A Different Kind of Programming Languages Course

Dorian P. Yeager

Grove City College
Grove City, Pennsylvania, USA
dpyeager@gcc.edu

Abstract

The complexity of the well-explored regions of the programming language design space has increased substantially in the last twenty-five years with the addition of a large number of object-oriented programming languages (OOPLs). This design domain was already known to be large and complex before OOPLs came on the scene, and many offerings of the standard programming languages course before that time did not even mention the object-oriented programming (OOP) paradigm. Now that OOP has become mainstream, undergraduate programs which include a course on programming language design and implementation have responded by expanding their existing course or by jettisoning some of the alternative ideas in favor of expanded coverage of OOPLs. There are two facts, however, which must be confronted as we consider what information we would like to pass on to our students in this fascinating area of our discipline. The first is that the design space represented by OOPLs is large enough to justify a separate course, and the second is that a large number of ideas from other language paradigms appear in subsets of well-known languages for OOP. This paper presents a course in the design and implementation of programming languages that (as OOP itself did in the eighties) turns some accepted notions "inside out" by proposing that the entire course be presented from the OOP point of view. Such a course has been offered by Grove City College for a decade and has matured into a very effective means of communicating essential programming language design and implementation ideas to our students. The course could be offered as the only advanced course in the area, as one course in a two-semester sequence, or as an alternative to the traditional course.

Categories and Subject Descriptors D.3.m [**Programming Languages**]: Miscellaneous – the programming languages course, history of programming languages, object-oriented programming languages, computer science education.

General Terms: Design, Languages, Theory.

Keywords: languages; education; design; implementation; object-oriented; event-driven; Simula; Smalltalk; Java; C++; C#

1. Introduction

The richness and relevance of the study of programming languages has only increased over the years as worthy hands have explored its vast potential. But the complexity of the programming language design domain presents an obstacle to its study on the undergraduate level. The typical bachelor's degree in Computer Science involves at most one course in programming language design and implementation, and in that course the instructor must make some difficult decisions, with which he or she often is not comfortable. Moving lightly over the surface of this difficult area is at best unsatisfying, so the instructor often makes the choice to teach the course as a series of case studies, preceded by a summary of the essential concepts. Although large areas of study may go uninvestigated, the languages studied are studied at a sufficient depth to form an appreciation for their utility and power.

This common occurrence in practice, this tendency of our colleagues toward a set of depth-first selections from the subject, naturally forces us to ask the question whether perhaps we should build the sidewalks where we find the footpaths. In other words, we should ask whether a course should be designed which introduces the undergraduate to the topic of programming language design in a substantive way without attempting to achieve broad coverage of the area. Certainly, if the course were taught in that fashion, additional breadth and depth could be achieved with another elective course or with graduate study. In the interim, a solid foundation for further study will have been achieved, and those who do not choose to continue in the area will come away from the course having a firm grasp of the essentials.

To come to the point, this author believes there is plenty of motivation for the discussion of the important issues in programming language design bound up in the study of a few historically and currently influential object-oriented languages, which form a very coherent thread and to which practically every junior-level undergraduate has a sufficient introduction for the purpose of beginning and successfully completing that study. A course in Programming Language Design and Implementation, based on the object-oriented paradigm (with an introduction to event-driven programming included), has been offered at Grove City College for ten years, and the concept has proven not only quite workable, but very effective.

The course has as its focus the basic principles of object-oriented languages and their historical evolution, but uses that organizational thread to introduce all the relevant concepts in programming language design.

2. Organization

The course at GCC begins with a discussion of what it means for a language to be object-oriented (a review for our students). In this context we review the concepts of data abstraction, information hiding, inheritance, polymorphism, overloading, and generic programming. We then offer by way of contrast a definition and brief high-level discussion of some of the other paradigms, namely procedural, modular ([2], [8]), functional ([1], [7]), and declara-

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

tive ([6]). We do not elaborate greatly on these other programming styles, since to a certain extent the modular style is subsumed by the object-oriented languages, and since our students will see the C language (procedural) in the operating systems course and the LISP (semi-functional) and Prolog (declarative) languages in the Artificial Intelligence course.

Essential programming language concepts are then taught, in the context of the object-oriented paradigm. These concepts include abstraction, syntax and semantics, machines and virtual machines, the semantic mapping, binding times and translation strategies. The student's view of referencing environments (meaning the various contexts in which meanings of symbols can be determined, as defined by Pratt in [9]) is expanded by a more general discussion of scope and scope rules, visibility, access levels, value and pointer semantics, and parameter passing mechanisms. Security in language design is highlighted with discussions concerning types, type checking, type transfers, existence checking and variable initialization, aliasing, and exception handling.

A comparatively brief discussion occurs early in the course, which concerns models for event-driven programming, starting with a discussion of interrupts and interrupt handlers as a primitive version of callback tables. We discuss message-passing as a unifying theme between OOPLs and event-driven environments, and move quickly to object-oriented models for event-driven programming. Some standard categories of objects which are generally associated with event-driven environments are discussed as examples of program units which both generate and respond to events, including standard GUI components such as text boxes, scroll bars, and list boxes.

The course is targeted toward case studies of significant OOPLs, because it is in those case studies that the students become involved in a hands-on way. To provide a context for their use of actual languages and programming environments we provide two initial case studies, one of them a comparatively "ancient" language no longer in general use and one of them both historically and currently relevant. These languages are, respectively, the Simula 67 and Smalltalk languages. The first is used to provide a historical setting, a "bridge" between the older procedural languages and the true object-oriented languages, whereas the second serves two functions: (a) to show the students a language with a small kernel and very few consistently applied design principles, and (b) to show the relative purity, power and persistence of this very first true object-oriented language.

Simula 67 was originally conceived as a "niche language", specifically designed for discrete event simulation. It was, in fact, also a general-purpose programming language, and it quickly attracted a following outside the simulation community because of its innovative language design. Because of this forward-thinking design, and also for historical reasons and to motivate the material on Smalltalk, we have introduced into the course at GCC a brief discussion of Simula. Simula is used as a stepping-off place, a bridge between the old procedural paradigm and the beginnings of the object-oriented paradigm. In the context of the Simula language one can begin to talk about the important issues of storage management and referencing environments, and point out to the student the seeds of the idea of programmer-defined data types. The language also points backwards, as it was strongly influenced by Algol. As does Algol, Simula requires a three-tiered approach to storage management equivalent to that of modern languages: static, stack-based, and heap-based. Also as does Algol, Simula requires its heap to manage variable-sized storage elements; moreover, Simula expands the purposes for which the heap is used by storing not only dynamic arrays but also the activation records of objects.

Interesting to students is the fact that the transition from procedural to object-oriented languages as we know them would probably never have occurred without Simula, which was certainly not invented with the idea of establishing a new paradigm. Although objects were a natural outgrowth of the need to represent the entities in a simulation, these were not quite objects in the modern sense. Classes were not considered to be a generalization of the abstract data type, which did not at that time exist as a language feature, but as a generalization of the procedure. Moreover, objects were not considered instantiations of a type but were simply procedure invocations whose activation records were not destroyed upon return and which "hung around for questioning" afterwards. In other words, it was only by accident that the class provided a facility for programmer-defined data types. But as a result of these simple ideas, Simula invented methods, inheritance, and polymorphism. Students enjoy hearing about this and other accidents in the history of programming languages, and although the discussion about Simula is not long (students are not expected to develop applications in Simula), it sets the stage for a series of discussions about language design and puts everything into a historical perspective.

Moving from Simula to Smalltalk ([4]), the students see not only a language with a very few elegant design principles, but a language which was the first to be built around a development environment which assumed a mouse and bit-mapped graphics. Tying into the students' avid interest in direct-manipulation interfaces, and using the excellent open-source Squeak environment, a direct descendent of Smalltalk-80, we take the students on a journey from the past into the future, and with relatively little pain show them just how radical a programming language design can be. We show them the amazing flexibility achieved with the idea of late bindings, we show them code as data, and we show them some aspects of the functional paradigm using the notion of block contexts. We also show them some of the most important design tradeoffs: the tradeoff between flexibility and efficiency; and the tradeoff between power through generality and power through structure and built-in features. We show them programmer-defined operators; but, perhaps more importantly, we show them a radically object-oriented view of programming, in which the "message" truly is delivered to the object itself, not to a statically declared variable.

Smalltalk is not only historically important, but it is very important in a course like this because it plays the role that Lisp plays in a more traditional Programming Languages course. Namely, Smalltalk is an example of a language with a very simple implementation and a strong but very flexible type structure, where types are not bound to variables at translation time and where existence checking and type checking are routinely done at run time. It also eschews traditional control structures, relying on message-passing and polymorphism to accomplish control flow.

3. Alternatives

After an introduction to Smalltalk, the course normally divides into one of two alternatives, depending on whether the student's main instruction was in C++ ([10]) or Java ([5]). (At GCC, where the students have already had ample exposure to C++, we do our next case study on Java.)

C++ and Java are superficially similar, and it is our duty to show the students just how radically different from each other they are. For students who have already seen the emphasis on safety, regularity, and maintainability of Java, a language based on the assumption that much human error can be avoided through

careful language design, the instructor can show them the result obtained in the design of C++ by the opposite assumption – that programmers sometimes need to have complete freedom in order to construct an application that is both powerful and efficient.

Conversely, having struggled to learn the byzantine details necessary to master C++, the student can be shown a completely different set of design choices in Java, some of which lead to more secure and maintainable systems and some of which may seem baffling to the student who has been trained to consider efficiency as well as readability. The instructor can show the students the radical effect on a language's design that is engendered by the choice between pointer semantics and value semantics.

In placing one of these languages for contrast alongside the other, we can show our students how syntactically similar source code elements can have radically different meanings, purposes, and implementations. Examples of this are (a) templates in C++ as opposed to Java generics; and (b) inner classes as statically nested types versus inner classes whose objects are dynamic entities requiring ownership by an object of the outer class type.

Certainly the notion of a library as an extension of a language is richly illustrated with either Java or C++, and the student should see the different directions those two languages have taken with their standard container classes and iterators. Libraries for event-handling are an eye-catcher for students. In the course at GCC we follow up on the introduction to event-driven programming in Smalltalk by giving the student time to explore the event/listener model in Java's Swing library. Alternatives for C++ could be the MFC document/view model or Windows Forms.

4. New Ideas

Finally, a lot can be learned from the upstart C# ([3]), which lately has been getting a lot of attention and a lot of the market share previously claimed by Visual Basic. An odd mixture of ideas from C++ (e.g. operator overloading, enumerations, namespaces, "unsafe code") and Java (e.g. interfaces, single inheritance, range checking, automatic initialization of variables), C# also adds a number of interesting new ideas which challenge our notions of what constitutes good language design. Delegates have strange semantics but pragmatically seem to be "just the thing" to support event-driven programming. "Properties" streamline the idea of access functions but again offend our sensibilities by blurring the lines between code and data. Expression trees and lambda functions have a strongly functional flavor. This language may or may not be here to stay, but like PL/I it will have a long run and will engender much useful discussion. Much can be gained by giving the students the chance, after a thorough acquaintance with the important issues, to look closely at this relatively new design and form their own opinions.

5. Experience with the Course

The course at Grove City College has evolved from fairly humble roots, but has matured into a lynchpin of our curriculum and has been enthusiastically accepted by our students. Our curriculum begins with a heavy dose of C++, and exposure to functional and declarative languages occurs in the context of the Artificial Intelligence class, while the Operating Systems class gives students experience with the procedural C language. Without this course, however, our students would not have the opportunity to put all these paradigms into historical perspective, and they would not see the very important ideas of safety and maintainability embodied in the Java language.

The secondary theme of event-driven programming in the course as offered here is more than a sidebar. It is a key ingredient for making the course seem relevant to our students, and it is a language/library feature design puzzle. Are event handlers to be simply function pointers, or are they to be objects which implement an interface, or should they be a specially designed language feature? Also, is the message loop a separate thread? Is it a part of the run-time system? Is it attached to an object? How is the correspondence established between the event source, the event object, and the handler? GUI design is a useful hook to get students interested, but once they begin to engage in that design we can ask them some important questions that might not have been meaningful to them before they were able to have hands-on experience.

6. Examples

Some specific teaching points and code examples are presented in this section, to illustrate the way some important concepts are taught. Note that the level of detail indicated in these examples would be difficult to achieve in a more general-purpose course in programming languages.

6.1 Interrupts and Events

Interrupts are introduced in the course as an example of a primitive type of "language support" for event-driven programming. The interrupt vector table and its "hard-wired" operation are described, and it is noted that this mechanism is the inspiration for call-back tables. The fact that synchronously initiated interrupts (software interrupts) were a natural addition to the instruction sets of interruptible machines illustrates the usefulness of adding a level of indirection between the occurrence of an interrupt (or event) and the actions which are initiated in response to it. The operation of the callback table is then discussed as a software-simulated way to achieve this same set of advantages in higher-level languages. It is then pointed out that a true object-oriented model for event-driven programming requires more than a function pointer – it requires an (object, method) pair. Throughout the course the students see multiple ways of realizing such an object-oriented model, ranging from (1) C++'s relatively primitive "pointer to member", to (2) Java's library convention which, for each "listener" object interested in the event, will invoke at run time a method having a specific signature, and (3) C#'s "first-class" delegate types whose objects are a curious combination of the "function pointer" and "listener list" ideas.

6.2 Messages

Like the term "heap", the term "message" has multiple meanings, and the students need to be warned of this fact. We present two definitions, namely that (1) a message is an encapsulated event notification, and (2) a message is a method call. We point out that in whatever manner the message is delivered, for example by direct invocation of a method or by placing an event notification on a queue, the same information is transmitted, namely a message type and a collection of accompanying data concerning that message. The difference in efficiency is evident, since direct invocation does not require a separate polling thread. Immediate but indirect invocation using a callback table or one of its more sophisticated generalizations (discussed above) is shown to be an intermediate position, offering both efficiency and flexibility.

6.3 Pointer Semantics and Value Semantics

One of the most valuable things we can give our students in any course in programming language design is a thorough understand-

ing of both of the common implementation strategies for the assignment and initialization operations – those dictated by pointer semantics and by value semantics, respectively. The student should learn by direct experience how it feels to program under each convention. The contrast between advantages and disadvantages is striking; the extra overhead incurred in a language with value semantics, by routinely making separate copies of large objects, is a disadvantage balanced by a marked reduction in the number of aliases produced. The disadvantages of value semantics are painfully apparent when a large number of large objects is inserted in a container, and the disadvantages of pointer semantics are seen when complicated cloning protocols must be followed to achieve the effect of deep copying.

For students who are used to value semantics, it is disconcerting when they make changes to an object recently removed from a container and find that those changes are mirrored in another object they had conceived as having a separate identity. For students trained in pointer semantics, the concept of "call by reference" seems unnecessary until they fully understand value semantics. We do our students a disservice if we do not make them struggle with these ideas.

A very good example of the confusion created with these two opposing semantic viewpoints is seen in the common mistake students make when designing copy constructors. In Java, a copy constructor is just another way of constructing a cloning facility, and the Java programmer pays his or her dues by learning about and enforcing the deep copy discipline. When that programmer must design a copy constructor for a C++ class, the deep copy issue is not at the forefront and usually is a non-issue. The mistake the Java programmer learning C++ makes in constructing a C++ copy constructor is that he more often than not attempts to pass the constructor's parameter by value. Since the call by value implicitly invokes the copy constructor, an infinite recursive call sequence is initiated and the programmer is baffled by the stack overflow that results. The resulting psychological impact is traumatic, but it is useful for helping the programmer to see the need for "call by constant reference" as an alternative to call by value.

6.4 Value Types, Inheritance, Templates, and Generics

Smalltalk has a radically object-oriented point of view which requires that all objects, even simple numeric quantities, be "first-class" objects, and that their classes must inherit from a common base class. One consequence of this requirement is that all generically constructed container classes are heterogeneous, and the corresponding container objects have the ability to store any object. Subsequent designs have tried to achieve this same kind of flexibility and power in their library container classes.

C++, with its emphasis on preprocessor and compiler actions, solved the problem using templates instead of inheritance. A consequence is that C++ container objects are type-specific instead of heterogeneous, but that type-specificity serves one of C++'s major goals of efficient run-time operation. Also, because C++ puts off some of its compile-time checks until template expansion time, the programmer can effectually make a class template as type-specific as she desires, by sending type-specific messages to template arguments which are not in fact explicitly bound by the template definition to a type. This design is flexible but does not obey the principle that interfaces to declared entities should be explicitly indicated in their declarations.

Java moved more in the direction of Smalltalk, requiring all classes to inherit from the same *Object* base class. Java terms all such types reference types, and like Smalltalk it provides automatic heap storage management for all objects of such types. Initially all Java library containers were designed to store only references to objects of type *Object*.

But Java's primitive types are value types and not first-class objects, and so cannot be stored in library containers. The solution to this problem is the wrapper classes and the compile-time trick of autoboxing. These decisions made it possible to store anything in a Java library container, if one was willing to pay the price of explicitly casting it from the *Object* base class type back to its proper type after fetching it from that container.

Then an interesting thing happened. Java's design expanded to make room for a "generic" class definition facility which superficially resembled C++'s class templates. An alternative generic container library allowed the programmer to use containers of type *LinkedList<Double>*, for example, rather than the heterogenous *LinkedList* class. Interestingly, although the one pattern, *LinkedList<>*, is in fact a generic facility for creating linked list containers, the containers which programmers are able to create from that pattern are less generic than the original *LinkedList*, and in fact are "type-specific", the opposite of "generic". The resulting advantage actually had nothing to do with genericity, but in fact allowed the programmer to dispense with casting and allowed the compiler to do compile-time type checking and thus improve run-time efficiency.

All these considerations are interesting to students, and it is a valuable exercise for them to see the complexities resulting from the different design decisions. It is also important that they see the stark difference between C++ templates and Java generics, as evidenced by the fact that a template definition cannot be fully compiled into conventional object code, whereas a Java generic class can be compiled into the same type of object code as an ordinary Java class.

6.5 Unifying Value Types and Reference Types

Through all of the above considerations, Smalltalk's simplicity stands apart in its ability to unify all types into one hierarchy. But languages which compile to a low-level code, whether for a real or virtual machine, must make allowances for the fact that the data types on that machine must have corresponding types in the high-level language if the efficiency and power of those data types is to be made available to the programmer. Hence all such languages include these low-level types as "value types". C++ in essence makes all types value types, but some of those types are pointer types, and the programmer can explicitly manipulate pointers as values rather than the values to which they point.

C# attempts to blend the C++ and the Java views by maintaining the distinction between value types and reference types as it exists in Java but extending value semantics to some structured types. Whereas in C++ the *struct* and the *class* are logically equivalent in terms of their abilities to construct new types, C# uses those two keywords in two radically different ways. A type constructed with the keyword *struct* uses value semantics, whereas a type constructed using *class* uses pointer semantics. Thus the notation

```
new MyType(<argument list>)
```

which in previous languages has always denoted a pointer or reference, in C# denotes a value if *MyType* is defined as a struct.

Not content with borrowing the idea of value semantics for large objects from C++, the C# language has also borrowed the "type unification" idea, folding all value types and reference types into a single inheritance hierarchy. Although value types may not be used as base classes, all of them have as base class the type *ValueType*, which in turn inherits from the ultimate base class *Object*. Interestingly, *ValueType* is a reference type. What is actually happening here is that C# is using its own style of "auto-

boxing", but rather than using specific "wrapper classes" for that purpose it wraps a *ValueType* object around any value type object being passed to a parameter of type *Object*.

Ironically, it appears that it was the desire to accommodate multiple programming styles (all within the object-oriented umbrella, of course) that led the C# designers to defeat their goal of unified types. The "unsafe" pointer types are borrowed directly from C++ and cannot be allowed to participate in the type hierarchy that enfolds the other types, for the simple reason that if a pointer type were allowed to be a first-class object, then it could escape its unsafe block by being used as the return value of a method call.

For Java programmers, used to using specific wrapper classes, there is a pitfall awaiting them when they begin to deal with C# value types, and that is its wholesale use of type name aliases. Seeing the correspondence between *int* and the library type *Int32*, the Java programmer jumps to the natural conclusion that the latter is a wrapper class for the former. He is surprised when he realizes that in fact there is no difference between the two, other than the fact that the former name is a keyword.

6.6 Inner Classes

Instructive to students is the difference between the inner classes of Java and of C++, and the reasons for that difference. In C++ an inner class is just a type definition within a type definition. For that reason, a C++ inner class may not reference an instance variable (non-static data member) of the outer class, since at any given time there may be no instances or many instances of the outer class and no clear way to associate the reference with a specific instance. This fact is illustrated by the instantiation code below, where *Inner* is the name of the inner class and *Outer* the name of the outer class.

```
Outer.Inner oInner = new Outer.Inner();
```

There is no indication in these notations of any particular instance of the outer class, which might be used to resolve references to data members in the outer class. Contrast this to the Java instantiation code below, in which *oOuter* is an object of the outer class type.

```
Outer.Inner oInner = oOuter.new Outer.Inner();
```

Here the notation explicitly provides an object to be used for resolving references to outer class instance variables, namely *oOuter*. Of course, when the instantiation occurs inside an instance method of the outer class, the outer class object need not be named if it is the object which received the message. The outer class object in that case is understood to be *this*.

Thus inner class objects can by means of non-local references deliver portions of the state of the outer class object to their "owners". For example, if *head* is the first element of a linked list in the outer class, then an object which was created as an instantiation of an inner class has access to the attribute *head*, and the client programmer could indirectly be allowed access to that attribute using an instantiation of the inner class. Referring to the above example, *oInner* might have the ability to return the first value in the list maintained inside *oOuter* via a method call, for example *oInner.first()*.

Even more instructive to the students is the use to which Java's inner classes are put. Inner class objects are naturals for iterators and for event handler objects. Here Java's interface facility (for which there is no equivalent in C++) comes in handy, because the inner class type need not be public, and indeed need not even be a named type, if it implements the interface needed by the client. A "listener list" maintained by an object having knowledge of the occurrence of an event can add the inner class object to its list of interested objects, without needing to know the actual type of the listener object, as long as the listener object implements the required interface.

6.7 Delegates and Event Handling

Java's listener lists are a library convention which makes use of language facilities not explicitly designed for event handling. By way of contrast, C#'s delegate types are "tailor-made" for event handling, but also seem to violate the software engineering principle of functional independence – delegate types are designed to do two things at once. Any declared delegate variable has the potential to denote either a "first-class" function object or a collection of such function objects. Consider the following C# class and the delegate type definition following it.

```
public class Cheers {
    public static void Generic() {
        Console.WriteLine("Yay!"); }
    public static void LongGeneric() {
        Console.WriteLine("Yaaaaaaaaaay!"); }
    private string teamName;
    public Cheers(string teamName) {
        this.teamName = teamName; }
    public void Specific() {
        Console.WriteLine("Yay, {0}!", teamName);
    }
    public void LongSpecific() {
         Console.WriteLine(
                "Yaaaaaaaaaay, {0}!", teamName);
    }
}

public delegate void VoidFunction();
```

The class *Cheers* encapsulates a private string – *teamName*, the name of a sports team for which we are to cheer. It also provides a constructor for instantiating a *Cheers* object, two static member functions *Generic()* and *LongGeneric()* which provide non-team-specific cheers, and two ordinary member functions *Specific()* and *LongSpecific()* which provide team-specific cheers. The delegate type *VoidFunction* is a pattern for creating first-class objects from methods which have no parameters and which return no result, a property shared by all four methods of the class. The code

```
VoidFunction cheerleader = new VoidFunction(
        Cheers.Generic
);
```

instantiates such a first-class object to correspond to our generic cheer, and the "call"

```
cheerleader();
```

produces the output

```
Yay!
```

on the console. But far from stopping there, C# was going after a more streamlined version of Java's "listener lists", so it allows the curious looking code

```
        cheerleader += Cheers.LongGeneric;
        Cheers irish = new Cheers("Irish");
        Cheers bama = new Cheers("Bama");
        cheerleader += irish.Specific;
```

```
cheerleader += bama.Specific;
cheerleader += irish.LongSpecific;
cheerleader += bama.LongSpecific;
```

after which the single call *cheerleader()* produces the output

```
Yay!
Yaaaaaaaaay!
Yay, Irish!
Yay, Bama!
Yaaaaaaaaay, Irish!
Yaaaaaaaaay, Bama!
```

Let us look more closely at this code. First of all, note that there is actually no type provided in the language for a method name *sans* arguments, such as *irish.Specific*. So how are we to parse the following line of code?

```
cheerleader += irish.Specific;
```

The first and most obvious response is that the function name is "auto-boxed" in a sense, so that the following line of code is equivalent.

```
cheerleader +=
        new VoidFunction(irish.Specific);
```

But this explanation fails too, since the entity "passed to" the "constructor" for *VoidFunction* will be vulnerable to the same question. What is its type?

The answer is simply that *irish.Specific* has no type, and the code above is not actually a "constructor call", nor is there a piece of code anywhere which we can identify as a "constructor" for *VoidFunction*. The parser identifies *irish.Specific* as belonging to the syntactic category "method group", and the CLR code generating the corresponding delegate object is produced by the compiler after a context-sensitive parse. The component elements of the token string

```
new VoidFunction(irish.Specific)
```

do not have any meaning when taken separately. This design clearly violates the software engineering principle of modularity.

The design of C# continues to evolve, at a pace which might be considered quite rapid if we take into consideration the deliberate way in which other programming languages have evolved. The question we must ask our students is whether continued radical departures from conventional design principles are justified by short-term pragmatic goals, or will there be a long-term price to be paid by making the design serve *ad hoc* objectives?

6.8 Polymorphism – Forced or Optional?

As originally included in Simula, polymorphism was an optional facility. If a method was declared *virtual*, then it behaved in a polymorphic fashion. If it was not, then compile-time actions bound member function calls to a particular statically-determined function definition. Smalltalk departed radically from that design, requiring that a message should be sent to an object, and that it should not matter by what name that object is identified. Smalltalk provides only one exception to that rule, as follows. If the object is identified by the name *super*, then the message is handled not by the object itself, but by the object considered as an instance of its superclass. This provides a curious example of an unintended infinite loop. Consider the following method definition, where f is the name of an instance method already defined in the superclass.

```
f
        super f f
```

Here the method call *super f* will be handled by the definition for *f* in the superclass. Assuming that definition does not provide a specific return value, Smalltalk will automatically use *self* as the value returned. But even though the code returning that value is superclass code, the notation *self* denotes the original object, not the object considered as an instance of the superclass. This means that the second call of *f* will *not* be handled by the superclass, but will be handled using Smalltalk's strict interpretation of polymorphism – the object receiving the original message will handle the call. Since the original call to *f* was handled by the code above, so will this call, meaning that we have begun an infinitely recursive loop.

The Simula view and the Smalltalk view of polymorphism both still have their proponents. (To recap, those views are respectively that (a) polymorphism should be explicitly requested and that (b) polymorphism should be the default behavior.) C++ and C# take the Simula view and use the *virtual* keyword in a way similar to the way it is used in Simula, and Java takes the Smalltalk view. C# goes one step further, however, adding the keywords *override* and *new* to fine-tune polymorphic behavior down the inheritance hierarchy. An override of a virtual function is expected to be given the attribute *override* as a notification to the reader that it redefines, and participates in the polymorphic behavior of, a function in a base class (although the omission of such notification should only produce a warning). More interesting is the attribute *new*, which "seals off" the method name and prohibits polymorphism for the method to which it is attached, from that point down in the inheritance chain. One result of this design is the following curious anomaly. Consider the three C# class definitions below.

```
public class MyIndirectBase {
    virtual public int f(int x) {
        return x;
    }
}
public class MyBase: MyIndirectBase {
    override public int f(int x) {
        return x * x;
    }
}
public class MyDerived: MyBase {
    new public int f(int x) {
        return base.f(x)*x;
    }
}
```

The reader should consider the following two lines of code, and ask herself what they will produce on the console.

```
MyIndirectBase b = new MyDerived();
Console.WriteLine(b.f(3));
```

The somewhat disturbing answer is that the method called with the expression *b.f(3)* will not be the method *f* associated with the declared type of *b*, nor will it be the method *f* associated with the actual type of the object to which the name *b* gives access. It will be the method associated with the intermediate type, *MyBase*.

The value returned is 9. The reason is twofold: (1) the declared type of *b* has a polymorphic form of method *f*, since the *virtual* keyword has been used in the definition of that method, so that any derived classes which have an override for the method have a stronger claim to handle the message; but (2) because the version of *f* in the actual class to which the object belongs is given the *new* attribute, it is forbidden for that version of *f* to participate in the implementation of polymorphism. Since by right of inheritance *b* "is" an object of type *MyBase*, that is the only logical place to resolve the call.

6.9 Functional Notions in the OOPLs Domain

Smalltalk shares with the functional languages a dependence on function composition as the major facility for controlling flow through a program. It is "corrupted", from the standpoint of the functional paradigm, by its use within each method definition of the default control mechanism referred to in structured programming terms as "sequence", which requires a series of machine states and hence completely disqualifies Smalltalk as a functional language in the purest sense. However, one important element often seen in functional languages is also seen in Smalltalk, namely functions as first-class objects. The data type *BlockContext* provides such a facility, and is one of the few library types for which special notations are provided for its literals. For example, we can represent the function $f(x,y) = x^2 y - 3x+5y$ as the literal

```
[:x :y | x*x*y - (3*x) + (5*y)]
```

Sending the message **value**: 3 **value**: 7 to the object denoted by this literal is equivalent to evaluating the function call *f*(3,7).

C++ adds no native facilities for functions as first-class objects, but there is a subterfuge which that language uses to give it some of the same benefits. The "function application" operator can be overloaded multiple times for any class, so that if *f* is an object of a type which provides such an overload, say with two integer operands, then the expression *f*(3,4) is translated by sending the *operator*()(3,4) message to object *f*. (Interestingly enough, to this author's knowledge no one has designed an object-oriented event-handling facility based on using such "function objects" as event handlers.)

We have mentioned above the delegate types of C#, which are (among other things) a facility for functions as first-class objects. An element of convenience is added to that facility by the use of lambda expressions, which are able to describe "on the fly" the actions represented by such a function object. A C# lambda expression similar to our Smalltalk example above might appear as follows.

```
(x, y) => (x*x*y - 3*x + 5*y)
```

The difference is that whereas the corresponding Smalltalk expression is a literal of type *BlockContext*, the C# expression has no type, and represents only a syntactic category. To give it any usefulness the programmer must instantiate an object of a delegate type. For example, consider the following two declarations of delegate types *IntFunction* and *DoubleFunction* and delegate objects fashioned from them.

```
int IntFunction(int x, int y);
double DoubleFunction(double x, double y);
IntFunction f1 = new IntFunction(
     (x, y) => (x*x*y - 3*x + 5*y)
);
DoubleFunction f2 = new DoubleFunction(
     (x, y) => (x*x*y - 3*x + 5*y)
);
```

Given these declarations, the expressions *f1*(1,1) and *f2*(2,0.5) evaluate to 3 and to –1.5, respectively.

7. Future Directions

Long ignored because of their relative simplicity, the scripting languages have shown enough staying power to deserve some coverage in a course like this. Consisting of a minimum emphasis on structure and a maximum emphasis on functionality, delivered by a powerful and dynamic library, these languages have more in common with Smalltalk than with any of the other languages mentioned above. Coverage of the salient features of the Python language will be introduced into the next offering of the course.

8. Conclusion

In summary, the ideas in the field of object-oriented programming language design and implementation have reached critical mass, are important ideas relevant to undergraduates, and deserve to constitute a course in their own right. It is this author's view that such a course is valuable and should seriously be considered as an option in designing undergraduate Computer Science curricula.

Acknowledgments

Support for much of the effort in developing course materials was given to the author by Grove City College via a grant for a sabbatical semester, for which the author is very grateful. The author is also grateful for the tireless dedication of his colleagues Bill Birmingham, Dave Adams, and Christiaan Gribble, who took up the slack during his sabbatical. Special thanks also go to Ed Gehringer, whose encouragement and helpful suggestions have made this a much better paper.

References

[1] Backus, J. 1978. Can Programming be Liberated from the Von Newmann Style? Communications of the ACM 21, 8 (August 1978), 613-641.

[2] Barnes, J. G. P. An overview of Ada. 1980. Software Practice and Experience 10, 11 (November 1980), 851-887.

[3] ECMA International. 2006. *C# Language Specification. 4th Edition*. DOI = http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

[4] Goldberg, Adele and Robson, David. 1989. *Smalltalk-80: The Language*. Addison-Wesley.

[5] Gosling, J, Bill Joy, et al. 2000. *The Java Language Specification. 2nd Edition.* Sun Microsystems. DOI = http: //java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

[6] Kowalski, Robert. 1979. Algorithm = Logic+Control. Communications of the ACM 22, 7 (July 1979), 424-436.

[7] McCarthy, J. 1960. Recursive Functions of Symbolic Expressions. Communications of the ACM 3, 4 (April 1960), 184-195.

[8] Paul, Robert J. 1984. An introduction to Modula 2, BYTE Magazine 9, 8 (August 1984), 195-202.

[9] Pratt, Terrence. 1975. *Programming Languages: Design and Implementation.* 1st Edition. Prentice-Hall.

[10] Stroustrup, B. 2000. *The C++ Programming Language: Special Edition.* Pearson.