The problem of teaching object-oriented programming

Part I: Languages[1]

Michael Kölling
School of Computer Science and Software Engineering
Monash University

mik@csse.monash.edu.au

1 INTRODUCTION

Object-oriented programming has, in recent years, become the most influential programming paradigm. It is widely used in education and industry, and almost every university teaches object-orientation somewhere in its curriculum. The software community more or less agrees that teaching object-oriented programming is a good thing. It elegantly supports the concepts that we have been trying to teach for many years, such as well structured programming, modularisation and program design. It also supports techniques for approaching problems that have only more recently made their way into the curriculum: programming in teams, maintenance of large systems and software reuse. In short, object-oriented programming seems to be a good tool for teaching those programming methodologies that we consider important.

Teaching object-oriented programming, however, remains difficult.

Many reports of the experience of those attempting to teach object-oriented programming include a long list of problems with many different aspects of the systems used. Why is it difficult? Or, to be more precise, why does the teaching of object-oriented programming seem to be more difficult than the teaching of structured programming?

Before we attempt an answer to this question, we should look at one other aspect of this problem: *when* should object-oriented programming be taught?

For a long time, object-oriented programming was considered an advanced subject that was taught late in the curriculum. This is slowly changing: more and more universities have started to teach object-orientation in their first programming course. The main reason for doing this is the often cited problem of the *paradigm shift*. Learning to program in an object-oriented style seems to be very difficult after being used to a procedural style. Anecdotal evidence (e.g. in [1]) indicates that it takes the average programmer 6 to 18 months to switch her mind-set from a procedural to an object-oriented view of the world. Experience, on the other hand, also shows that students do not seem to have any difficulty understanding object-oriented principles if they encounter them first. It is the switch that is difficult, not object-orientation.

For teaching programming, the lesson is clear: if we want to teach object-orientation, we should do it first. The path to object-orientation through procedural

---

[1] This paper has been published as: Kölling, M., "The Problem of Teaching Object-Oriented Programming, Part 1: Languages," *Journal of Object-Oriented Programming*, 11(8): 8-15, 1999.

programming is unnecessarily complicated. Students first learn one style of programming, then they have to "un-learn" the previously learned, before we show them how to do it "right".

Unfortunately, many textbooks use procedural programming as a pathway to object concepts. One of the main influences in this is the language C++. Its popularity in industry has led to it being often used for teaching as well. C++ is a hybrid language that supports procedural ("C style") programming and object-oriented programming. It was developed as an extension to C. This has led to a misunderstanding: many people view object-orientation as just another language construct that can be taught after control structures, pointers and recursion. This is a serious mistake.

Object-orientation is an underlying paradigm that shapes our whole way of thinking about how to map a problem onto an algorithmic model. It determines in fundamental ways the structure of even simple programs. It cannot be "added on" to other language constructs; rather it replaces the fundamental structure of procedural programming.

Because of this, we firmly believe that object-oriented concepts should be taught from the very beginning. If it is seen as necessary that students can also program in a procedural style, then a procedural language can be introduced later. The paradigm shift backwards (from object-orientation to procedural) is much easier. Programmers can just think about it as writing the complete solution within one class (a solution an object-oriented programmer might not be very happy about, but easy enough to understand).

Having come to this conclusion, we can now ask the question about the difficulties in teaching object-orientation more precisely: Why is it difficult to teach object-oriented programming to first-year students?

In our view, it is not object-orientation in principle that causes the problems, but the tools available to teach it. Programming languages used are too complex and programming environments – if they exist at all – are too confusing. Some systems used for teaching were really developed for professional software engineers, making it difficult for first-year students to cope; others were not "developed" at all but grew out of historic coincidences.

In short: in our view the reason for all the trouble is that the wrong languages and environments are being used.

That, of course, immediately leads to the next question: What tools should we use? What should a good language and environment for teaching object-oriented programming look like?

This is the question that we try to answer here.

We will, in a series of four columns of which this is the first, discuss the problems and solutions around this issue. This month's column provides a detailed discussion of requirements for a teaching language for object-oriented programming. What characteristics should a language have to be useful for teaching? After identifying the requirements we will go on to evaluate some languages against those requirements. The languages covered are C++, Eiffel, Smalltalk and Java.

One of the results will be that the programming environment available for teaching has a major influence on the quality of the overall system. Because of its great

importance, we will devote a separate column in a coming issue of JOOP to the discussion of programming environments for teaching. The last two columns give a detailed description of the Blue system – our attempt to design a system that is better suited to the task.

2 REQUIREMENTS FOR A TEACHING LANGUAGE

The requirements for object-oriented languages in general have often been discussed in the literature. Many arguments have been brought forward for and against specific language constructs; every possible feature has been argued for and against. The question as to whether or not multiple inheritance is necessary, for example, has sparked ongoing disputes, often carried out with religious fervour.

General discussions like those are often pointless and unproductive. Languages are not good or bad *per se*; they are good or bad *for a specific purpose*. A language that is bad in one context can be excellent in another (or vice versa). It is more a question of the right tool for the right job: it is pointless to argue whether a hammer is better than a screwdriver if you are not arguing in the specific context of what you want to do.

The following discussion of requirements for an object-oriented language is in the context of the use as a teaching language for beginners. The following criteria should be met by a language *to be useful as a teaching language* for first-year students.

Requirements overview

The requirements can be summarised by the following key words:

1 *Clean concepts*
2 *Pure object-orientation*
3 *Safety*
4 *High level*
5 *Simple object/execution model*
6 *Readable syntax*
7 *No redundancy*
8 *Small*
9 *Easy transition to other languages*
10 *Support for correctness assurance*
11 *Suitable environment*

Note that some issues, such as efficiency, which are often considered extremely important for production programming languages, are of little significance for a teaching language; it is only required that the language be able to be supported in a teaching environment with reasonable response time. Similarly, it is not important that the language be flexible enough to develop real-world applications (e.g. by the inclusion of operations such as arbitrary bit manipulation) – it will never be used for this purpose.

We now discuss the points listed above in more detail.

Language requirements in detail

1 *Clean concepts*

The concepts that we want to teach should be represented in the language in a clean, consistent and easy-to-understand way. In particular, they should be represented in the same way we want to talk about them when teaching. We should avoid presenting a model of a construct in lectures in one way and using a language that implements variations of the model (or does not implement the model at all). We should also not let the language dictate what we have to talk about in first year lectures. The implementation language should reflect the level of abstraction that we want to use for our conceptual models.

2 *Pure object-orientation*

The expression "pure object-orientation" is chosen to mean the opposite of "hybrid" languages – languages that support the object-oriented but also non-object-oriented paradigms. (C++, for example, is a hybrid language.)

Many people have argued for hybrid languages, in particular for C++, on the basis that the hybrid character of the language might ease the entry to object-oriented programming for students with prior programming experience [1, 2]. If a student knows C already, a hybrid language like C++ might provide an easy entry path. Object-orientation could be gradually added to an existing body of knowledge, making understanding of the problems easier by flattening the learning curve.

While this argument sounds plausible at first, it is fundamentally flawed.

It is true that many students enter universities with prior programming experience, and it would be foolish to ignore this fact, or not to try to exploit its benefits. On the other hand, it is often the case that students come into the institution with a self-taught "cowboy" style of programming that in no way resembles the good programming practices which we try to convey. In teaching, we have to make sure that those students change their habits over time. Changing one's habits, however, is harder than learning new ones.

The danger with hybrid languages is that students with prior programming experience in a procedural language are not encouraged to change their style. On the contrary – they can write programs for a long time, believing them to be object-oriented, while missing all of the important concepts. Experience shows that beginning students often believe their work to be finished as soon as the compiler accepts their program. Equally, as soon as a C++ compiler accepts their code, they believe that they have written an object-oriented program. Of course, this might not be the case at all. The fact that a C++ compiler accepts non-object-oriented C programs as valid input, turns out to be a hindrance rather than a help in getting students to write object-oriented programs.

3 *Safety*

The principle of safety is that errors that can easily be detected by the compiler or the runtime system should be detected. Furthermore, they should be detected

early, and clear messages should be given about their cause. If error-prone constructs can be avoided altogether, they should be avoided.

While this statement sounds obvious, it is far from being the current state of affairs in some of today's most popular languages. An example of this principle is a good type system: the language should be strongly and, as far as possible, statically typed. In dynamically typed languages the point of detection of the error might be a long way away (in time and location) from the actual source of the problem. This makes understanding and eliminating program errors an unnecessarily difficult task.

Other examples of safety are checking of array bounds or checking for the use of uninitialised variables. Constructs known to be problematic should be avoided, where possible. Explicit pointers, for example, are known by every programming teacher as a source of major difficulties for students. Their use in a programming language can be avoided altogether – several programming languages do not require pointers to be explicitly dereferenced. Pointer arithmetic has no place in a first year programming course.

4 *High level*

The programmer should not need to be concerned about machine internals. Tasks that can easily be carried out by the compiler or the runtime system should not be the responsibility of the programmer.

The most prominent example of a violation of this requirement is the explicit management of dynamic storage. Putting the responsibility for storage management into the hands of a programmer (especially if it is a beginning student) is unnecessary and leads to frustrating experiences. The problems with errors that are hard to find due to dangling pointers, double deallocation, or other forms of memory corruption are well known and form one of the hardest-to-debug groups of errors in programming in general. All of this can be avoided by using a system that provides automatic garbage collection.

5 *Simple object/execution model*

The model of execution should be simple and easy to understand. This includes the object model and/or the memory model. Many languages distinguish between, for example, two different kinds of memory layout for objects: objects on the stack and objects on the heap. Some languages require some objects to be explicitly allocated while others are allocated automatically. Some of these differences might be necessary internally. None of them, however, should be visible in the language itself.

6 *Readable syntax*

The syntax used should be easily readable and consistent. There are many reasons for this. First of all, a readable program is more likely to be correct. While readable syntax, of course, does not guarantee correctness, there are regular cases where errors are discovered in programs that are present only because a programmer did not understand another part of the code. If we assume that readability increases understandability, then the case for readability is clear.

But does readability really increase understandability? The case might be different for beginners and experienced programmers, but we believe the answer is yes in both cases. Before we go further into this, let us state more clearly our view of the meaning of readability.

The most important aspect is that we favour keywords over symbols. Words are much more intuitive than symbols in many cases – carefully chosen keywords can reveal most of the semantics of many instructions. This makes the language both easier to learn and its programs easier to read.

A readable language has many advantages for teachers and students. Teachers are often reluctant to learn a new language only to teach a new course. Programs of a well-designed language can be immediately readable for anyone who is experienced in another language with similar concepts. Pascal, for example, is easily understandable for an experienced C programmer (but not the other way around). In this way, a readable language takes a burden off the teachers who have to learn the language themselves. For students it means that they can, after only a very short time, take educated guesses at the meanings of constructs. Many people, for example, when they get a textbook about a programming language, really read only the programming examples in that book. This should not be considered "wrong" – learning by example is a powerful way of learning that we all apply at some time. The easier the examples are to understand, the better it works. We should try to exploit and encourage this way of learning. Keywords also can be looked up in the index of a good textbook – an important point for a teaching language.

Another aspect of readability is consistency. The same syntax should be used for same semantics, different syntax for different semantics.

For both beginners and experienced programmers readability has clear advantages. It makes the learning of the language easier and helps to reduce the number of errors. Maybe a lesson should be taken here from the art of writing language as it has developed over many centuries. In the writings in human languages it has long been recognised that the fundamental aim for useful or aesthetically pleasing writing is in the effect on the reader, not the speed of the writer. This is true for all forms of writing, from technical manuals to poetry. The quality of the writing lies in the way that it is able to convey meaning to the reader – easy understandability for technical documents, emotional association for poems. In none of these cases would the writer dream of arguing that some of the words should be omitted because the reader can still work out the meaning, and the writer has to type less. On the contrary: writing is done *for the reader*.

In an age in which we have recognised that programmers actually spend much more time reading programs than writing them, and in which we agree that reading and understanding code may be more difficult than producing it, writing for the reader should become an accepted principle in computing as well. Gone is the time when programming can be considered as "writing for the machine". With improving compiler technology, producing a program understandable to a machine has become the easier part. Producing a program that is understandable to humans is the real challenge.

7 *No redundancy*

"No redundancy" means that for everything we want to do in the language, there should be one, and only one, way to do it.

Redundancy is often connected with flexibility and efficiency: having different constructs to achieve the same task is often useful to optimise code. These alternative constructs might differ in low level details such as performance, memory layout, etc. To write efficient code it may be essential to influence these low-level details.

For beginners flexibility leads more often to confusion than to efficiency. Having three different mechanisms for the same thing, which differ in sometimes subtle detail, often poses problems for students when they cannot make a complete judgement as to the effects of their choice.

8 *Small*

The language should be as small as possible while including all important features that we want to discuss in the first-year programming course. Use of larger languages usually requires the instructor to teach a subset of the language. We do not consider that a good solution. One obvious problem is that students use parts of the language that are not in the officially sanctioned subset (because they read examples from a textbook). The teacher then needs to explain why it should not be used, or accepts its use and is forced to deal with it.

With a small language students can reach the point at which they know all of the language constructs. This is psychologically very important. If students write their programs knowing that there are still several chapters to come in their textbook introducing new constructs, they always retain a degree of insecurity. With many problems they try to solve, they cannot be sure whether they are really doing it the right way, whether there is not another construct just around the corner that would solve the problem in a much easier, more elegant way. We consider it important to get to a point where we can say to a student: "This is all. You have now seen all of the available constructs; from now on programming is not about learning new constructs any more, but about how to put them together." We believe that this allows a student to better focus on algorithms and design aspects, rather then spending too much time on specifics of many single constructs. Pascal and LISP are languages where this is possible, which was often a reason for their use in first year teaching.

9 *Easy transition*

An essential aspect of any teaching language is that what is learnt by using it must be relevant to what is needed later.

The result of this argument is often that a language that is popular in industry (like C or C++ or, increasingly, Java) is used for first year teaching. This might not be the best choice.

Most institutions are not free of forces from the "real world" – we have to teach our students to program in a language that is relevant to industry. But it is a wrong conclusion that this means that we have to start teaching in C++. Most people would agree that, by the time students leave a university, they should be

competent programmers in an industry-relevant language. This only means that students should end up with strong, say C++, skills, but not that they must start with it. On the contrary: We, as teachers, have to ensure that students learn *programming* as opposed to *a programming language*.

This means that we have to teach them the principles of programming. We might do this in any language we think appropriate. (See also [3] for a good discussion of the difference between teaching programming principles and teaching a programming language.) On the other hand, students should not leave our universities without being able to use a current real-world language. The language used for first-year teaching should be relevant to current industrial languages. Concepts learned with the first teaching language must be easily transferable to the next language following it.

10 *Correctness assurance*

Many software engineering principles are now taught in first year programming courses. Among those are techniques such as design by contract, which relies on the use of pre and post conditions, and correctness support through, for instance, class invariants.

The language used for teaching should support these as first class language constructs. While pre and post conditions, for instance, can be emulated in many languages (with assertions or comments) their existence as first class language constructs much better reflects the importance that we now give them. Students take these techniques much more seriously if they are supported by the language rather than given in style guidelines.

11 *Environment*

The language must have a good graphical integrated program development environment to support it. The environment must hide details of the underlying operating system and allow students to focus on the programming task at hand.

One of the reasons, why teaching object-oriented programming has been difficult in beginners' courses is the complexity of the environment. We would like an environment that, firstly, lets us concentrate on the programming language rather than the operating system, and secondly, supports the object-oriented paradigm.

The environment is, in fact, one of the most important points in this list of requirements. A suitable language can be unusable for teaching because of a lack of a good environment. If we were to rate them in importance, the environment requirement alone would probably be as important as all the other points together.

Since the environment is so important, and since it is not at all clear what it means to have a "good" environment that "supports the object-oriented paradigm", we will discuss the requirements for the environment separately later.

Some of the requirements listed above contradict each other. The inclusion of software engineering constructs such as pre and post conditions and class invariants, for instance, and the goal that the language should be small are in conflict. For a

good teaching language it will be essential to strike a good balance between these conflicting goals. There will, of course, always be discussions about how small is too small, whether really *all* redundancy can or should be removed, and so on. For the design of a teaching language it is important to use these requirements as guidelines to design decisions and weigh the effects of different requirements against each other. They are not a straightforward recipe to create the perfect language, but they are a basis on which to judge the suitability of a particular language for first-year teaching.

3 EVALUATION OF LANGUAGES

Many papers have been published comparing different aspects of a variety of languages (e.g. [4-7]). Most of those, however, concentrate on technical aspects of the languages themselves, rather than assessing their suitability for teaching. Here we evaluate *C++*, *Java*, *Eiffel* and *Smalltalk* in light of their suitability for first year teaching.

C++

C++ fails to meet almost all requirements on our list. It is a hybrid language that does not support the concepts in a clean way, has a highly redundant set of constructs, an unsafe type system and a highly complex execution model. Some of the most important points in more detail:

Clean concepts – Whatever can be said in favour of C++, it is not that it represents any abstract concepts in a clean way. Two of the most influential design decisions in the development of C++ were to retain full backwards compatibility with C and to regard efficiency in time and space as a main goal. (Stroustrup stated that "time and space overheads above those for C are considered unacceptable for C++" [8].) Both those characteristics have the effect that the representation of many important language constructs is heavily influenced by low-level considerations that distinguish the concrete representation from the abstract concept. Often, subtle implementation details need to be understood to correctly use fundamental constructs.

Safety – One of the most serious criticisms of C++ is its lack of type safety. Almost all its higher level constructs can easily be corrupted. The explicit dynamic storage allocation in C++, in combination with the lack of garbage collection, also greatly increases the risk of errors. Often C and C++ programs that have been tested and used for some time have "memory leaks" and other bugs that are caused by improper storage handling.

High level – C++ includes numerous constructs for low-level manipulations. Bit operations and pointer arithmetic, for instance, are frequently used in typical C++ programs. The use of explicit storage management not only allows but forces the programmer to think at an unnecessarily low level.

Simple object/execution model – The object model supported by C++ is clearly the most complex of the languages included in this survey. This is caused in part by C++'s support for pointer and non-pointer ("automatic") variables, as well as the semantics of some important constructs.

Readable syntax – The lack of readability is another negative point for C++. Its syntax is overly terse and has as one of its most obvious features its favouritism of symbols over keywords. As a result of this, C++ often overloads the same keyword for different purposes or uses unintuitive symbolic constructs (consider, for example, the syntax for the definition of an abstract function: `virtual void f () = 0;`). By employing a syntax similar to variable assignment, the meaning of the construct is hidden and the declaration becomes unreadable to a non-C++-programmer. There are numerous similar examples in the language.

No redundancy – The decision to keep C++ upwards compatible with C has led to the evolution of a large number of redundancies and surprising interactions of concepts, and greatly increases the complexity of the language.

The list of problems mentioned here could easily be extended. Many other problems have been pointed out and discussed in much detail in the literature (e.g. [9-11]). Detailed discussion of all of them would exceed the space we want to devote to this language.

Overall, C++ is one of the worst candidates for our needs.

Smalltalk

Smalltalk [12] is sometimes referred to as “the most object-oriented” language. It carries object-orientation further than other languages. Smalltalk states that “everything is an object” and very consistently sticks to this rule. Even control structures in the language are considered objects. It supports a very clear and consistent object model. This consistency makes it attractive as a teaching language. Some issues, however, arise. The most important are discussed below (again grouped by the requirements identified earlier).

Clean concepts - While most fundamental concepts of object-orientation are supported in a clean and consistent manner, some constructs, which are now accepted as being important to object-orientation, are not well represented in the language. An important example is the poor support for implementation and interface distinction. In Smalltalk, all methods are automatically public. This, combined with the lack of special constructor functions and an encapsulation model that prevents class methods from accessing instance data, makes it necessary to write initialisation methods as public operations, even though they should never be publicly called. They are typically tagged with a comment declaring them private, and it is hoped that clients follow this recommendation. This is not an ideal situation for teaching about encapsulation. Another example is the lack of support for pre and post conditions.

Safety – The main drawback of Smalltalk is its lack of static typing. Sakkinen [11], in a discussion of programming languages, distinguishes languages for “exploratory programming” and languages for “software engineering”. The former aim at great dynamism and run-time flexibility, the latter have static typing and other features that aid verifyability and/or efficiency. In most modern university courses we aim to prepare students for a software engineering view of the world, thus favouring languages of the second kind. Smalltalk clearly is a representative of the first group. It was developed with a requirement in mind that the user be able to rapidly change the application structure [13]. The lack of static typing typically causes a range of unnecessary problems for students.

Readable syntax – Smalltalk's syntax is another characteristic, which we view as a drawback. It is alternately referred to as "simple" and "obscure". Both are true in some sense. It is simple in that it is composed of only a few syntactic concepts and structures. Thus, it is simple in the same sense in which LISP syntax is simple. In another sense, Smalltalk's syntax is obscure. This, unfortunately, is the sense that is important to our discussion: readability. The unification of concepts in Smalltalk, the view, for instance, that even control structures are objects, results in a syntax that is less readable and less intuitive than the ALGOL-style syntax adopted by many other languages (e.g. Pascal, Modula, Eiffel).

Although syntax is to a great extent a question of experience (and thus the argument that "they just have to get used to it" is often brought forward), we should not ignore two certain aspects of experience. Firstly, many students entering university courses today have prior experience with a programming language, and we would be foolish not to try to build on this experience. Secondly, we want the experience gained by students in our course to be as helpful and relevant to future work as possible. In both respects, adopting an ALGOL-like syntax seems to be beneficial.

Small – The size of Smalltalk can be viewed in two ways. The language itself (in terms of the number of constructs) is small; the Smalltalk system, however, is large. Smalltalk usually offers a huge class library. Since everything in Smalltalk is an object, extensive use must be made of the library from the very beginning. Virtually all publications about teaching Smalltalk, although generally positive about the system as a whole, report problems coping with the size of the class library. Tempte [14], in a paper describing his experiences with teaching Smalltalk, writes: "It is easy to underestimate the difficulty of learning to use Smalltalk effectively. Smalltalk is not an isolated language but a programming system which uses very simple language syntax ... in conjunction with a very large class library within an interactive environment. Software development requires facility (sic) not only with the object-oriented paradigm but also with the library and the environment." He goes on to state that more time than expected had to be devoted to teaching about the structure of the library. Similarly, Skublics [15] reports that a survey of students after a course using Smalltalk indicated that they found the existing class library overwhelming; LaLonde and Pugh [16] report students to be "more apprehensive" because of the "sheer amount of code provided by the Smalltalk library".

We stated earlier that the teaching language should be small, so that the students do not feel overly intimidated or lost in the programming system. This requirement must also include those class libraries that students necessarily encounter. We do not at all argue against the use of class libraries in general. On the contrary: we think that experience with the use of library classes is essential to foster a culture of good programming practice and software reuse. But similar criteria to those that we apply to the evaluation of the language proper must also be applied to the libraries used in the first year course. In this respect the Smalltalk environment is overwhelming. Because of the nature of the Smalltalk system, a considerable amount of time must be spent getting acquainted with the structure of the class library.

Eiffel

Of all the languages surveyed, Eiffel probably comes closest to fulfilling our language requirements. It is statically typed, supports object-oriented concepts in a clean way, avoids redundancy and has a clear, easily readable syntax. This

combination makes it a better candidate than any other language. Some remaining criticisms are:

Clean concepts – Although Eiffel represents most concepts in a clean way, there are some exceptions. Eiffel has, for example, two different storage modes for objects, the normal (reference) mode and "expanded" (value) mode. This is unfortunate from a pedagogical point of view. Its only purpose is to increase runtime efficiency, with no justification at a conceptual level. The effect of supporting these two modes is that programmers are forced to think about implementation details during class design (since some designs, such as self referencing structures, can only be implemented with one of the alternatives).

Simple object/execution model – Eiffel's object model is not as simple as would be desirable. It is unnecessarily complicated through the use of the two different storage modes.

Its attempt to support a very large number of constructs sometimes leads to overly complex structures. Nørmark [17] points out an example: the common idiom in which a redefined routine in a subclass calls the original routine in the superclass. To do this in Eiffel, it is necessary to inherit the superclass twice, redefine the routine in one inherited version, rename it in the other and select the first one. The fact that a technique such as repeated inheritance is necessary for such a common pattern is extremely unfortunate. This use of inheritance does not fit well with the main purpose of inheritance as specialisation which we try to convey in first-year courses.

This example shows that the complexity of the language cannot be ignored by teaching a subset. As Meyer himself put it [18, p500]: "... the idea of orthogonality, popularised by Algol 68, does not live up to its promises: apparently unrelated aspects will produce strange combinations, which the language specification must cover explicitly". Advanced features of the language (repeated inheritance) affect the appearance or behaviour of other features (superclass calls).

Small – While Eiffel mostly avoids multiple constructs for the same concept, it supports a large number of concepts. This makes Eiffel large, even though it has little redundancy. It contains numerous constructs which cannot be included in a first year course. The way to deal with this problem would be to teach a subset of Eiffel – a solution which we have found to be undesirable (see discussion of requirement 8 above).

Environment – The most fundamental problems with Eiffel are not with the language itself, but with the libraries and the programming environment. The libraries are far too extensive and too specialised for an introductory course. They are described by teachers and students as overwhelming and difficult to navigate [17, 19]. The collection class library, for example, is extremely hard to understand and difficult to use. The linked list class, which will be used in many first-year projects, has more than 50 interface routines!

The most severe problem is the environment. Currently, only a handful of Eiffel implementations exist. Of the few that are available, even fewer include a graphical development environment. The most sophisticated and most widely used Eiffel environment is "EiffelBench" from ISE. But even this environment, the most promising for our purposes of those available, is unsuitable for use by first year students. In a report about using it in a fifth-semester course, Nørmark [17]

described it as low in quality and unreliable. He describes numerous severe problems with the environment: compilation is too slow (even though three different compilation modes are provided), the different compilation modes (which are only meant to save time) are inconsistent, and it is difficult to use. In a student survey, 37% of the students said that they chose not to use it (and used a Unix editor and shell instead). Of those who used the graphical environment, 64.7% found the experience “negative” or “very negative”, 35.3% judged it as “neutral” and no one (0%) described it as “positive” or “very positive”. Nørmark concludes that it “is not good enough for teaching purposes”.

Considering the importance we attach to the environment of an object-oriented language in a first-year course, this is a serious problem. The environment is needed to hide some of the complexity inherent in the implementation of object-oriented technology (such as the need to deal with multiple files) from the student. The Eiffel environment, on the other hand, adds complexity. Eiffel (at least as it is currently available) must therefore be considered unsuitable.

Java

Java is very quickly becoming one of the most popular object-oriented languages on the market. Never before has a language spread so quickly and been taken up by so many programmers in such a short amount of time. (Never before has a language been pushed with such an immense organised marketing effort, either!)

Unfortunately, the reasons for its popularity have as much to do with the marketing push and with coincidental side aspects as with the quality of the language itself. Three aspects, among all others, can be identified that lead to the great success of Java: the fact that it was the first language to produce applets to run inside a web browser, the wide support of its virtual machine that promises platform independence and the marketing as a “better C++” – by being compared almost exclusively to C++ it appeared very small and simple.

In a more sober analysis Java still looks like a well designed language, but not like the final solution to all our problems, as the hype makes one believe. Here are some points in detail:

Clean concepts – Many of the important concepts of object-orientation are represented in a fairly clean way. On the positive side is the handling of multiple inheritance: Java supports separate type inheritance through a concept called interfaces. It allows only single class inheritance, but multiple type inheritance. This is a very good compromise which allows the programmer to use much of the functionality provided by multiple inheritance while avoiding most of its problems.

On the negative side are numerous small problems. Simple types, for instance, are not regarded as classes. This sometimes causes problems, so a second type – a class type – is defined for each simple type. As a result, there are types *boolean* and *Boolean*, one of which is a class and the other is not. The duality exists for all the predefined scalar types.

The language forces some of the more advanced concepts into the foreground very early. This is most prevalent in the very first function: the *main* function that every application needs to provide. The required signature for this function is

```
public static void main (String[] args)
```

There are several concepts used in this line that a teacher might not want to introduce at this point: static functions, the *void* return type, array parameters. Yet this is necessarily the first code that students see. (It gets worse if the class uses input: then an exception declaration has to be added to the signature.)

So while each concept on its own is well represented (except for syntactical aspects, see below), the fact that they cannot be initially avoided introduces problems.

High level – While Java operates mostly on a high level, there is one notable omission: genericity. The lack of genericity in Java is very unfortunate and forces the use of type casts at certain points.

Readable syntax – The syntax is a weak point in Java. It uses C/C++ syntax which we have already criticised above. A more detailed description of the problems associated with the C style syntax can be found in [9-11].

Correctness assurance – Pre and post conditions are not supported in Java. This, together with the lack of support for genericity, is the most unfortunate design decision in the language.

Overall, the criticisms listed here are by far not as severe as those of C++. When only compared to C++, Java appears like the saviour we've been waiting for. When evaluating against our requirements, however, Java is far from being an ideal solution.

Conclusion

A survey of the languages commonly used for teaching object-orientation reveals problems with all of them. The problems are of a different nature and of different scale for each of these languages. Two methods of analysis are available to us: experience reports from teachers who have used the language in practice and a comparison of the language against our requirements listed in section 2.

Experience reports are mixed in their conclusions, but a general trend can be found. Object-orientation itself was seen unequivocally as a powerful and valuable teaching tool, while almost all authors reported problems with specific languages and systems they used. Most of the studies reported difficulty in switching to the object-oriented paradigm from the procedural approach.

In analysing the languages, the language itself and the programming environment used both have to be considered. A well designed language is only half of what we need and is rendered useless by the absence of a suitable environment. With some languages, the environment was the source of the most serious problems. Thus, programming environments will be discussed in more detail in a future column.

Acknowledgments

The work described in these columns is a cooperation of Prof. John Rosenberg, Monash University, and the author.

References

[1] Stroustrup, B., *The Design and Evolution of C++*, Addison-Wesley, Reading, MA, 1994.

[2] Biddle, R. and E. Tempero, "Teaching C++ - Experience at Victoria University of Wellington", University of Wellington, Technical Report CS-TR-94/18, 1994.

[3] Knudsen, J. L. and O. L. Madsen, "Teaching Object-Oriented Programming is more than teaching Object-Oriented Programming Languages," in *Proceedings of ECOOP '88*, pp. 21-40, Springer-Verlag, Oslo, Norway, 1988.

[4] Blaschek, G., G. Pomberger and A. Tritzinger, "A Comparison of Object-Oriented Programming Languages," *Structured Programming*, 10(4), 1989.

[5] Henderson, R. and B. Zorn, "A Comparison of Object-Oriented Programming in Four Modern Languages", University of Colorado at Boulder, Technical Report CU-CS-641-93, February 1993.

[6] Kristensen, B. B. and K. Østerbye, "A Conceptual Perspective on the Comparison of Object-Oriented Programming Languages," *SIGPLAN Notices*, 31(2): 42-54, 1996.

[7] Schmidt, H. W. and S. M. Omohundro, "CLOS, Eiffel and Sather - A Comparison", ICSI, Berkeley, Technical Report TR-91-047, 1991.

[8] Ellis, M. A. and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.

[9] Joyner, I., "C++?? - A Critique of C++ and Programming Language Trends of the 1990s, 3rd edition", Unisys - ACUS, Report , 1996.

[10] Pohl, I. and D. Edelson, "A to Z: C Language Shortcomings," *Computer Languages*, 13(2): 51-64, 1988.

[11] Sakkinen, M., "The darker side of C++ revisited," *Structured Programming*, 13(4): 155-177, 1992.

[12] Goldberg, A. and D. Robson, *Smalltalk-80: The Language*, Addison-Wesley, 1989.

[13] Goldberg, A., "What Should We Teach?," in *Proceedings of OOPSLA '95*, pp. 30-37, ACM, 1995.

[14] Temte, M. C., "Let's Begin Introducing the Object-Oriented Paradigm," in *Proceedings of SIGCSE '91*, pp. 73-77, ACM, 1991.

[15] Skublics, S. and P. White, "Teaching Smalltalk as a First Programming Language," in *Proceedings of SIGCSE '91*, pp. 231-234, ACM, 1991.

[16] LaLonde, W. and J. Pugh, "Smalltalk as the first programming language: The Carleton experience," *Journal of Object-Oriented Programming*, 3(4): 60-65, 1990.

[17] Nørmark, K., *An Evaluation of Eiffel as the first Object-oriented Programming Language in the CS Curriculum*, Aalborg University, Denmark, May 1995, ftp://ftp.iesd.auc.dk/pub/projects/normark/eiffel-eval.ps.

[18] Meyer, B., *Eiffel: The Language*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[19] Mazaitis, D., "The Object-Oriented Paradigm in the Undergraduate Curriculum: A Survey of Implementations and Issues," *SIGCSE Bulletin*, 25(3): 58-64, 1993.