to OO in the context of either systems analysis and/or practical software development during their 1st year of study at University level.

The intention behind the approach offered here is to build upon these enabling foundations so as to reveal the fundamental *synergies* that exist between the various entities that are central to the OO paradigm (such as Abstract Data Types) and abstract mathematical structures (such as Rings and Groups). By inviting students to revisit certain fundamental OO constructs and techniques (such as classes, polymorphism, encapsulation and inheritance) within the context of pure mathematics, students may perhaps gain a deeper insight into mathematical abstractions and related algebraic structures. The idea is to generally reinforce students' learning and to generally enhance their mathematical maturity through a process of "live engagement" with a familiar and indeed ubiquitous OO paradigm.

The classical role of Computer Science (and Computer Algebra Systems in particular) within Mathematical education has been that of a pragmatic tool useful for reducing the amount of time students spend in "boring and repetitive drill exercises", and increasing the time given to more "interesting and motivating" aspects of mathematics [5]. In [6] for example, Java's role in Computer Algebra is mainly seen to be purely pragmatic: as a convenient means to create a graphical user interface or as a useful tool to explore distributed programming tasks. Our approach aims to go further. We attempt to show how gaining a deeper insight and understanding of object oriented techniques can provide an ideal opportunity for our target audience to better visualize and understand abstract mathematical concepts and entities.

2. Object Oriented Techniques and Mathematics

We start by reviewing some fundamental object oriented techniques using some mathematical examples. The aim is to introduce the reader who is not familiar with object oriented programming to the relevant concepts and also to show how mathematical structures are intimately linked to object oriented concepts. For a classical overview of object oriented concepts, unrelated to mathematics, the reader is referred to any of the various textbooks such as [7] or [8].

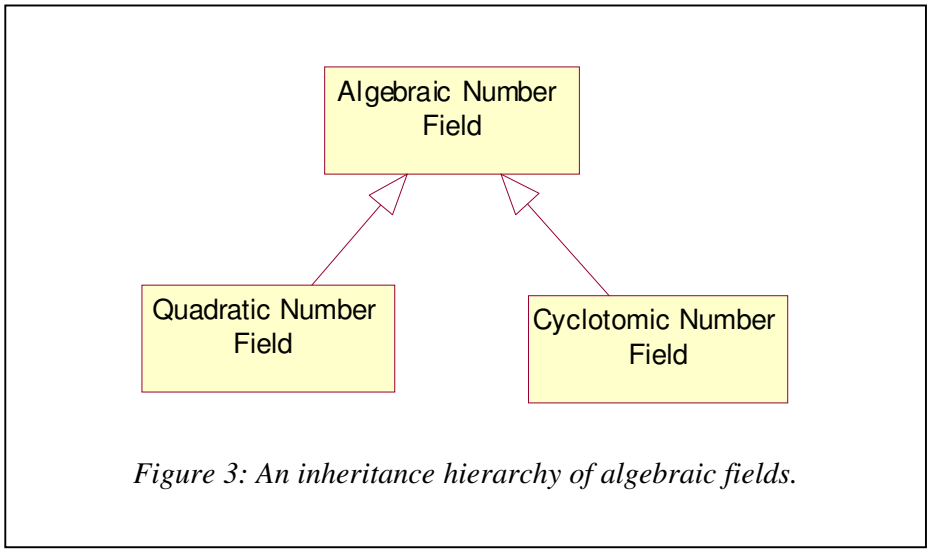

Figure 3: An inheritance hierarchy of algebraic fields.

An *object* is an entity that combines data and behaviour. For example, Figure 2 above shows the definition of a class *Finite Field.* The class defines Addition, Multiplication, etc. as behaviours. Here, the data are the *characteristic* and the *generating polynomial*. One way to structure classes is to use an inheritance hierarchy. An inheritance relationship, also known as a child-parent relationship, allows one class (the parent class) to provide all its methods to a second class (the child class). This is also known as *specialization* or an "*is a"* relationship. Figure 3 above shows how a simple inheritance relationship is defined between algebraic number fields. In this case the child classes comprise a cyclotomic number field and a quadratic number field. Using *"is a"* is usually a good rule of thumb to discover inheritance relationships: As a quadratic number field *is an* algebraic number field, it is natural that all methods implemented in the algebraic number field are available for the quadratic number field class.

Java and indeed other OO languages allow us to *reimplement* methods in a child class which have already been implemented in a base class. For instance from the relation $(a + b)^2 = a^2 + b^2$ *mod 2* it follows that squaring of polynomials in characteristic 2 can be performed in linear time. In an inheritance relationship of a class *Finite Field* and a class *Finite Field Characteristic 2* as shown in Figure 4, for performance reasons, we might want to use a different squaring method in characteristic 2 than in the parent class. Such a *reimplementation* of a method is known as *overriding* a method. The power of overriding is revealed in the context of *polymorphic* techniques i.e. in this particular example we could (generically) implement, say, a vector space over finite fields, using a reference variable to the *Finite Field* class of Figure 4.

When running the application we can initialise this reference variable with a *Finite Field Characteristic 2* object. In a programming language supporting dynamic linking (such as Java or C++ via the keyword virtual), the method of the object (Here: squaring in *Finite Field Characteristic 2*) is executed during program run-time execution.

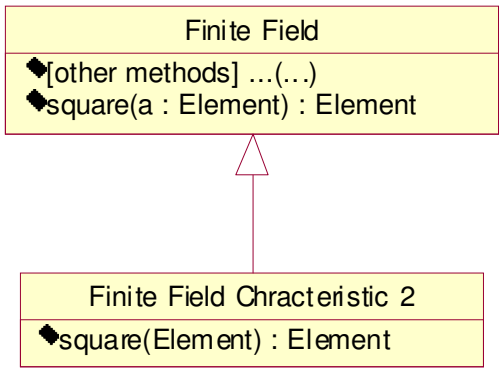

Figure 4: The method square() is reimplemented (overridden) in the child class Finite Field (for performance reasons)

The arithmetic of finite fields, number fields, etc. is explicitly known and can be implemented in a straightforward way. However, using an object oriented architecture we can allow our students to go further. We are able to implement classes where it is sufficient to know the *existence* of a method. These classes are called abstract classes and the methods are called abstract methods.

This is illustrated by considering the example of a class *Ring* that corresponds to an algebraic ring in mathematics. Because a ring has addition and multiplication, we know that there should be methods for addition and multiplication. These methods cannot be implemented, so they are abstract. In an OO language such as Java we can define them by using the keyword *abstract* as follows:

```
abstract public class Ring {
                                        ...
    abstract public RingElt add( RingElt a, RingElt b);
    abstract public RingElt mult( RingElt a, RingElt b);
                                        ...
    }
```

where the class *RingElt* is a polymorphic reference to the elements of a ring. Note that of course not all methods of an abstract class need to be abstract. For instance squaring could be implemented as:

```
public RingElt square(RingElt a) { return mult(a,a); }
```

The postulated properties of a domain are reflected as *abstract methods* in Java classes: i.e. 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. Thus, an abstract class is able to mimic the axiomatic definition of a mathematical entity, here a "Ring". As squaring is not an axiom, it can be implemented via the multiplication method. Note that structural axioms such as Associativity or Distributivity cannot be directly declared as methods, and have to be formulated as *constraints*.

Furthermore, abstract classes cannot be directly instantiated as objects. The following examples show the context where the *Ring* class may be used.

- A *Ring* may have child classes, for instance a class *Integer Ring* implementing the integers **Z**, or classes for the field of rational numbers, modular integers etc. To make these classes usable the abstract methods (as *add()* and *mult()*) need to be overridden. The implemented (non abstract) methods (as *square()*) are automatically available to the child classes.

- The class *Ring* is used via association or aggregation for the definition of other (abstract or non-abstract) classes. For instance we may implement a *Polynomial Ring* over a *Ring*. As a *Polynomial Ring* itself is a *Ring* we can instantiate a *Polynomial Ring* over a *Polynomial Ring,* leading to multivariate polynomials.

We conclude this section by discussing how the *RingElt* class is designed to represent ring elements. The design requirement of this class is that it is able to carry the data necessary for representing elements of any possible ring in the system (as integers or polynomials) . Object oriented languages offer various possibilities. A simple Java solution attributes *RingElt* with a reference variable to the Java base type *java.lang.Object*. The referenced object could then be of any type as Java *BigInteger* or an Array of *RingElt*s (for representing polynomials) etc. This approach is suitable and sufficient for *ad hoc* implementations. However, a more structured approach as supported by the *com.perisic.ring* package **[9]** makes *RingElt* itself an abstract class having the special designed classes *PolynomialRingElt* or *IntegerRingElt* as child classes. Note that the ring elements are implemented as immutable, which means they are constructed with a certain value, say an array of coefficients for polynomials, and never changed afterwards.

Method	**Documentation**	**Comments/Discussion**
RingElt add(RingElt a,RingElt b)	returns a+b	Methods for the basic arithmetic as discussed in section 2.
RingElt mult(RingElt a, RingElt b)	returns a*b	
RingElt neg(RingElt a)	returns –a	
RingElt zero()	returns the 0 of the ring (the neutral element of addition).	Each ring has an additive neutral element. In the design we assume that each Ring has a different zero element. (An alternative design may assume "null" as a zero representation common to all rings.)
boolean equalZero(RingElt a)	returns true if a = 0.	The basic arithmetic operations are nc sufficient to decide if two elements a and b are the same. Note that equality on the Object level (two object references are equal if they refer to th same memory location, i.e. to the same Object) is not sufficient, as we can store the same numerical value in two different objects.

Table 1: Abstract methods of the com.perisic.ring.Ring class

Method	Documentation	Comments/Discussion
RingElt one()	returns the 1 of the ring (the multiplicative neutral element)	Not all rings are rings with one. Ring with one will override this method. If this method is not overridden it wil throw an Exception.
RingElt inv(RingElt a)	returns 1/a	Of course not every element of a Rin has a multiplicative inverse. The default behaviour of this method is t throw an Exception if a ≠ 1 and to return 1 for a = 1.
RingElt tdiv(RingElt a, RingElt b)	returns an element x such that b * x = a.	tdiv stands for "true division" in orde to distinguish it from Euclidian division. Note that a ring that is not Euclidian still might be able to implement true division for certain elements, e.g. the ring **Z**[x] has no Euclidian division but we can still compute $(4x^2 - 1)/(2x - 1)=2x + 1$.
RingElt ediv(RingElt a, RingElt b)	returns q such that a = q * b + r (Euclidian division)	
RingElt mod(RingElt a, RingElt b)	returns r such that a = q * b + r (Euclidian division)	
boolean isField()	true if and only if the Ring is a field.	Each field is Euclidian and each Euclidian ring is a UFD. This is reflected in the default implementation, e.g. *isEuclidian()* is implemented to return *true* if *isField(* is true and to return false otherwise. Therefore a field only needs to override *isField()*.
boolean isEuclidian()	true if and only if the Ring is Euclidian.	
boolean isUFD()	true if and only if the Ring is a unique factorisation domain (UFD).	
RingElt gcd(RingElt a, RingElt b)	returns the greatest common divisior	Implemented using *mod()*. That mean this method is not available in UFDs that are not Euclidian. These rings have to override this method if they want to provide a gcd().
RingElt map(int k) *RingElt map(BigInteger k)*	returns k as an element of this ring.	Implemented for rings with 1 via 1+1+...+1 (k times).
RingElt map(RingElt a)	returns a as an element of this ring.	Here a belongs to another Ring objec than this. This method is implemente in the case that *a* is an integer via addition 1+1+..+1 (*a* times) and in th case that *a* is a fraction *p/q* via *div()*.
RingElt map(java.lang.String str)	returns str as an element of this ring.	The method responsible for input of ring elements. Most child classes wil override this method using a parser that is appropriate for this specific ring. By default we parse for integers and fractions.
RingElt map(java.lang.Object ob)	maps the Object as an element of this ring.	Works for classes java.math.BigInteger, java.lang.Ring and RingElt. Child classes may override this for other objects, e.g. matrices could accept two dimensiona arrays as parameter.

Table 2: Semi-abstract methods of the com.perisic.ring.Ring class

Method	Documentation	Comments/Discussion
RingElt sub(RingElt a, RingElt b)	returns a – b	Implemented as a + (-b)
RingElt div(RingElt a, RingElt b)	returns a /b	Implemented as a * (1/b)
boolean equal(RingElt a, RingElt b)	true if and only if a = b	Implemented as a – b = 0.
RingElt pow(RingElt a, int k) *RingElt pow(RingElt a, java.math.BigInteger k)*	returns a^k	Implemented using multiplication.
RingElt evaluatePolynomial(RingElt p, RingElt a)	returns the evaluation of the polynomial p at the element a.	Implemented using basic arithmetic.

Table 3:Implemented methods of the com.perisic.ring.Ring class

3. Example: The Ring class of com.perisic.ring

In order to gain a better understanding of the implementation of an abstract mathematical entity as an abstract class we now take a closer look at the implementation of the abstract class *Ring* within the com.perisic.ring Java package [9]. This package also contains child classes of the *Ring* class and an implementation of the *RingElt* class and is available free for educational purposes. The package has been designed so as to be suitable for use within a practical session to accompany a lecture on a relevant topic area as typically covered within Mathematics and Computer Science undergraduate curricula. The *Ring* class is in fact intended to serve as a reference implementation (exemplar) for other abstract structures such as groups, vector spaces, metric spaces etc. While syntactically we can distinguish between only abstract and non-abstract classes this distinction is in practice semantically too restrictive, as we encounter methods that *have to be overridden* in *some cases* only. We call these methods *semi-abstract*. In Java, semi-abstract methods are implemented as normal (non-abstract) methods.

Tables 1-3 give an overview of all the methods that are presently available in the *Ring* class. The first column contains the method names and the second column presents some necessarily brief documentation of the method. The third column containing explanatory remarks seeks to emphasise the conceptual underpinnings that underlie the method in the context of object oriented design and mathematics.

We conclude this section with a short overview of some of the child classes of the *Ring* class. The four classes in Table 4 below are to be instantiated with exactly one object (There is only *one* ring of integers, field of rationals, etc). This object can be directly referenced as a member of the *Ring* class.

Class name	Description	Referenced as
IntegerRing	Implements the integers **Z** via java.lang.BigInteger	*Ring.Z*
DoubleField	The "field" of real numbers **R** approximated by the Java double type	*Ring.R*
F2Field	The field of 2 elements $\mathbf{F}_2$. This is implemented as a wrapper of the type boolean.	*Ring.F2*
RationalField	The rational numbers **Q**.	*Ring.Q*

Table 4:Child classes of com.perisic.ring.Ring with exactly one object instance.

Class name	Description
PolynomialRing	Implements polynomials *R[x]* (where x can be any string).
ModularRing	Implements *R/fR*. Here *R* is a polynomial ring and *f* a polynomial. It is useful for instantiating algebraic extensions as objects. The com.perisic.ring package features also a subclass *CyclotomicField* for the implementation of cyclotomic fields (an algebraic extension of **Q** by an unit root) showing how the *ModularRing* class can be used..
QuotientField	The field of fractions *p/q* with $p,q \in R,\ q \neq 0$. Useful for rational function fields when *R* is a *Polynomial Ring*.

Table 5: Child classes of com.perisic.ring.Ring that are quipped with a reference attribute to a Ring object R.

The objects of the classes in Table 5 are constructed with a reference attribute to a *Ring* object *R*.

4. Examples of student activities

The package ***com.perisic.ring*** may be used as a useful reference model for the visualization of abstract mathematical entities (in this case an algebraic ring) in the form of abstract software classes. These abstract classes can made workable by deriving child classes and through the use of overriding methods. In the following two sections we attempt to give some ideas which might be helpful for developing practical sessions or projects for mathematical students using Java.

4.1 Developing abstract classes

- In a similar manner to the *Ring* class implemented within the package as previously described, an abstract class *Group* can be developed and evaluated. Here the abstract methods are the operation and the neutral element. Exponentiation can be implemented using several varieties of techniques found in literature such as [10]. For example some child classes of relevance to building 'real' applications could be $(\mathbf{Z}/n\mathbf{Z})^*$ and the group of an elliptic curve. Advanced students could in this way be introduced to cryptographic algorithms that can be implemented for abstract (abelian) groups (cf. [11]).
- A metric space could be implemented as an abstract class *MetricSpace* with an abstract method *distance()*. Then methods operating on a set of points such as computing the closest two points can be implemented via the *distance()* method, similar as the *square()* method is implemented via *mult()* for *Ring*.. Some suitable child classes might include: points on a plane, a three dimensional space, $\mathbf{F}_2^n$, etc
- Other mathematical structures that could be visualized as abstract classes in object oriented software include vector spaces, topological spaces, monoids, etc.

4.2 Extending the com.perisic.ring package

- Here our prime focus of attention will be the direct child classes of the *Ring* class. Small size projects could comprise a field of 4 or 9 elements, a wrapper class of Java's *BigDecimal*, or algebraic orders as $\mathbf{Z}[\sqrt{2}]$.

- Algebraic extensions similar to the *CyclotomicField* class and *UniversalCyclotomicField* classes can be developed via the *ModularRing* class. The easiest example here being quadratic extensions. More advanced usage might include the implementation of infinite extensions, say the extension of **Q** by all square roots of primes.
- Finite Fields as child classes of *ModularRing.* An interesting and challenging point here is the discussion of the uniqueness of finite fields: Each irreducible polynomial of a given degree leads to a "different" finite field, however they are all isomorphic.
- Using an abstract class (finite, abelian) *Group* and *Ring* to implement a group ring.
- Using the abstract class *Ring* for implementing a vector space over arbitrary fields.

5. Conclusions

We have identified and evaluated a fundamental relationship between object oriented techniques and Mathematical entities. Especially the concept of abstract classes can successfully be used as a visualization of abstract mathematical concepts such as "Ring", "Group", "Metric Space", etc. The ubiquity of object oriented languages in undergraduate education strongly suggests to use these relationships to improve student conception in Computer Science/Mathematics undergraduate degree programs. As we have shown student activities could then include extensions of the Java package com.perisic.ring in the context of algebraic rings or the development of similar software for the implementation of other algebraic structures using com.perisic.ring as a reference model.

References

[1] OMG - The Object Management Group*, http://www.omg.org.*
[2] Java, *http://java.sun.com.*
[3] C#, *http://msdn.microsoft.com/vcsharp/.*
[4] UML – The unified modeling language, *http://www.omg.org/uml.*
[5] *L. Bernardin, B. Char, E. Kaltofen,* 1999, Symbolic Computation in Java: an Appraisement*, Proceedings of ISSAC'99, ACM Press*.
[6] *C. Leinbach, D. C. Pountney, T. Etchells,* 2000. Appropriate use of a CAS in the teaching and learning of mathematics, *Int. J. Math. Educ. Sci. Tehcnol., 33, 1-14.*
[7] *T. Budd,* 1998, An Introduction to Object-Oriented Programming, *Addison-Wesley*.
[8] *E. Braude,* 2003, Software Design - From Programming to Architecture, *Wiley*.
[9] *M. Conrad,* 2002, com.perisic.ring, A Java package for abstract mathematics, http://ring.perisic.com.
[10] *H. Cohen,* 1993, A Course in Computational Algebraic Number Theory, *Springer*.
[11] *I. Blake, G. Seroussi, N. Smart,* 1999. Elliptic Curves in Cryptography. *Cambridge University Press.*