The Humble Fraction

Nigel Perry
Massey University
Palmerston North, New Zealand
<N.Perry@massey.ac.nz>

Abstract

In this paper we examine how trends in programming methodology support the definition of value types. We ask the question have we really advanced in the move from traditional to OO methodologies, have we produced square wheels, or have we just changed terminology? We ask this to determine what we should be teaching our students. The work was motivated by a concern that students, through what is, or is not, taught, are missing or rejecting the lessons of history and producing poorer designs because of this.

Introduction

Object-oriented techniques and languages have grown in popularity over recent years to become a key methodology, and are often seen as a universal panacea to the "software crisis" and promoted with evangelical zeal. It is indeed unusual to find a university today where OO is not taught as the primary or only style – modules have been replaced by classes, data structures by classes, and functions by methods. Moreover with the hype surrounding OO and each new development within it, students often arrive at university *knowing* they should be taught C++, or now Java, and express this view quite forcibly. This has further accelerated the change in what is taught.

But what is the effect of this change on our students and the development of their critical thinking skills? How often does a student when asked if some method/language is good simply respond with "yes, its OO"? Is OO the universal panacea it is claimed to be, or have we sometimes been teaching the design of square wheels which are deemed good by nature of being objects? This work was motivated by a concern that students, through what is, or is not, taught, are missing, or on principle rejecting, the lessons of computing history. This in turn is leading to poorer design skills and lower levels of understanding.

In this paper we use fractions to examine how trends in programming methodology support the definition of value types. Fractions are a simple data type and were chosen for their familiarity – it must be a rare programmer indeed who has never had a maths lesson at some stage on fractions! They are an example of what we term value types; other examples include integers, magnitude & direction vectors, playing card suits, and time durations. Programming languages usually support a number of standard value types; such as integer, floating point, and boolean. However programming languages cannot provide all of the many different value types used in different applications, so as for other kinds of type the ability to defined new ones easily and naturally is important.

In our investigation we ask the question have we really advanced, have we produced square wheels, or have we just changed terminology? What should we be teaching our students?

Value Types In Programming Languages

For the purposes of this paper we define two broad categories of types; *value* and *state*. A value type is one which is naturally viewed as a set of values with functions which operate on values of the type to produce new values. Integers are an example of a value type. The standard addition function on integers maps two integer values to their sum, it does not in any sense change the value of either of its arguments. A state type is one which is naturally viewed as having some state together with functions which change that state. For example, a stack is a state type. The standard addition ("push") function on stacks changes a stack's value by the addition of a new element.

The standard numerical types in programming languages; such as integers, floating point numbers and booleans; are value types as are non-numerical ones; such as characters. There are of course an unlimited number of value types, some other examples are; currency quantities, time durations, magnitude/direction vectors, colours, and playing card suits.

We use fractions as a prototypical value type and evaluate how they can be defined and used in the "traditional" and object-oriented methodologies. For brevity we will only define the addition operation over fractions and omit much of the detail and standard optimisations. Fractions happen to be a numerical type, with obvious similarities to integers and floating point numbers, which raises the issue of the overloading of standard operator symbols. This is an accident of our choice and does not effect the general issues discussed and we comment only briefly on it.

Traditional and Object-oriented Abstract Data Types

The traditional ADT methodology is based on the concept of types as sets of values together with a collection of functions which define mappings between elements of the set. Under

this methodology the types required to solve a problem are identified along with the functions that need to be perform``ed on those types. The values are passive participants in the operations of the functions. The language features developed to support programmer-defined types include records, modules and information hiding. This methodology, along with *structured programming*, was standard in industry and education until object-orientation became popular. We show how fractions are defined using this methodology in Design 1 below.

During the 80's the computing industry turned to object-oriented programming (and OO software engineering in general) as the panacea to the software crisis, and in universities C++ slowly replaced C, Pascal and other languages as the core teaching language. In the last few years Java, arguably a much better language than C++, has help fuel the wholesale switch to OO.

The object-oriented methodology can be defined ([1, 5, 7]) as a method in which:

- An object is characterized by a number if operations and a state which remembers the effect of these operations; that is an object is an active entity which posses a changing value.
- Objects are the fundamental building blocks
- Each object is an instance of some type/class

Put another way, the OO methodology is based on simulation and anthropomorphism where active objects replace the passive values of the traditional ADT. Designs 2 and 3 below show the standard OO solutions for value types, and as will be seen they are lacking when compared to the traditional ADT of Design 1.

However, OO languages have developed over time, and this opens up possibilities to improve on the representation of value types. Designs 4 and 5 below suggest a possible approach.

We now present the five designs for fractions along with some brief notes. We then discuss the designs and the questions posed in the introduction.

Design 1: "Traditional" Abstract Data Types

The following code shows a partial implementation of a traditional ADT fraction type in Ada.

```
package Fract is

   type Fraction is private;

   function val (a, b : Integer) return Fraction;
   function add (a, b : Fraction) return Fraction;

   procedure Put(a : Fraction);

private

   type Fraction is record
            num   : Integer;
            denom : Integer;
         end record;

end Fract;

package body Fract is

   -- reduce 2/4 to 1/2 etc..
   procedure normalise(f : in out Fraction) is ...

   function val (a, b : Integer) return Fraction is ...

   function add (a, b : Fraction) return Fraction is
      ans : Fraction;
   begin
      ans.num := a.num * b.denom + a.denom * b.num;
      ans.denom := a.denom * b.denom;
      normalise(ans);
      return ans;
   end;

end Fract;

// sample use
procedure trial is
   f, g, h : Fraction;
begin
   f := Fract.val(2,5);
   g := Fract.add(f, Fract.val(1, 10));
   h := Fract.add(Fract.val(1, 10), f);

   Put("f = "); Put(f); New_Line;
   Put("g = "); Put(g); New_Line;
   Put("h = "); Put(h); New_Line;

end trial;
```

Examining this example we note:

- The form of the function calls which use fractions follows that for language-defined types (we defer comment on operator notation till after Design 2). Whether a value type is programmer or language-defined makes no difference.
- In the definition of fraction addition the two operands have equal status. The function is clearly one of two operands, both of which are referred to in the same manner.

As the example shows, the methodology supports programmer-defined value types naturally and consistently with those defined by the language.

Design 2: Immutable Objects

The immutable object, that is one whose state is set once at creation, has been the standard OO approach for value types since Smalltalk. The following code shows a partial fraction package written using immutable objects.

```
public class FractObj
{  private int n, d;

   private static int gcd(int a, int b) ...

   public FractObj() ...
```

```
    public FractObj(int n, int d) ...

    public String toString()
    {  return n + "/" + d;
    }

    // disguise the new operation - looks more like a value
    public static FractObj val(int n, int d)
    {  return new FractObj(n, d);
    }

    public FractObj add(FractObj b)
    {  int n, d;

       n = this.n * b.d + b.n * this.d;
       d = this.d * b.d;
       return new FractObj(n, d);
    }
}

// sample use
public static void main()
{  FractObj f, g, h;

    f = FractObj.val(2, 5);
    g = f.add(FractObj.val(1, 10));
    h = FractObj.val(1, 10).add(f);

    System.out.print("f = "); System.out.println(f);
    System.out.print("g = "); System.out.println(g);
    System.out.print("h = "); System.out.println(h);
}
```

This design has a number of drawbacks, including:

- The use of fraction values isn't very natural. The expression `a.add(b)` reads naturally as "add b to a" not as "add a and b to give a new value", as would the expression `add(a, b)`.
- The definition of fraction operations is not natural.

In the definition of `add` the two operands are referred to differently, the operation does not look like an operation on two operands. Though operations are defined less often than they are used, this issue is still important. (By renaming the local variables the use of "`this.`" can be avoided, however the operands would still be referred to differently.)

Note on Programmer-defined Operators

Many languages allow standard operator (infix function) symbols to be used for programmer-defined types. They can be used to disguise the unnatural form of expressions which use immutable object value types by enabling `a.op(b)` to be written as `a op b`. However this only works for those types for which infix operator notation itself is natural, and it does nothing for the definition of the type; for this reason it is an orthogonal issue to that discussed here. They would be a worthwhile extension to Java, especially for users of numerical value types, and this has been proposed by others [4, 8].

Design 3: Static Methods

A standard variation in the design of OO value types is the use of *class* methods. These differ from instance methods in that they have no implicit object argument, but as part of the type have access to its internal structure. Class methods are found in most OO languages, for example the *static* methods of Java and C++. *Friend* methods in C++, defining two classes in the same Java package, or two types in the same Ada module, provide a similar feature.

A fragment of our fraction example written in Java using class methods is now shown.

```
public class FracBin
{  private int n, d;
   ...
   public static FracBin add(FracBin a, FracBin b)
   {  int n, d;

      n = a.n * b.d + b.n * a.d;
      d = a.d * b.d;
      return new FracBin(n, d);
   }
}

// sample usage
FracBin f, g, h;

f = FracBin.val(2, 5);
g = FracBin.add(f, FracBin.val(1, 10));
h = FracBin.add(FracBin.val(1, 10), f);
```

This method provides a "look and feel" for programmer-defined value types which is fairly close to those defined by the language. A confusion of the approach is that the methods visually appear to be part of the object instance, yet they are part of the object type. (For friends the methods are disassociated from the type.)

Design 4: Value and Behaviour Classes

The OO approach to *state* types has changed over time. One development is the use of two classes; one for storing most of the state/value, and another containing mostly operations; to produce a value/behaviour division. This design can be seen, for example, in Weiss' design for binary search trees [10].

Though examples do not abound the same general approach can adapted to value types. When applied to state types the division of value and behaviour is not complete and each distinct instance of the type has an instance of the behaviour class, which itself contains the instance(s) of the value class. For a natural model of value types we need only one instance of the behaviour class, which should contain no values at all. We of course need many instances of the value class.

In Java the method is supported through static classes, that is a class containing only static members, and packages. Our fraction example written using a value/behaviour division is as follows.

```
package Fractions;
```

```
public class Fraction
{  protected int n, d;

   protected static int gcd(int a, int b) ...

   protected Fraction() ...
   protected Fraction(int n, int d) ...

   public String toString() ...
}

public class FractOps
{  public static Fraction val(int n, int d) ...

   public static Fraction add(Fraction a, Fraction b)
   {  int n, d;

      n = a.n * b.d + b.n * a.d;
      d = a.d * b.d;
      return new Fraction(n, d);
   }
}

// sample usage
Fraction f, g, h;

f = FractOps.val(2, 5);
g = FractOps.add(f, FractOps.val(1, 10));
h = FractOps.add(FractOps.val(1, 10), f);
```

There is little against this method. For the client the values, variables and operations behave in the same way as for built-in value types. The addition of programmer-defined operators would remove any lingering distinction for the client.

For the implementor the approach is also natural; a type is defined to represent values in the required domain and functions to operate on these. A minor drawback is that the data type is defined by two separate entities, not one.

Design 5: Value/ Behaviour Nested Classes

Nested classes can also be used to produce a value/behaviour division, for example see [11]. They were introduced in Java 1.1 [6] and are provided in other languages.

Following the same approach of Design 4 the model can be adapted for value types with a static outer class containing the behaviour. This addresses the minor drawback of Design 4, allowing the type to be defined by a single class (albeit with a member class). The following code shows this approach applied to our fraction example.

```
public class FractInn
{  private static int gcd(int a, int b) ...

   public static class Var
   {  private int n, d;

      private Var() ...
      private Var(int n, int d) ...

      public String toString() ...
   }

   public static Var val(int n, int d)
   {  return new Var(n, d);
   }

   public static Var add(Var a, Var b)
   {  int n, d;

      n = a.n * b.d + b.n * a.d;
      d = a.d * b.d;
      return new Var(n, d);
   }
}

// sample usage
FractInn.Var f, g, h;

f = FractInn.val(2, 5);
g = FractInn.add(f, FractInn.val(1, 10));
h = FractInn.add(FractInn.val(1, 10), f);
```

Discussion

The traditional ADT approach, as shown in Design 1, fits value types well. It provides a natural form for both client and implementor, there are no hidden implicit arguments, values which are passive participants in a "real world" function do not get represented as active objects in the programming language model, and it provides all the desirable features of encapsulation and information hiding.

Turning to the object-oriented methodology there is an apparent mismatch between it and value types, as it is based on an active model of data as its fundamental building block.

An easy solution to this mismatch would be to ignore it, and this has often been chosen. Look at a CS2 textbook which uses a language with good support for the traditional ADT methodology, for example see [2] which uses Ada 95, and you are likely to find a number of programmer-defined value types covered. However pick a CS2 textbook which uses the OO methodology and the first data type covered will probably be some linear container type.

As shown in Design 2, it is hard to argue that the use of immutable objects is natural as the explanation of addition is "send the method add x to the object y". This is a confusing and forced model to teach to students (or use in industry).

Design 3 and the use of programmer-defined operators are an attempt to disguise the differences between language and programmer-defined value types. It has little to recommend it, lacks the clarity of Design 1, and goes against the OO philosophy as the operations are not characteristics of instances. The overuse of static methods in the OO method is often treated akin to the overuse of global variables in the traditional approach.

Designs 4 and 5 demonstrate that clear and natural approaches to value types are possible in OO languages.

However they do embody a significant shift in philosophy from that of "traditional object-orientation".

Design 4 provides the benefit lacking in Design 3, there is a clear distinction between the class which represents the value and the static class which provides the functions. Here Java's package system provides a similar benefit to the traditional module, as typified by the Ada package, by allowing dependant types/objects to be encapsulated within a single entity while providing appropriate restrictions on external clients. However, is a class which has no instances conformant with a methodology were *objects*, not classes, are the fundamental building blocks?

There is little to detract, from a client or implementor viewpoint, from Design 5 and its use of nested classes. But again is it *object-oriented*? Comparing the nested class solution to Design 1 shows that they are, apart from minor syntax differences, fundamentally the same. The behaviour class has replaced the module, the nested value class the type defined within the module. The process we have followed to discover a good method of supporting value types in OO languages has simply rediscovered the original ADT method.

It is now not hard to understand why texts which teach the OO methodology exclusively often ignore value types, they simply don't fit the model very well. But what does that teach students? Many arrive at our universities already persuaded that OO is the only approach, that older methods are out-dated and to be ignored. If we then teach the OO methodology exclusively, what happens when they are faced with a problem requiring value types? Having taught now for a number of years a graduate class attended by students who undertook their graduate studies at many different universities in different countries, the unfortunate answer is "square wheels" justified on the grounds that they are "object-oriented" and often additionally "that's what industry wants".

Would it not be better to tackle the mismatch of value types and OO methodology directly and teach students a more holistic approach which includes the clear separation of value and behaviour when appropriate, as well as their combination? In some texts, for example [10, 11], we see movement in this direction for state types but the development has gone largely unremarked. The use of behaviour-only classes of static methods is also seen for algorithms, such as sorting, in many texts. It is certainly disappointing that Java, a recent language, still uses the simple immutable objects of Design 2 for some of it's language-defined value types. For example the language-defined library types `BigInteger` and `BigDecimal` [9] follow this model and texts have followed this and used it for programmer-defined value types as well, for example see the complex numbers defined in [3].

Conclusion

It would seem that the humble fraction is not quite as humble as it first appears, it has resisted the might of the OO Borg and refused to be assimilated; our final "OO" solution turns out to be the original traditional one in a rather thin disguise.

The starting point for this research was concern over many of my students arriving in university, as undergraduates or graduates, already knowing that object-oriented programming was *the* best way of designing and implementing programs. Such students often demand, and are given, OO languages and OO methodologies. Older languages and methodologies are not favoured by them, and the OO texts available to them tend to skimp when covering value types, or present awkward solutions.

As Design 5 shows, an OO language such as Java is quite capable of supporting value types in a natural and straightforward manner; and of course broader languages such as Ada 95 also support both traditional and OO approaches. What is needed is a better blending in design methodologies of the traditional and the modern, so that our students both learn from the old and benefit from the new.

References

[1] Cardelli, L. and Wegner, P., On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys 17*, 4 (1985), 471-522.

[2] Feldman, M.B., *Software construction and data structures with Ada 95*. Addison-Wesley, 1997.

[3] Flanagan, D., *Java Examples in a Nutshell*. O'Reilly & Associates, 1997.

[4] Gosling, J., *The Evolution of Numerical Computing in Java*, Javasoft, URL <http://www.javasoft.com/people/jag/FP.html>, 1998.

[5] Horowitz, E., Sahni, S., and Mehta, D., *Fundamentals of Data Structures in C++*. Computer Science Press, 1995.

[6] Horstmann, C.S. and Cornell, G., *Core Java 1.1, vol. 1 - Fundamentals*. Sun Microsystems Press/Prentice Hall, 1997.

[7] Jacobson, I., *Object-Oriented Software Engineering*. ACM Press/Addison-Wesley, 1992.

[8] Steele Jr., G.L., Growing a Language. In *Proceedings of OOPSLA'98*, Vancouver, 1998.

[9] Sun Microsystems, *JDK™ 1.1 Documentation*, URL <http://java.sun.com/products/jdk/1.1/docs.html>, 1996.

[10] Weiss, M.A., *Data Structures & Algorithm Analysis in Java™*. Addison Wesley Longman, 1999.

[11] Winder, R. and Roberts, G., *Developing Java Software*. John Wiley & Sons, Chichester, 1998.