VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Changing Programming Languages for Teaching

Robert Biddle and Ewan Tempero

Technical Report CS-TR-99/11
November 1999

Abstract

When we teach how to program, the language to use is not the only decision we must make, but it is an important decision that involves many higher level concerns, and lower level details. As such, the decision to change programming languages is one that requires careful consideration of all the issues.

In this paper we present the issues involved in changing the language we use to teach programming. We first concentrate on the issues that involve the language itself, discussing what we look for in a teaching language. We then discuss the many practical issues involved in making the decision whether to change. The new language we consider is Jade, and the older languages we use for comparison are those most commonly used for teaching programming in recent years: Pascal, C++, and Java. We base our discussion on our extensive experience in teaching these languages, and working with both industry programmers and tertiary students.

Publishing Information

This paper was presented at the JADE Tertiary Conference, 16-17 December 1999, Christchurch, New Zealand.

1 Introduction

Whenever a new programming language is introduced, those teaching introductory programming must decide whether or not to change to the new language. When we teach how to program, the language we use can have a big impact on how successful the course is. Using a language that has many complex features can result in students struggling with those features, rather than learning the fundamentals of programming. Using a language that provides only simple features can result in students who are not able to adapt their knowledge to languages they may encounter in the workplace. Nor is it just the language itself that impacts the students' learning experience. The infrastructure in which the students learn the language is also important. So there are many concerns that must be addressed when deciding to change programming languages. In this paper, we discuss these concerns with respect to Jade [5].

The question of which language is best to use in an introductory programming course has been a matter for great debate for many years. It is not our intention to continue that debate here, but rather we will discuss guidelines that can be applied to determining whether a language is good enough. Specifically, we will be considering the guidelines for evaluating an imperative object-oriented programming language for use at the beginning of a computer science curriculum.

This paper is organised as follows. We begin in section 2 by discussing the desirable characteristics of a first language. The discussion will focus on characteristics relating to principles, pedagogy, and practicalities. We will then, in section 3, examine Jade from the point of view of principles and pedagogy. Section 4 will examine practicality issues in more detail and discuss them with respect to Jade, while section 5 will look at the cost of changing a language. Finally, we present our conclusions.

2 Desirable First Language Characteristics

There are several issues that must be addressed when choosing a programming language to teach at the beginning of a computer science degree. In this section we outline the desirable characteristics of the language itself. There are several broad categories of characteristics involved, and there are tradeoffs between them. There has never been a perfect first language, and there has never even been complete agreement on which is the best choice. However, some choice must be made, and the characteristics and tradeoffs must be considered carefully in context.

2.1 Principles

The primary characteristic we look for in a first language is support for computer science *principles.* That is, whether the language has enough of the fundamental concepts of programming we feel should be taught as a foundation for our curriculum. One difficulty here is that different programming paradigms emphasise different principles. Logic programming languages (such as Prolog) , functional programming (such as Haskell), and others, all have advantages. This diversity of paradigms, and well as the diversity of languages, makes choice even more difficult.

We have chosen to select an imperative object-oriented (OO) programming language. We believe this paradigm supports principles as well as any other, and has the advantage that it is most likely to be useful in later programming work, either in our curriculum or in other work. We ourselves use the imperative OO paradigm languages in later courses, and it is widely used in industry. We do also use other language paradigms later in our curriculum, as appropriate to particular course topics.

Having made the decision to look for an imperative OO language, our concern for support of principles then focuses at the detail level. We hope for support for a number of concepts, but realise we must be flexible in what we will find in a single language. We will want a language that makes the OO model clear, using classes, objects as instances of classes, and with methods for behaviour. We will look for languages that provide strong encapsulation, emphasising data hiding and the independence of interface and implementation. We also believe that good static type-checking is an advantage in a language. Even in an introductory language, we will expect both inheritance and polymorphism, and will want both to be clear and understandable. We also favour support for genericity, especially generic container classes, if possible.

Beyond any simple checklist, we must more generally be concerned at the overall effect. In particular, OO languages have been successful especially because they have supported widely acclaimed OO techniques for design reuse, including OO design patterns [4] and OO frameworks [6]. Whatever language we choose should support these techniques.

2.2 Pedagogy

Another major issue we must address is whether the language is pedagogically sound. As we are considering the first language students will learn, the pedagogical characteristics are critical. The language must be easy to teach, easy to learn, and must make *programming* easy to teach and easy to learn. Many of the implications are similar to those common in user interfaces designed for high usability and especially high learnability. While still supporting principles of good programming, the language should be simple, and minimise both key concepts and critical technical details. Even if the full language has many more ideas involved, the language should allow progressive disclosure. That is, learning a small set of concepts should allow simple programming, and it should then be possible to later learn more concepts allowing more sophisticated programming. It will also help if the language uses concepts or constructs that may already be familiar to the students, or fosters helpful metaphors that assist quick learning. Where new technical details must be introduced, it will help in the long term if they facilitate later learning of other languages.

2.3 Practicalities

Finally, we must also take into serious consideration various practicalities. This includes checking the support infrastructure available for the language, such as development environments and related software, and whether these are suitable for use by beginners. We must also consider portability, and support for the language on a variety of machine and operating system platforms. Also important is textbook support, with preferably a variety of textbooks to allow selection of one that works well within our overall curriculum. Moreover, we must remember that cost really is an issue in considering this support, both cost to the university, and cost to students who will buy the textbook, and, increasingly, who will want copies of the software to use on their home computers.

Practicality consideration must also include assessment of the end result we aim for. We must think carefully about how useful learning the language will be to our students. Having decided to select an existing language in the imperative object-oriented paradigm, we feel assured that we cannot go far wrong. However, it will also assist our students if our first language is one that can be directly applied in others courses or work situations. In recent years the computing industry has developed a high profile, and many of our students, even at first year, are aware of computing industry trends. In particular, many students take a strong interest in what skills and experience industry looks for when recruiting. All this means that using a programming language in demand by industry not only benefits our students, but may also attract students in the first place.

In the next section, we discuss the Jade language explicitly. We discuss how well it supports the computer science principles we need to teach, and how well it supports our pedagogical needs for easy learnability. In section 4 we discuss the other practicalities of using Jade as a first programming language.

3 Teaching Jade as a First Language

In this section, we discuss how the Jade system measures up to the desirable characteristics of principles and pedagogy. This represents an initial attempt to evaluate Jade and is based on our experience in teaching other languages. We ourselves have no experience teaching Jade, and so where we have indicated a possible problem, that should be interpreted as something to watch out for, rather than an actual problem. Due to space constraints, we have concentrated on the more obvious issues.

3.1 Principles

Jade's underlying philosophy includes that it have no complicated syntax to remember, and that it is consistent, with no special rules [3]. Its syntax is Pascal-like, and is in fact based on Oberon, a language developed to succeed Pascal as a teaching language. This heritage means it has many of the attributes of a good teaching language.

Overall, Jade has all the characteristics we expect in an object-oriented language, as described in section 2.1. There are two issues however. Firstly, there is no direct support for generic container classes in the language (although, as we discuss below, the development environment does support this concept), but in this Jade is no worse than Java. Secondly, Jade lacks support for explicit separation between interface and implementation. While it does have the notion of abstract class, our experience is that this in combination with single inheritance can be restricting. With explicit separation of implementation and interface, such as with the Java `interface` feature, many of the benefits of multiple inheritance are possible [2]. These are quite minor issues, and don't by themselves detract from using Jade as a first language.

3.2 Pedagogy

3.2.1 Detail Issues

In this section we address the detail level of the Jade language. This level of the language is quite significant for beginning programmers, because beginners really need to understand the fundamentals of data and control to understand how computing works.

Jade provides a number of built-in data types, the "primitive" data types. These include standard basic types such as Integer, Boolean, Real and a standard set of operators for those types. It also includes String as a primitive type, with concatenation, comparison, and subscript operators. Strings that are used as properties (see below) can have a specified maximum length, and in fact the environment assumes that as a default. It also provides a number of other primitive data types, such as Date, not found in other languages. Operators for these types will generally have the obvious meaning (for example, `+` can be used to add days to a Date).

Jade also provides a standard set of control-flow statements: if (with optional else, and elseif clauses), while, and repeat. Although there is no for statement, as is found in most languages, it does provide a foreach statement that iterates over the members of a collection. This is more general that the standard for statement. There is a break instruction, for terminating loops, and continue, for ending the current execution of a loop body.

Jade follows the Pascal tradition for assignment (`:=`) and equality (`=`), and so avoids the common confusion that beginners have between the equivalents in C++ and Java (`=` for assignment and `==` for equality).

Like Java, Jade makes a distinction between primitive types and object types (classes). Unlike Java, the distinction is generally quite clear — data of primitive type have value semantics whereas data of object type have reference semantics. Java has the concept of built-in object type, which behaves somewhat like a primitive in some situations, but has reference semantics. We feel that Jade has a distinct advantage in this case. However, it is unfortunate that variables of both primitive and object types can be assigned the special value `null`, and that the array versions of the primitive types (see below) have reference semantics. These two points will confuse the distinction to students who have not grasped the concepts involved.

A consequence of object types having reference semantics is that, because object types will be encountered by the beginner, the beginner must really understand the idea of reference semantics. In many ways, understanding object references is very similar to understanding pointers. While it could be argued that not having explicit pointers simplifies Jade (this argument is also made by Java advocates), our experience is that students can struggle if there is no syntactic cue that reference behaviour is present [1].

Understanding reference assignment also means understanding that when an object variable is assigned to, the original referenced object value is no longer accessible via that variable. This in turn means understanding that when that variable had been the *only* reference to the object value, the object value is now inaccessible. And this, at quite an early stage, may lead to the need to explain the complex topic of memory management.

There is one unique aspect of Jade that reinforces the distinction between primitive and object types. Classes in Jade consist of methods and properties, where properties are further divided into *attributes* and *references*. Attributes are properties that are of primitive data type, whereas references are properties that are of object type. We do not anticipate this distinction causing any great problems to students, however anyone teaching a different language in future courses must be aware of this feature.

Beginners need to use objects even at an early stage to access the world beyond their program, and there are pedagogical reasons for beginners to get experience in using objects before they go on to create their own classes. Simple use of pre-existing classes and objects is reasonably straightforward, and uses syntax similar to most other object-oriented languages.

Jade provides 13 different primitive data types, and for each primitive type it also provides a *array* version for that data type. This is unfortunate as in most languages, the "array" concept provides good motivation for generic container classes, whereas in Jade this concept is hidden. There is, however, support for generic container classes in the development environment in combination with the *Collection* class hierarchy. This means that generic classes can be discussed, although the implementation of such concepts will be somewhat mysterious.

Despite the comments above, the presentation of primitive data types, particularly strings and arrays, is simpler than languages such as Java. This is fortunate, as strings and arrays are very common in programming, and cannot easily be avoided or postponed for long.

3.2.2 Program Structure Issues

The top level structuring mechanism in Jade is the Schema. A Schema is a grouping of classes, and schema themselves are organised in a hierarchy. A potential source of confusion for beginners is that one application may involve more than one schemas, and there may be more than one application based on a single schema. However, early in the course we would expect to follow a "one application – one schema" convention, and so don't anticipate any problems.

The concept of encapsulation and data hiding is fairly clear in Jade. Properties and methods can be declared protected, indicating they can only be accessed by the class in which they are declared, or that class' subclasses. However the concept of the "interface" to a class is somewhat undermined. In Jade, the class declaration involves properties and methods, as with C++. However, in C++ the class can specify only the function signatures for the methods, and the function implementations can be specified elsewhere. By grouping all `public` signatures, a C++ class makes the interface to the class apparent, and the distinction between the interface and the implementation is made clear to beginners. A Jade class, however, requires method implementations to be given in detail in the class declaration, which makes the interface unclear. The Jade environment does help in this regard, in that it shows all the names of the methods available, but it would help if the signatures were also shown.

An interesting feature of Jade is that object can be easily made persistent, that is, the object can last beyond the lifetime of the process that creates it. This is possible because Jade has a built-in object-oriented database. Since so many applications involve persistent data, it is important that programmers understand the issues involved, and so being able to expose beginners to this concept is of great benefit.

There is an issue relating to persistence that will impact teaching Jade. Objects can be both persistent and transient, with persistence being the default. Jade requires that all changes to persistent objects must be inside a *transaction*, which is used to ensure the integrity of the database. Since any interesting application the students are likely to encounter early on will involve changes to objects, it is difficult to avoid explaining a complex concept such as transactions early on. Students who forget to use the transaction mechanism will get a "update outside transaction" error. Unfortunately this is a run-time problem, and so it will not be immediately obvious which segment of code caused it. (The Jade debugger will provide this information, but students may encounter this error before it is appropriate to introduce the debugger.) In this case, care must be taken in presenting the transaction concept to beginners.

Programmers have a number of options available to them when declaring methods in Jade. As well as being able to declare them abstract (as part of the support Jade provides for abstract classes), programmers can declare methods as "updating". This means that the method will be making a change to the state of the object it operates on. We feel this is a complex but important concept, and one that beginners often have difficulty with. The fact that the language explicitly supports this concept will help in teaching it.

There are also a number of options for parameters to methods. These are: input, output, i/o (both input and output), and constant (no change can be made to the *value* of the parameter, even if it is a reference to an object). The default is constant, which is the ideal default for beginners. This means that this complexity does not have to be discussed early on, but can be introduced later in the course.

3.2.3 Development Environment

Jade is both a language and an integrated development environment (IDE). IDEs can be a mixed blessing for beginners. On the one hand, IDEs specifically intended for beginners can be set up to "hold the student's hand" in many ways, and so help the beginner come to grips with just the concepts they need to. On the other hand, environments not set up for beginners can behave in (what appear to be) mysterious ways, leading to a great deal of frustration. Not surprisingly, Jade is intended for professional programmers, and has some pitfalls for the unwary. For example, if the properties of a class are changed (something beginners are likely to want to do frequently), then they will encounter the message: "Changes need a reorg. Continue?". Such messages will strike fear into the heart of the student who has forgotten (or never understood) what this means.

Jade comes with a graphical user interface (GUI) builder that can be used to create the forms needed for a user interface. This means that students will be able to quite quickly produce applications that are quite professional looking, which provides them with good encouragement. The GUI-builder is reasonably easy to use, but the number of options would be overwhelming for beginners. However, it would be easy to create a general purpose form that students could use early on, and so the more sophisticated aspects of the GUI-builder could be delayed until they have more experience. Such a general purpose form would be quite restricted in what it can do, but we prefer this approach to tempting the students to spend too much time creating fancy GUIs [1].

4 Jade and Support for Practicalities

In the previous section, we looked at the Jade programming language itself, and considered how it would work as a first programming language. In this section, we look at other practical issues associated with choosing a programming language that will affect the success of the courses, but are not directly related to the language itself. These issues include some practical issues involving the development environment, availability of supporting materials such as textbooks, and use of the language on different platforms and in different application domains.

4.1 Software

The program development environment is a key concern. In order to learn program development, students should have a suitable text-editor, class browser, debugger, and so on. The Jade product does include all these. They are designed to support professional programming, but they do not seem too complex for learners. A related issue is library support for programming. Much programming involves using libraries, and learners need to do this both to access useful library functionality, and also as a way of gaining experience in using code they did not write themselves. Jade does include useful library functionality, and it should fulfill this requirement.

While there is a development environment and there is library support, there is not a diversity of options. Jade is a product of only one supplier, and it does not look as if this will change. Many languages begin as ventures from one company or one group, and some are very successful on this basis. However, we generally prefer to see languages and language support spread more widely. For one thing, this leads to a greater diversity of options for us as educators, and we can select which language implementation or development environment suits us best. It can also mean that some suppliers will focus on certain application domains or certain platforms that may interest us. Our interest or preference may also change over time, as may our curriculum, and diversity again may yield useful options. Of course, every language starts in one place, and it may be that Jade offerings become more diverse. In the meantime, we should remember that a single supplier does help avoid the problems of diversity, in particular the nuisance of incompatibility between versions of the language.

4.2 Textbooks

Beyond the computer itself, the most important support is documentation, and for beginner programmers, that means a textbook that uses the language to teach programming. There is only one book that addresses this need, written by Bevan Clarke [3]. Clarke's book is well-written and does address beginner programmers. However, it is hard in a single book to address all the kinds of beginners that are learning programming. Clarke's book uses a running information systems example, and this example system drives the presentation style and order in the book. In particular, Clarke gives detailed step-by-step guidance on how to create the Jade code for the running example.

Clarke's book seems perhaps well suited to an audience interested primarily in business information systems, but there are other audiences. In particular, an audience interested primarily in computer science might prefer more emphasis on the technology and principles, and wish for more diverse examples. Of course business information management is one of the main applications of software technology, but it is not the only one. Moreover while the principles of programming are there to achieve results, in computer science it is our task to identify and study those principles so that we can learn how to apply them more widely and more wisely. The detail guidance Clarke gives is more typical of the guidance we give in individual programming assignments, where we choose a different example every week. We tend to rely on textbooks more for an alternative detailed presentation of the principles we introduce in lectures, and for a source of reference material useful to the students.

4.3 Applicability and Portability

As we discussed earlier, a primary decision we made was to look for an imperative object-oriented language. This paradigm is widely applied in industry, and we also use it in several of our later courses. Jade is an imperative OO language, and so fulfills our basic requirement. As such, we do feel that it would allow students to learn principles that would apply in other work where the same paradigm was used. For example, students who have learned Jade would have no major shocks in later learning other imperative OO languages, such as Java, C++, or Smalltalk.

However, while changing paradigms can be an ordeal, even changing languages can be non-trivial. Students knowing Jade would still have work to do before becoming confident in Java, C++, or Smalltalk. Learning other languages is certainly a useful experience for programmers, but still not something to be done lightly, especially when other matters have priority.

In the context of a computer science curriculum, later, more specialised courses can involve some serious programming, and having to master a new language at the same time can interfere with learning and productivity. For example, if a course in data communications required use of C++, then students would probably make faster progress if they knew C++, than if they knew Jade and had to learn C++. This kind of reasoning may possibly be countered by other advantages of Jade, but the issue requires careful consideration. Jade does have networking capabilities, but in our courses the practical work involves implementing low-level network behaviour, and Jade may not be the appropriate level of technology.

In a typical computer science curriculum, there are possibly several courses that could be taught in Jade, perhaps including database management courses, and courses in software engineering. A great strength of Jade that it facilitates development of integrated systems involving databases and client-server interaction. In recent years, such systems have been common in our software engineering team project course, using a variety of development tools, particularly focusing on the world-wide-web. Integrated development environments for this kind of system probably deserve more attention in our curriculum.

In the wider world beyond our curriculum, diversity reigns. Some application domains are more suited to Jade than others, so if a student is interested in programming embedded systems, for example, they are more likely to need C++ than Jade. On the other hand, development of integrated information systems is an important domain, and Jade addresses that domain well. Moreover, the growing interest in e-commerce motivates keen interest in this area. In terms of student awareness, Jade does have a high profile, but as a language is not mentioned as widely as others, including C++ and Java.

In addition to applicability, we must also consider portability, meaning which machine platforms support Jade software. According to the suppliers, Jade is available for Intel PCs running Microsoft Windows 95 and NT, and for IBM RS/6000 machines running the AIX version of Unix. That both Windows and Unix is supported is good. Our main platforms use Unix, but we do not use IBM servers or AIX; we use Compaq Alpha and Sun Sparc servers, and also Unix on Intel PCs. We also use Apple Macintosh machines extensively in our first year teaching. Accordingly, we have a heavy investment — both financial and in reliance on software — in operating systems and machines that cannot run Jade. This is a significant problem for us in contemplating the possibility of using Jade, and we feel that computer science educators at other universities would be in a similar situation.

5 The Cost of Change

In the sections above, we have discussed the issues pertaining to the suitability of a programming language for teaching beginning computer science students. In this section, we discuss issues relating to the *process* of changing the language used from one to another, highlighting the costs involved. This discussion is based on our recent experience in converting our first-year programme from Pascal to Java, and on other language changes we have made elsewhere in our curriculum.

To understand how a language change affects our courses, we must explain some background. Our courses follow a fairly conventional model, whereby we give lectures to present material, and have programming assignments for the students to gain practical experience. We assess the students on the basis of their practical work, but mostly on the basis of formal tests and examinations. We acknowledge this model is not perfect, but it does allow us to teach and assess a large number of students while keeping to a tight budget.

5.1 Lectures

Perhaps the most visible parts of our courses are the lectures. In this discussion we will restrict our attention to our first-year courses designed primarily for computer science majors. These courses have three lectures a week for 12 weeks, and we have two first-year courses, for a total of 72 lectures. In the first of our courses we have recently been starting with about 300–350 students, and in the second course we start with about 150–200. The drop in numbers reflects many reasons, including students with other majors who only wanted the introductory course, students failing the first course, and students deciding midway that computer science was not right for them.

Each of the 72 first-year lectures would typically involve some use of the programming language, whether it be actually teaching the language, working through examples, or explaining higher level concepts that the language supports. Each week we also have at least one tutorial session where we present more examples and go over problem areas. Lectures at first-year level require careful preparation, partly because the students have a diverse and limited background, and partly because the large class sizes and tight timetable forces us to keep to schedule.

It is not uncommon to spend an entire work day preparing a single lecture. When lectures are well prepared, we can often use them for several years, but our fast-moving subject matter limits this. In particular, a language change can force us to re-prepare many or all of the lectures in these courses. We also periodically change the academic staff who teach the courses. As a university, promoting the link between research and teaching is an important part of our philosophy, so new lecturers will often adjust course material to fit their own approach to the subject matter.

5.2 Assignments

While the lecture material might be the most visible part of our teaching, lectures usually play a pre-structuring role, and the real learning happens more when our students work through our programming assignments. We work on a weekly cycle of assignments, whereby students receive the assignment handouts and any prepared code, spend some time in a supervised laboratory or tutorial situation to get them started, and they then work on the assignment throughout the week, and submit it for assessment at the beginning of the next cycle. Our laboratories and tutorials are small groups of about 20–30 students, supervised by a tutor. Tutors are also available at times throughout the week to help students working on their assignments.

Each assignment typically consists of a few small programming exercises, often beginning with prepared code, and increasing in scale and complexity as the course develops. The assignments should typically be taking students about 6 hours to complete, though some students take more time and some less. Of course, the students have other courses they are doing at the same time, and total student workload is aimed to create a working week of about 40 hours.

Creating a weekly assignment can take a significant amount of time, most importantly to craft the prepared and expected code to provide the right level and focus of programming exercise for the students. It can even take a significant amount of time to write a a carefully worded handout document, particularly for early assignments where the students need detailed guidance in using the development environment. Creating each exercise could easily take several days total work. Once again, a well-written assignment can be used or adapted minimally for several years, but change, especially language change, can force us to develop new assignments. Sometimes we can adapt exercises to new languages and new environments, but this is often impractical.

The issues discussed above, lectures and assignments, involve the bulk of our teaching materials that will be directly affected by a language change. We also prepare formal tests and exams, but they must always be changed significantly every time we teach the course, so a language change does not affect that work in a major way.

5.3 Learning for Teaching

While lectures and assignments are the main artifacts of our courses, our real investment is in people. In our first-year courses, we will often have 3–5 academic staff directly involved in a significant amount of lecturing at some time. Also, we have tutors to run the laboratories, tutorials, provide help during assignments, and to assess assignments. Our tutors are usually chosen and trained from the best of our own senior students, who do the work on a part-time basis. For these first-year courses we would typically have 20–30 tutors. We also have programming staff who will provide technical support for the software and its platform. All these people need to learn the new language beforehand, and learn it well enough to teach, support, and to develop examples and assignments, and to help students with programming problems.

Organising all this requires a significant investment. Academics and technical staff are well used to mastering new programming languages by themselves, but it still takes time and requires working through programming exercises. The tutors need more support in learning. When we changed from Pascal to Java, we taught a three day course on Java to our tutors before the main course began. Many of these people will contribute to the courses for more than one year, and eventually some students in the courses will themselves become tutors. However, there is typically a residual effect that means a significant number of tutors need to learn the new language for the first two or three years.

5.4 Other Factors

So far we have discussed the impact on our first-year programming courses alone. However, the impact of changing the first language inevitably affects the whole curriculum. Even where the new language is not used elsewhere explicitly, and even when the paradigm of the new language is an appropriate introduction to languages used elsewhere in the curriculum, it will have an effect, because the student background will be affected. Students will see all programming coloured by the concepts essential to their first language. Lecturers will have to learn to explain new concepts in terms of the new language, even if they do not intend to use it themselves.

Another implication to consider is that new software for the language and programming environment must be installed and managed on our computers. This can be an arduous process or reasonably straightforward, depending on details of the goodness of fit between the software and the machines and operating systems in use. It should certainly be thoroughly investigated well in advance to anticipate difficulties. Many students, even at first-year, also expect to be able to use such software on their home computers. Making this possible and straightforward, and handling difficulties that arise, is also a consideration to remember.

We must also not forget the direct financial impart of a language change. The software for the language development environment must be obtained, and license arrangements worked through. Even determining what cost *applies* can be difficult in a heavily networked environment with multiple machine architectures and multiple operating systems. The cost to students to run the software at home is also a factor, not only because this is important to the students themselves, but because students ability to use their home computers affects the pressure on own laboratories.

To sum up, the costs of changing languages for first-year programming courses are large and diverse. Lectures and assignments must be re-created; tutors must be trained; software must be installed and managed; and the impact can carry on elsewhere. Perhaps surprisingly, universities typically budget very little, if anything, for this kind of change, and we end up relying on the goodwill of staff. University administrators do not typically understand the implications of a subject that moves as fast as computer science. The overall result is that changing first languages is something to be done with extreme caution, and only after careful planning about the inevitable costs.

6 Conclusions

There are a number of programming languages that are suitable from the point of view of language issues for using in an introductory programming course. However, changing a programming language in an introductory programming course can have considerable cost, and therefore considerable risk. We must ensure that the benefits justify taking that risk when considering any change.

In the case of changing to Jade, the risk from using it from the language point of view seems small. There are some aspects of the language that cannot be avoided but will require careful explanation to students. However, this is no worse than any other language. On the other hand, the benefits gained from moving to Jade also seem small. Jade is a fairly simple language, which helps from a pedagogical view, but otherwise does not seem to make the process of teaching programming significantly easier.

The risk from changing to Jade from the environment point of view is significant. However there are also many potential benefits in this regard. (Having access to the developers of the system can only be regarded as a good thing!) Exactly what the risk is depends on the institution's philosophy and current environment. In the case of VUW Computer Science, the cost associated with changing platforms are too great to consider such a change at the moment. However, for institutions that already have the necessarily hardware and technical support for the platforms that Jade is available for, Jade has to seriously considered.

References

[1] Peter Andreae, Robert Biddle, Gill Dobbie, Amy Gale, Linton Miller, and Ewan Tempero. Experience teaching CS1 with Java. Technical Report CS-TR-99/3, Victoria University of Wellington, March 1999.

[2] Robert Biddle and Ewan Tempero. Simulating multiple inheritance in Java. *Journal of Systems and Software*, To appear, 2000.

[3] Bevan J. Clarke. *An introduction to object-oriented systems development with Jade.* Cardinal Group Limited, 1998.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[5] JADE — the complete OO development tool. URL: http://www.discoverjade.com/.

[6] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.