The problem of teaching object-oriented programming

Part II: Environments[1]

Michael Kölling
School of Computer Science and Software Engineering
Monash University

mik@csse.monash.edu.au

1 INTRODUCTION

In last month's edition of JOOP we started a series of columns discussing the problems with teaching object-oriented programming to first year students. We talked about requirements for an object-oriented teaching language and analysed how close different languages come to fulfil these requirements. We also mentioned one important aspect: the importance of the environment. In short: a suitable programming environment is crucial for the success of an introductory course. Of all the problems reported by educators connected to teaching object-orientation, problems with the environment used were the most frequent and the most severe. Because of this importance, we devote this month's column to the discussion of environments. Again, the context we focus on is the suitability of environments for first-year teaching.

Environments for teaching and the requirements for those environments are much less discussed in the literature than languages. It seems that, when academics argue for or against a particular teaching language, a lot of energy is spent on examining language features, and comparatively little on discovering benefits or drawbacks of the programming environment. The reason might be partly historic: for early languages there were no substantially different environments. If a department used a Unix machine, for example, it was clear that the environment would be the Unix shell.

This has changed considerably within the last decade. First of all, substantially different environments have become available. This development was to a large degree driven by the spread of integrated graphical environments on PCs. Today the same language can be used from a command line prompt or from within an integrated graphical environment, creating vastly different programming experiences.

Secondly, better environments have become necessary. Earlier introductory courses focused on the development of algorithms in procedural or functional languages. To do this, an editor and a compiler was all that was needed for the practical part of the work. Modern courses now use object-oriented languages and subject material taught includes testing, debugging and code reuse. This creates the need to deal with multiple source files and multiple program development tools from the very start. To give a beginning student a chance to cope with this increased complexity, better environment support is needed.

---

[1] This paper has been published as: Kölling, M., "The Problem of Teaching Object-Oriented Programming, Part 2: Environments," *Journal of Object-Oriented Programming*, 11(9): 6-12, 1999.

As object-oriented languages became more popular, developments in environment research and languages were brought together and integrated environments for object-oriented languages emerged.

In examining the requirements and the suitability of environments for our task, two aspects are of particular significance: support for object-orientation and suitability for teaching.

2 REQUIREMENTS FOR A TEACHING ENVIRONMENT

We can summarise the requirements in seven keywords:

1 *Ease of use*
2 *Integrated tools*
3 *Object-support*
4 *Support for code reuse*
5 *Learning support*
6 *Group support*
7 *Availability*

Each of these aspects will now be discussed in more detail.

Ease of use

One of the most important factors in deciding about the suitability of a software development environment for teaching is ease of use. The environment must be easy enough to manage for inexperienced students to be useable for programming tasks after a very short time. This virtually implies that the environment must have a graphical user interface. The tasks that have to be managed by the environment internally are quite complex: file management, editing, compilation, management of compilation dependencies, debugging, browsing, testing, execution, etc. Even with this incomplete list it is clear that the available options are not trivial to master. Relying on a textual interface in non-trivial systems has been shown to require a steeper learning curve, especially for inexperienced users.

Ease of use also means the hiding of unnecessary detail. Operating system aspects, for example, such as file management details, should be managed by the environment automatically, while letting the user work at a higher level of abstraction. When a user creates some classes, for instance, there is little interest in the number of files and in which subdirectories information about these classes are stored. In a good environment, it should be possible to operate for a long time without the need to be proficient in the use of the underlying operating system. This would avoid a common problem for first year courses: while the subject of the course is programming, the first two or three weeks are actually spent mainly on explaining operating system commands. This is necessary to get to the point where students know enough to use the computer system to type in, compile and execute their programs.

Most existing systems fail to meet our ease-of-use requirements. They typically suffer from one of two common problems: too little support or too much support. Both make them unsuitable for teaching first year students.

The first category is that of systems that rely on a loose collection of (at least partially) text based tools. They provide a command line interface (typically Unix or DOS) and little integration. An editor and a compiler are used as separate tools, which communicate through the user: the compiler uses line numbers to indicate locations of errors, and the user must locate the corresponding lines herself in the editor to correct the error. A rich set of file management commands is needed (listing, copying, deleting, etc.). Sometimes some of these tools provide graphical interfaces (an editor or a debugger). Nevertheless, the interface of the separate tools is typically different enough that the use of each additional tool is a major task that requires a significant amount of separate instruction and practice. The effect of this usually is that some tools, such as debuggers or class browsers, are not used in the first year, because the time required to become familiar with them is not available.

The second category of available systems is that of very sophisticated integrated environments. These systems are often very powerful: integration between components can provide new functionality and a unified “look and feel”. The user interface is typically graphical.

The problem with this second category of system is that those that are available today are all developed for professional software engineers. They require experts to use them. Typically dozens, often hundreds, of functions are available to the user for many sophisticated tasks. For a professional software engineer it can be worthwhile to spend several months becoming an expert in the use of a powerful system, since it can pay off in increased productivity or quality of work over many years to come. For a teaching situation with beginners it is impossible. A system that is too powerful can be as unusable as a system that is too weak.

What is needed is a system that is simple enough to be useable for beginners after a very short time, yet powerful enough to provide many of the tasks of software development easily or automatically.

Integrated Tools

The requirement of tool integration is a direct result of the ease-of-use requirement. Above, we have already hinted at some advantages of an integrated system. The integration of tools can have many benefits:

- *A unified interface*. If the interface of all components appears in a unified way, the use of additional components is easier to learn.
- *Smaller interfaces*. Often, one tool can make use of an interface already provided by another tool. A debugger, for instance, can use the text display provided by an editor to mark the current execution position, or to let the user specify the location of a breakpoint. This results in fewer different interface components for the user to learn.
- *Increased productivity*. Integration can easily provide shortcuts for the edit – compile – execute cycle. If, for example, the compiler detects an error, it can open an editor window and highlight the corresponding location in the source code. The step of finding the line and column of the error can be automated. The same can be done for runtime errors if the execution of the program is integrated as well (e.g. via a virtual machine).
- *Better functionality*. Often, a tool can provide additional functionality by making use of information that was generated by another tool. An editor can,

for example, provide language dependent functions (such as structure editing or pretty printing) by making use of the parser in the compiler (without writing a second parser, as is done in some systems). Even more interesting in our context is the automatic generation of class interface views in an editor by using symbol table information generated by the compiler. This goes far beyond the ability of a general purpose (non-integrated) editor, especially since it involves information from other files (such as inherited routines).

Systems that support the use of non-integrated tools have some advantages as well. The main one is the ability to use previously known tools for a common task, thus avoiding the need to learn to use a new tool. The most common example is the ability of an environment to let users use a text editor of their choice. Many experienced users have become very familiar with one editor and resent being forced to use another one.

A second advantage may be the availability of many general purpose tools for a system that is used as the basis for a non-integrated environment. Unix, for example, provides a large number of tools (e.g. for text formatting, counting words, printing, etc.) that might be used if a Unix shell is chosen as the development environment.

These advantages, however, do not outweigh the advantages of integrated systems. For beginners especially, the arguments for non-integrated systems are weak. First year students typically are not experts in the use of a particular text editor, and they do not know a relevant number of tools on a given platform. And even for more experienced users the disadvantages of an integrated system can be made less serious by providing a good environment: a text editor, for instance, that can be adapted to users' preferences with regard to key bindings, or a good set of tools which are easy to learn. The advantages of integration, on the other hand, benefit all kinds of users.

Object support

Many existing software development environments have evolved over time. Most were originally developed for non-object-oriented languages and later adapted for object-orientation. This adaptation, however, usually fails to exploit the possibilities that come with object-orientation – they are, in their character, still structured (not object-oriented) environments. While there are numerous environments for object-oriented languages, few of them are object-oriented environments. This difference is important to understand.

Traditional procedural development environments facilitate the design and construction of *programs*. The basic entity they operate on is source code and their functionality revolves around the convenient manipulation of source code. The ultimate goal is to produce a program, an algorithm description with exactly one entry point that can only be built and executed after all its parts are (in some sense) completed. No notion of runtime objects exists within the environment, since those objects cannot exist independently from an active execution of a program. All data available outside the execution is in the form of files. Consequently, the environment has only to deal with source and data files.

When these environments were adapted to object-oriented languages, source files were replaced by class descriptions. Typically more source files exist in the object-oriented environment than in the procedural form. In addition, these files have more

relationships with each other, e.g. usage and inheritance relations. Tools have been added to the environment to manage these class files and some of their relationships. The general paradigm of the environment, however, has not changed. The environment is still used to build an application with exactly one entry point that can be compiled and executed only after all its parts have been completed. This is the program-oriented paradigm.

The object-oriented paradigm is based on the idea that objects exist independently of each other, and that operations can be executed on them. Consequently, a user in a true object-oriented development environment should be able to interactively create objects of any available class, manipulate these objects and call their interface routines. The composition of objects at the user level should be possible. This has also been referred to as an *instance-centred environment* [1].

Such a facility, if provided, leads to the possibility of incremental application development, familiar from Smalltalk systems. Any individual class can be tested independently as soon as it is completed. Testing then becomes much more flexible than in procedural systems. In most current object-oriented environments, objects have to be wrapped in a non-object-oriented main function or script to create and invoke the first object or objects. A direct call of object interfaces is not possible, since object instances are not supported by the environment. In short: the programmer must fall back to the procedural paradigm to start and test a program. Thinking in two mind-sets is required: one for thinking about the model of the application itself and one for thinking about environment interactions.

To avoid this problem, classes and objects should be the main abstractions used for user-level interaction in the environment. They should be treated as user level objects on which the user can perform operations by interacting with them through their interfaces.

Support for code reuse

The facilitation of reuse of existing code is one of the main motivations for object-oriented programming. In teaching, we must aim at giving our students a realistic impression of the task of programming. We must try to include as many real programming experiences into students exercises as possible, for students to understand the issues of software development and to form good habits.

Because of this, it is important to facilitate the reuse of existing code from the very beginning. The development environment must provide a class browser that lets a student find out about existing library classes. It should also have the ability to build new libraries of classes that the student has written. These can then be reused later by himself or by other students. This is important to enable students to experience the need to write code for reuse.

Learning Support

The environment must support some techniques that are known to support learning of programming concepts. Among those are:

Interaction / experimentation

To provide a hands-on approach by enabling interaction with classes and objects can greatly increase the understanding of object-oriented concepts. The ability to experience the notion that, for example, many objects may be created from one

class, and that those objects behave similarly, but have a separate identity and a different state, can greatly help in clarifying the relationship between classes and objects. Equally, the experience of stepping through code and seeing the effect of control structures and watching the value of variables change can contribute a great deal to the understanding of writing code.

Visualisation

The structures talked about when teaching object concepts should be, as far as possible, made visible on the screen. It is a common experience of teachers teaching object-oriented programming, that it is initially difficult for some students to think in terms of classes. The reason for this might be that all they ever see on the screen are lines of code. If students are expected to think and talk about classes, then the classes (and their relationships) should be visually represented. The same can be said about objects at runtime: relationships between objects are better understood when made visible.

An environment should make use of both graphical and textual representations. Studies (e.g. [2]) have shown that neither text nor graphics can be considered generally superior for representation tasks. Both have their place. The advantage of adding a graphical notation is the ability to provide richer secondary cues. Graphical secondary notation (such as proximity, grouping and white space) can offer immediate access to information that is difficult to extract from the textual representation.

Manipulation using the graphical or textual representation of a class should be possible interchangeably. This means, for instance, that it should be possible to graphically edit the inheritance relationships of the classes in an application. The changes made graphically should automatically be reflected in the source code of the classes. The same should work the other way around: if a class is specified as an ancestor in the source code of another class, this relationship should automatically be reflected in the graphical representation.

Visualisation should not be confused with “visual” GUI building. Some development environments, such as Visual C++ or Delphi, use graphical support only to build the user interface of an application, but neglect the internal structure of the program itself. This is not the kind of graphical support that is helpful to beginning students. The use of a graphical user interface builder might be helpful to construct good looking programs (which can be a positive factor for motivation), but it should only be a second step in the process of learning how to develop computer programs. The first and fundamental issue is to understand the abstraction process – the underlying structure used to model the problem domain. The graphical user interface detracts from these issues and conveys a completely misleading picture about the character of object-oriented programming.

Group Support

Another characteristic of programming in the real world is the need to work in teams. The success of object-oriented languages is partly due to their advantages in group projects. Ideally, we also want to teach our students about the techniques needed for teamwork. To do this, it is essential that the environment has some form of support for group work.

It should be possible for a group of students to simultaneously work on different parts of a system under development, and to have a controlled system integration process. Some form of group communication mechanism would also be beneficial.

Availability

Some of the requirements mentioned so far are implemented in existing CASE tools. There is, however, one serious problem with some of those tools: cost. Being developed for professional use with large companies as customers in mind, the pricing of some of those tools is outside the reach of many university departments. In addition to that, some of them require very sophisticated hardware (either very fast machines or large memory). In order to be useful as a university teaching language, a system must be available at reasonable cost, and it must be able to run on commonly available hardware.

None of the requirements named above is really new. Each of them has been implemented in some existing system. The combination of these requirements, however, is what is really important in our context. In particular the combination of the requirement of ease-of-use with that for fairly sophisticated technical support may at first glance seem contradictory. We are asking for a powerful system that appears simple to the user. The degree to which this combination is achieved will be the determining factor in assessing the suitability of systems for first year teaching.

We now evaluate some existing environments in more detail.

3 EXISTING ENVIRONMENTS

As object-oriented languages have grown in popularity there have been attempts to bring together environment and object-oriented technologies [3-5]. Environments were developed that support the development of programs in object-oriented languages, sometimes supported by object design tools. In most cases the approach has been to adapt an existing software development environment to an object-oriented language. Such attempts have not, in general, managed to capture the potential advantages offered by object-orientation.

We cannot, in the scope of this column, discuss a comprehensive list of environments in detail. There are too many of them, and new ones appear every month. Individual environments should be judged against the principles we have discussed above. We will, however, comment on some individual environments as an example of some of the most common shortcomings of popular systems.

Smalltalk environments

Some of the most interesting systems in our context are Smalltalk environments. They provide a browser to support the use of the library classes. Programming as a process of modelling with a combination of user-defined classes and reuse of existing libraries is presented elegantly and in a very consistent manner. They also provide the high level of interactivity and object support that we demand from a good teaching environment. The effect of this is highlighted by a quote from Adele Goldberg: “Smalltalk denotes fun and success. Developers learn how to create software systems by changing existing applications, and redefining existing tools.

Tangible results are immediate and rewarding" [6]. This immediate feedback and reward and the sense of fun created by it represents the most powerful support in teaching programming to beginning students for which a teacher can wish. Every teaching environment should try to create this sense of fun through interactivity and immediacy.

Smalltalk, however, lacks other important facilities: no visualisation tools for class relations are available. The main problem with this lies in the Smalltalk language itself: since it is not statically typed, it is not possible to extract usage relations from its source code. No indication exists before runtime as to the call relationships between classes. Inheritance relationships as shown in the browser do not present the relationships of one application but rather the whole Smalltalk environment and so the browser is not used as an application modelling tool. Smalltalk blurs the distinction between the environment and the application under development.

Reports about the use of Smalltalk systems for teaching also point to another problem: its size. While the language itself (in terms of the number of constructs) is small, the class library and tools are large and often confusing. Several authors reported difficulties with the students ability to cope with the environment [7, 8], especially that experimentation and self directed learning was not working well because students were overwhelmed by the system. They also found that the functionality of the browser should be limited, since its power and flexibility caused more problems than it solved.

EiffelBench

One environment that comes close to our requirements in its stated objectives is EiffelBench, an integrated development environment for the Eiffel language. Eiffel and EiffelBench have the advantage of not being derived from a procedural predecessor. The developers have thought about appropriate mechanisms for working in an object-oriented context. Meyer, in a description of EiffelBench [5], discusses in detail the question of what characteristics an object-oriented environment should have. He states five principles, most of which we agree with. The realisation of these ideas in EiffelBench, however, seems to violate most of his own principles. Meyer's principles are:

1 *Method-environment consistency*: A development environment meant to support a particular method or language must rely on a consistent set of user interaction conventions which closely parallel the concepts promoted by the method or language.

2 *Data abstraction*: In an object-oriented environment, the basic way of working must be through direct manipulation of visual representations of developer abstractions.

3 *Object-oriented tools*: In an object-oriented environment, each tool must be based on an object type (not on a type of operation).

4 *Semantic consistency*: An object-oriented environment must enable its users, for any symbol (textual, graphical, or otherwise) representing a development object in the user interface, to select the object through its symbol and apply any operation that is semantically valid for the object, regardless of the symbol's context.

5 *Typed environment*: In an object-oriented environment supporting a statically typed language and methods, the environment's visual conventions should display and enforce the type constraints on development objects.

While these principles are a good guideline, they are not realised well in the EiffelBench environment (although the developers claim that they are). Principle 2, for example, is a very important one: developer abstractions should be directly manipulatable. Yet EiffelBench offers no facility to interact with object instances – clearly one of the most important developer abstractions. The objects the environment recognises are development entities, such as classes and variables, but not the objects with which the application under development is concerned. And even this restricted set of objects is handled in an inconsistent way. The interaction mechanism passes object representations to object tools – a mechanism that much more closely resembles the passing of the object as a parameter than the invocation of an operation on the object. The environment interaction style therefore is procedural, not object-oriented.

Principle 1 demands method-environment consistency, yet at the language level operations are invoked by selecting a method from the object's interface, while in the environment it is done by passing the object as a parameter to a tool. Principle 5 touches on a related subject: the type in the language is represented by its class, which, in turn, is defined by its name and features (the interface). If operations were selected directly from the interface of each developer object (rather than passing the object as a parameter) then the type check would be implicit (no routine could be called that is not in the interface). The requirement demanded in principle 5 is, in fact, only necessary because EiffelBench does not support an object-style invocation of operations. The mechanism used in EiffelBench also makes it necessary to invent an explanation for other apparent inconsistencies. A routine, for example, can be passed to the class tool (an apparent type violation). This is explained with inheritance-based conformance, although it is hard to see how a routine is a descendent of a class. Overall, the principles stated here highlight the right problems, but the realisation in EiffelBench fails to properly achieve these goals.

Another problem is the complexity of its interface. Students (and many experienced programmers) generally have difficulties coming to grips with the meaning of the many interface controls, most of which are labelled with obscure icons that do not convey any intuitive meaning. Several implementation issues also arise. After using it for several years for teaching, Nørmark [9] described it as low in quality and unreliable. He names compilation speed, incorrect compilation, primitive text editing capabilities and a difficult-to-use interface as some of the problems.

Borland C++

Borland C++ is a typical example of an environment that evolved from an earlier, procedural predecessor and inherited many of its characteristics. It has been extended to support object-orientation, but in many cases this appears as a half-hearted, second-best solution. For this review we evaluated Borland C++ version 5.0.

The environment is based around a project which, in turn, is a collection of files. Files can be made to loosely correspond to classes, but nothing enforces such a relationship. Also, as is the convention in C++, classes typically consist of at least

two files (a header file and an implementation file). The problem becomes apparent when we consider the connection with the class diagram. Borland C++ provides a simple, automatically generated diagram showing classes involved in the current project. This diagram cannot be edited, but a class name can be double-clicked to open its source. This operation, however, only finds and opens the header file – it is left to the user to find the implementation.

Borland C++ provides no support for runtime objects or testing. The implementation is stable, although it still suffers from a number of errors (such as occasional mistakes in its dependency analysis and resulting incorrect compilation).

It uses the Microsoft help system as a replacement of a class browser. As a result, the class documentation is separated from its implementation. This becomes a problem if teachers add their own classes to libraries – no help is available for those through the normal tools. Borland's help texts are often minimal, full of jargon and do not include fundamental information. The system is difficult to navigate and students typically do not manage to use the help system in their first semester.

Visual Age

Visual Age is an integrated development environment developed by IBM which is available for different languages, including Smalltalk, C++ and Java. The environment evaluated for this review was Visual Age for Java, version 1.0.

Visual Age borrows much of its interface principles from early Smalltalk environments. It uses a browser that can display the packages and classes in the system. The browser is divided into panes very similar to those known from Smalltalk-80: one pane shows a list of packages, another one a list of classes in a selected package, a third the functions in the selected class. A bottom pane displays the code for one selected function. The user always views and edits one function at a time. As soon as the function is saved it is automatically compiled. This is intended to create an "immediate execution" environment: as soon as a class is written, it can be executed. All of this is very similar to the interface of Smalltalk environments. There are, however, some differences.

One immediately noticeable difference is compilation speed. Because compilation is automatic and fairly slow, the user is regularly interrupted and forced to pause to wait for the system to catch up. When a data member is entered into a 20-line class, for example, the system is unresponsive for more than 30 seconds while automatic recompilation takes place (measured on a 120Mhz Pentium/32Mb machine running Windows 95). Overall, the speed of the system (or lack thereof) becomes a serious annoyance when trying to work with it for an extended period.

Visual Age also inherits most of the complexity from Smalltalk, and adds additional complexity of its own. The browser is used to display an overview of the complete Java universe (merging the view of the current application with the view of all existing class libraries), just as Smalltalk does. It offers several different views of this universe as well as different specialised browsers. Overall, the functionality far exceeds what is practical to be learnt by first year students. It is evident from the interface alone that Visual Age is aimed at a professional programmer who needs to spend considerable time to familiarise herself with the environment.

Object creation and interaction is not supported in the environment. The tools are aimed at generating one monolithic application which can then be executed as a

whole. A graphical application structure is not provided. Classes in the libraries and the application are presented as a nested list.

Overall, Visual Age is a typical example of a system that does not support the object concepts at the environment level, and is clearly aimed at professionals, making it much too complex to use for first year students. It cannot be considered suitable for introductory education.

Visual C++

Visual C++ is an integrated development environment from Microsoft Corp. It exists in almost identical form for other languages (e.g. Visual J++, a Java version). The language independent part of the environment is sometimes referred to as the *Microsoft Developer Studio*. We evaluated Visual C++, version 5.0.

Visual C++ is clearly a very mature environment with many well thought out features. It has a highly flexible user interface which can be easily and quickly customised for personal preference or specific tasks. Toolbars can be easily enabled or hidden, they can be present in free floating windows or at fixed screen locations. Output can often be presented in separate windows or in panes arranged in a single window, with easy and flexible pane arrangement. Extensive help is provided in easily accessible format.

Unfortunately, though, this maturity is only present in the traditional areas which were already present in procedural environments, not in specific support for object-orientation. All the areas that we emphasised most in our requirements, visualisation of class relationships, object support, direct object interaction and ease-of-use, are not supported well (or not supported at all) in this system.

Classes are displayed in a list, and no class relationships are graphically shown. Objects cannot be interactively created or invoked – they do not exist as an abstraction in the environment. Extensive file management is necessary to set up a project. The environment includes a large number of functions and options which are only interesting for professional development and have a strong intimidating effect on beginners. Overall, it is obvious that this environment is aimed at a more expert user who needs the full flexibility of professional software development and is expected to spend considerable time learning to use the system. The target group clearly is not first year students, and trying to use it in such an environment would lead to complexity problems without offering the advantages of good object-oriented support.

4 CONCLUSION

The programming environment is an important, often neglected, part of the programming experience. Some of the main requirements for the environment are similar to those for the language: it should present the underlying concepts and tools in a consistent manner, and it should not be overly complicated to use. This means in particular that we need an object-oriented environment which is simple enough to be used for teaching.

Many of the environments on the market today are not object-oriented. They support object-oriented languages, but they fail to support and exploit object-orientation at the environment level. It is essential that we provide a more appropriate

environment if we wish our students to produce truly object-oriented programs and to capture the potential of object-orientation.

Those environments which offer good support for object concepts are too complex to be effectively used in a first year teaching course. Problems with environments have been identified as the most common problems with teaching object-orientation to undergraduates. Mazaitis [10], investigating this issue, concludes: “It seems that before pure languages, Smalltalk and Eiffel, are more widely accepted, they must come bundled with support tools tailored to students’ needs.”

Providing a good object-oriented teaching environment is not simple. After having discussed the problems with languages and environments and criticised existing systems, we will, in the next two issues of JOOP, present the Blue system, our attempt at a solution to these problems.

Acknowledgments

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

References

[1] Gold, E. and M. B. Rosson, "Portia: An Instance-Centered Environment for Smalltalk," in *OOPSLA 91 Conference Proceedings*, pp. 62-74, ACM, 1991.

[2] Petre, M., "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming," *Communications of the ACM*, 38(6): 33-44, 1995.

[3] Haarslev, V. and R. Möller, "A Framework for Visualizing Object-Oriented Systems," in *ECOOP/OOPSLA 90 Conference Proceedings*, pp. 237-244, ACM, 1990.

[4] Holt, R. C., "Introducing Undergraduates to Object Orientation Using the Turing Language," *SIGCSE Bulletin*, 25(3): 324-328, 1994.

[5] Meyer, B., "What is an object-oriented environment?," *Journal of Object-Oriented Programming*, 6(4): 75-81, 1993.

[6] Goldberg, A., "Why Smalltalk?," *Communications of the ACM*, 38(10): 105-107, 1995.

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

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

[9] 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.

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