An Objective Comparison of Languages for Teaching Introductory Programming

Linda Mannila
Turku Centre for Computer Science
Åbo Akademi University
Dept. of Information Technologies
Joukahaisenkatu 3-5, 20520 Turku, Finland
linda.mannila@abo.fi

Michael de Raadt
Department of Mathematics and Computing and
Centre for Research in Transformational Pedagogies
University of Southern Queensland, Toowoomba
Queensland, 4350, Australia
deraadt@usq.edu.au

ABSTRACT
The question of which language to use in introductory programming has been cause for protracted debate, often based on emotive opinions. Several studies on the benefits of individual languages or comparisons between two languages have been conducted, but there is still a lack of objective data used to inform these comparisons. This paper presents a list of criteria based on design decisions used by prominent teaching-language creators. The criteria, once justified, are then used to compare eleven languages which are currently used in introductory programming courses. Recommendations are made on how these criteria can be used or adapted for different situations.

Keywords
Programming languages, industry, teaching

1. INTRODUCTION
A census of introductory programming courses within Australia and New Zealand [5] revealed reasons why instructors chose their current teaching language (shown in Table 1). The most prominent reason was industry relevance, before even pedagogical considerations. This suggests academics perceive pressure to choose a language that may be marketable to students, even if students themselves may not be aware of what is required in industry.

The primary objective of introductory programming instruction must be to nurture novice programmers who can apply programming concepts equally well in any language. Yet many papers from literature argue that one language is superior for this task. Such research asserts a particular language is superior to another because, in isolation, it possesses desirable features [2, 3, 4, 9, 21] or because changing to the new language seemed to encourage better results from students [1, 11]. What is shown in literature is surely only a reflection of the innumerable debates that have undoubtedly taken place within teaching institutions.

While the authors of this paper do not believe that language choice is as critical as choice of course curriculum used to deliver teaching, it is important to choose a language that will best support an introductory programming curriculum.

Table 1: Reasons for instructors' language choice

Reason	Count
Industry relevance/Marketable/Student demand	33
Pedagogical benefits of language	19
Structure of degree/Department politics	16
OOP language wanted	15
GUI interface	6
Availability/Cost to students	5
Easy to find appropriate texts	2

1.1 Background
The choice of programming language to use in education has been a topical issue for some time. In the early 1980s, Tharp [22] made a language comparison of COBOL, FORTRAN, Pascal, PL-I, and Snobol, primarily focused on efficiency of compilation and speed of code implementation, in order to provide educators with information needed to choose a suitable language. Today, considerations focus more on pedagogical concerns and the range of languages is even broader.

George Milbrandt suggests the following list of language features for languages used in high schools in [20].

- easy to use
- structured in design
- powerful in computing capacity
- simple syntax
- variable declaration
- easy input/output and output formatting
- meaningful keyword names
- allowing expressive variable names
- provide a one-entry/one-exit structure
- immediate feedback
- good diagnostic tools for testing and debugging

Many of the criteria in the list above are echoed by McIver and Conway [15] who list seven ways in which introductory programming languages make teaching of introductory programming difficult. They also put forward seven principles of programming language design aiming to assist in developing good pedagogical languages. Neither of these studies demonstrates application of these criteria to make comparison between languages.

Instruments to facilitate the process of choosing a suitable language have also been suggested (e.g. [18]), but without presenting any comparable results.

1.2 Goal
This paper is intended to be an objective comparison of common languages, based on design decisions used by prominent teaching-language creators, drawing conclusions that allow instructors to make informed decisions for their students. It is also intended to provide ammunition for those who are, for pedagogical reasons, seeking to make a language change, in an environment where industry relevance can be overvalued.

The following section lists the criteria used to make a comparison of languages in section 3. Finally conclusions are drawn in section 4.

2. CRITERIA

A list of seventeen criteria has been created and is presented in the following subsections. Each criterion has been suggested by creators of languages that are considered "teaching languages".

1. Seymour Papert (creator of LOGO) [1]
2. Niklaus Wirth (creator of Pascal) [2]
3. Guido van Rossum (creator of Python) [3]
4. Bertrand Meyer (creator of Eiffel) [4]

Each criterion is drawn from the design decisions made by each of these language creators as they describe their languages.

The criteria refer to languages in general. There is no mention of paradigm within the criteria and this allows comparison of languages across paradigms.

Criteria are grouped into related subsections for ease of application. The criteria are shown in no particular order of priority.

2.1 Learning

The following criteria relate the programming language to aspects of learning programming.

2.1.1 The language is suitable for teaching

This first criterion was suggested by Niklaus Wirth [25]. Wirth points out that widely used languages are not necessarily the best languages for teaching.

> *The choice of a language for teaching, based on its widespread acceptance and availability, together with the fact that the language most widely taught is therefore going to be the one most widely used, forms the safest recipe for stagnation in a subject of such profound pedagogical influence. I consider it therefore well worth-while to make an effort to break this vicious circle.*

It is interesting that Wirth was able to break this cycle for almost twenty years, but how easily we have reverted to use of commercial languages for the same reasons.

This criterion is echoed by Guido van Rossum [23].

> *...code that is as understandable as plain English.*
>
> *...reads like pseudo-code.*
>
> *...easy to learn, read, and use, yet powerful enough to illustrate essential aspects of programming languages and software engineering.*

Bertrand Meyer also suggests this criterion [16].

> *In some other languages, before you can produce any result, you must include some magic formula which you don't understand, such as the famous public static void main (string [] args). A good teaching language should be unobtrusive, enabling students to devote their efforts to learning the concepts, not a syntax.*

☑ To meet this criterion the language should have been designed with teaching in mind. The language will have a simple syntax and natural semantics, avoiding cryptic symbols, abbreviations and other sources of confusion. Associated tools should be easy to use.

2.1.2 The language can be used to apply physical analogies

This criterion was suggested by Seymour Papert [17]. Papert believed physical analogies involve students in their learning.

> *Without this benefit [using students' physical skills], seeking to "motivate" a scientific idea by drawing an analogy with a physical activity could easily denigrate into another example of "teacher's double talk".*

This idea is extended to "microworlds", a small, simple, bounded environment allowing exploration in a finite world.

☑ To meet this criterion a language would need to provide multimedia capabilities without extension. Perhaps more critical is the effort needed to get students to a stage where they could access this potential and how consistently it is applicable across environments (say between operating systems).

2.1.3 The language offers a general framework

The primary goal of any introductory programming course is to introduce students to programming. As such, the language itself is not the focus of instruction and any skills learned in one language should be transferable to other common languages. Bertrand Meyer suggests the following philosophy [16].

> *A software engineer must be multi-lingual and in fact able to learn new languages regularly; but the first language you learn is critical since it can open or close your mind forever.*

☑ To meet this criterion the language should make it possible to learn the fundaments and principles of programming, which would serve as an excellent basis for learning other programming languages later on.

2.1.4 The language promotes a new approach for teaching software

In an introductory course, language is but one part of the learning for a novice. It may be valuable where a language itself and associated tools can assist in learning to apply the language. Bertrand Meyer [16] suggests an introductory 'programming language' should be...

> *...not just a programming language but a method whose primary aim — beyond expressing algorithms for the computer — is [to] support thinking about problems and their solutions.*

☑ To meet this criterion the 'language' should not only be restricted to implementation, but cover many aspects of the software development process. The 'language' should be designed as an entire methodology for constructing software based on 1) a language and 2) a set of principles, tools and libraries.

2.2 Design and Environment

The following criteria describe the aspects of the language that relate to design and the environment in which the language can be used.

[1] http://www.papert.org/

[2] http://www.cs.inf.ethz.ch/~wirth/

[3] http://www.python.org/~guido/

[4] http://se.ethz.ch/~meyer/

2.2.1 The language is interactive and facilitates rapid code development

The potential to apply new programming ideas without requiring the context of a full program is valuable to novices [17]. The possibility to quickly start writing (and understanding) simple programs motivates and inspires [23].

- ☑ To meet this criterion the language and environments supporting its use should allow novices to implement newly acquired ideas, without having to establish the context of a full application. The language environment should provide students with interactive and immediate feedback on their progress.

2.2.2 The language promotes writing correct programs

The language Eiffel implements "Design by Contract" – a set of concepts tied to both the language and the method [14]. The aim is to move away from the prevalent "trial and error" approach to software construction.

> *By equipping classes with preconditions, postconditions and class invariants, we let students use a much more systematic approach than is currently the norm, and prepare them to become successful professional developers able to deliver bug-free systems.*

- ☑ To meet this criterion, students should be given ways to ensure that the code they write is correct and does not contain bugs.

2.2.3 The language allows problems to be solved in "bite-sized chunks"

It is desirable for any programmer to be able to focus on one aspect of a problem before moving onto the next. A language which supports problem decomposition is desirable as suggested by Papert [17].

> *It is possible to build a large intellectual system without ever making a step that cannot be comprehended. And building a system with a hierarchical structure makes it possible to grasp the system as a whole, that is to say, to see the system "viewed from the top".*

- ☑ To meet this criterion the language should support modularization, in functions, procedures or equivalent divisions.

2.2.4 The language provides a seamless development environment

When a novice begins to program it is valuable to understand the process that takes their program source to an executable program. Some Integrated Developments Environments can hide these details, obscuring this process for the sake of simplicity and rapidity which may be advantageous for an expert but less so for a novice. Other environments can assist in bridging the gap between design and implementation by, for example, converting architectural diagrams to code, and possible also reversing this process. Meyer suggests a "seamless development" can aid novices [16]. Such a language…

> *...enables us to teach a seamless approach that extends across the software lifecycle, from analysis and design to implementation and maintenance.*

- ☑ To meet this criterion the development environment should have an intuitive GUI for design and implementation which provides access to features and libraries, both for basic and advanced programming.

2.3 Support and Availability

The following criteria describe the support community and availability of the language and resources to teach the language.

2.3.1 The language has a supportive user community

Whether a language is a commercial creation or an open source project, its longevity will depend on the support for that language in the wider programming community. Where support is limited, resources and support may be a restriction for instructors and students. This criterion is suggested in [23].

- ☑ To meet this criterion, there must be sufficient support for students, faculty and others interested in learning and using the language. This support can come in different forms, such as web pages, course books, tutorials, exercises, documentation and mailing lists.

2.3.2 The language is open source, so anyone can contribute to its development

One of the benefits of an open source software project is the reduction of cost. Another benefit of an open source project is interoperability – where a commercial venture may seek to avoid compatibility with other systems to create a reliance on their creation. Beyond requiring a standard on which the language is based, this criterion seeks to differentiate languages whose development is the collaborative product of individuals. This criterion is suggested in [23] and continues over the following two criteria, even though the following criteria may also be applicable to languages outside the open source world.

- ☑ To meet this criterion the language should be the invention of a group who do not seek to create a commercial product and to which anyone can contribute if they wish.

2.3.3 The language is consistently supported across environments

Programming is conducted within different operating systems and on different machines. It is useful to be able to offer tools to students, which can be used in many environments rather than just one. This gives access to students regardless of location or setting.

- ☑ To meet this criterion the language should be available under various platforms.

2.3.4 The language is freely and easily available

For students, cost can be a significant issue. Students who are unlikely to continue programming beyond an introductory exposure will see little return from an expensive language or IDE.

- ☑ To meet this criterion the language should be free from subscription or obligation and available worldwide without restriction.

2.3.5 The language is supported with good teaching material

It is beneficial for both instructors and students if teaching materials are available for a particular programming language.

These can provide alternate perspectives and suggest appropriate curricula. This criterion was suggested by Meyer in [16].

- ☑ To meet this criterion, current textbooks and other materials should be available for use in the classroom.

2.4 Beyond Introductory Programming

The following criteria describe considerations beyond an introductory course. It is useful to consider how well a language can be used into various levels of a computing degree program as learning new languages, although valuable, can be a costly exercise if every new course requires its own language. As such an introductory language should also be examined from the perspectives of advanced levels of undergraduate programming and the programming industry. Moreover, using a language which is applicable in other contexts beyond introductory programming allows students to explore real world application domains using a powerful language and environment. This is especially relevant to students who wish to learn more, or at a faster pace.

2.4.1 The language is not only used in education

Students may be more motivated by a language that is not simply used within an educational setting. This criterion was suggested by van Rossum in [23].

> *...suitable for teaching purposes, without being a "toy" language: it is very popular with computer professionals as a rapid application development language.*

- ☑ To meet this criterion the language should also be relevant in areas other than education, e.g. in industry, and be suitable for developing large real world applications.

2.4.2 The language is extensible

Novices may not be expected to write extension modules within an introductory programming course, but using existing language extensions has potential to make the learning experience more motivating and exciting. Using modules, teachers can tailor tuition according to the interests of the students, allowing them to accomplish more than with the base language alone. Extensibility also allows a language to be applied to a larger variety of problems later in learning and professional use. This criterion is suggested in [23].

- ☑ To meet this criterion a language, which can be effectively used with only a small integral subset of features, should make it easy to access advanced functionality that is not directly accessible.

2.4.3 The language is reliable and efficient

Compilation speeds no longer seem to be as much of an issue as they were in 1971, when Wirth announced Pascal [25], however the ease with which a novice can take their source and produce an executable is still relevant.

> *...dispelling the commonly accepted notion that useful languages must be either slow to compile or slow to execute, and the belief that any nontrivial system is bound to contain mistakes forever.*

This is balanced by the need to involve the novice in this process and facilitate debugging.

Speed of execution still differentiates some languages, for example a distinction can be made between the products of the procedural and functional paradigms because of how closely each relates to the model execution used by processors. It could be argued that novice programmers rarely use the potential for speed in a language; however it could equally be argued that an academic setting is the perfect place to explore such limits.

It almost seems unforgivable that any compiler or environment for programming could be, in itself, flawed. Perhaps with modern 'bloated' industry languages, complexity within monumental libraries can bewilder novices.

From a pedagogical perspective, this criterion has a low priority in relation to other criteria. However, the decisions made by instructors are never purely motivation by pedagogy.

- ☑ To meet this criterion the language must be useful in creating high speed applications.

2.4.4 The language is not an example of the QWERTY phenomena

Papert suggests some languages continue to be used because of historical reasons and the justification for this continuation is often manufactured [17]. He defines the QWERTY phenomena in relation to the QWERTY keyboard.

> *There is a tendency for the first usable, but still primitive, product of a new technology to dig itself in.*

- ☑ To fulfill this criterion the language must show its usefulness now and into the future beyond its applicability in the past.

3. LANGUAGE COMPARISON

With criteria given it is possible to compare languages in an objective fashion. It should be stated that construction of such criteria suggests the biases of the authors of this paper. By choosing other inspirational figures, another set of criteria could have emerged and even within the works of the prominent figures chosen here, new criteria could have been promoted and others given less prominence. Also, any application of these criteria is subject to the authors' judgment.

Not all criteria can be applied equally to each language. For instance some languages are defined as a standard which is implemented by multiple groups in the form of compilers. Such a language may or may not be accompanied by additional tools in its delivered form. Other languages are developed by one group only and delivered with a specific set of tools. For this reason it is often difficult to judge if a criterion is applicable to a particular language and to what extent additional tools should be considered as part of the 'language'. For this reason, several notes have been added with the comparison.

The languages chosen in this comparison are chosen because they were in use during the most recent Census shown in [6] and are known to be in current use. This Australian/New Zealand source is a comprehensive survey of languages currently used in introductory programming. The inclusion of other languages could also be argued.

In this comparison all criteria are equally weighted, but the ordering presented here could easily be changed if criteria weightings were applied. All features are hardly equally important to all educators in all education situations.

Compared to the afore-mentioned feature lists given by Milbrandt [20] and McIver and Conway [15], the enumeration presented in this paper is more extensive. Features related to learning, design and environments have been considered previously, but including criteria concerning support, availability and possibilities beyond introductory programming seem to be unique to this study.

Some of the languages compared here can be regarded as "non-traditional" to introductory programming and might be avoided by some educators. Lack of strict typing in some languages (e.g. Python and JavaScript) is of concern to some educators. Some argue that teaching a language that is removed from a full industry relevant language can disadvantage students. Introducing programming using a simple language may cause students to run into a wall when having to deal with a more complex one later on. However, Mannila *et al* suggest students are not disadvantaged by having learned to program in a simple language when moving on to a more complex one [13].

For many instructors the choice of paradigm is primary and the language used must fall into a paradigm. There is as much literature discussing the value of teaching within one paradigm or another as literature discussing language choice (for example [7, 8, 9, 10, 12, 19, 24]). Certainly, if such an approach is necessary, then the results presented here must be qualified according to the instructor's choice of paradigm.

This given, a comparison has been attempted by the authors and the results are shown in Table 2. By this comparison, three languages are arguably the most suitable languages of those compared. Python and Eiffel rate highest which justifies their design as teaching languages. These are closely followed by Java, which is commonly associated with industrial applications. Other evaluated languages rated lower.

The comparison given is limited by the authors' experience with each language. The authors encourage all readers to consider how they would rate these languages, or perhaps others, according to the criteria.

4. CONCLUSIONS AND RECOMMENDATIONS

Not surprisingly, the authors' comparison suggests that the most suitable languages for teaching, Python and Eiffel, are languages that have been designed with teaching in mind. However this study also showed that Java, which is designed primarily for commercial application, has merit when considered as a teaching language.

By providing well founded criteria, this study has attempted to

Table 2: Languages Compared by Features

	C	C++	Eiffel	Haskell	Java	JavaScript	Logo	Pascal	Python	Scheme	VB
Learning											
Is suitable for teaching (§2.1.1)			✓				✓	✓	✓		
Can be used to apply physical analogies (§2.1.2)			✓		✓	✓	✓		✓		✓
Offers a general framework (§2.1.3)	✓	✓	✓		✓	✓		✓	✓		✓
Promotes a design driven approach for teaching software (§2.1.4)			✓	✓	*1		✓			✓	
Design and Environment											
Is interactive and facilitates rapid code development (§2.2.1)				✓			✓		✓	✓	
Promotes writing correct programs (§2.2.2)		*2	✓		*2				*2		
Allows problems to be solved in "bite-sized chunks" (§2.2.3)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Provides a seamless development environment (§2.2.4)			✓		*1						
Support and Availability											
Has a supportive user community (§2.3.1)	✓	✓	✓	✓	✓	✓			✓	✓	✓
Is open source, so anyone can contribute to its development (§2.3.2)									✓		
Is consistently supported across environments (§2.3.3)	✓	✓	✓		✓	✓	✓	✓	✓	✓	
Is freely and easily available (§2.3.4)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Is supported with good teaching material (§2.3.5)		✓	✓		✓		✓	✓	✓	✓	✓
Beyond Introductory Programming											
Is not only used in education (§2.4.1)	✓	✓	✓		✓	✓			✓		✓
Is extensible (§2.4.2)	✓	✓	✓		✓				✓		✓
Is reliable and efficient (§2.4.3)	✓	✓	✓		✓	✓		✓	✓		✓
Is not an example of the QWERTY phenomena (§2.4.4)		✓	✓	✓	✓	✓	✓		✓	✓	✓
Authors' Score	8	11	15	6	14	9	9	7	15	8	9

*1 Possibly with some IDEs, e.g. BlueJ (http://www.bluej.org)
*2 Possibly with unit testing

provide objectivity into what has been, up to now, frequently an emotive argument. The value of this work is to provide the potential for a strong argument to those who seek to promote a language change in an introductory course or perhaps over an entire undergraduate degree program.

This work may also be useful to communities of developers attempting to produce better programming languages for future novices and experts.

Extensions of this study in future work may attempt to clarify the criteria presented here, perhaps extending the criteria for specific purposes. Other languages may also be compared using this framework and it may be shown that other languages are useful for introductory languages according to these or other criteria.

5. REFERENCES

[1] Andreae, P., Biddle, R., Dobbie, G., Gale, A., Miller, L., and Tempero, E., *Surprises in Teaching CS1 with Java (School of Mathematical and Computing Sciences, Technical Report CS-TR-98/9)*. 1998, Victoria University of Wellington: Wellington.

[2] Bergin, J. *Java-- GOOD, BAD, and NOT C++*. 2000 [cited 30th August, 2006]; Available from: http://csis.pace.edu/~bergin/Java/SomegoodthingsaboutJava.html.

[3] Biddle, R. and Tempero, E., *Java pitfalls for beginners.* ACM SIGCSE Bulletin, **30**(2), 1998, 48 - 52.

[4] Chandra, S.S. and Chandra, K., *A comparison of Java and C#.* Journal of Computing Sciences in Colleges, **20**(3), 2005, 238 - 254.

[5] de Raadt, M., Watson, R., and Toleman, M. Language Trends in Introductory Programming Courses. In *The Proceedings of Informing Science and IT Education Conference*. (Cork, Ireland, June 19-21, 2002). InformingScience.org, 2002, 329 - 337.

[6] de Raadt, M., Watson, R., and Toleman, M., *Introductory programming languages at Australian universities at the beginning of the twenty first century.* Journal of Research and Practice in Information Technology, **35**(3), 2003, 163-167.

[7] Decker, R. and Hirshfield, S. A case for, and an instance of, objects in CS1. In *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*. (Vancouver, B.C. Canada, October 18 - 22, 1992), 1992, 309-312.

[8] Decker, R. and Hirshfield, S. The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS1. In *Selected papers of the twenty-fifth annual SIGCSE symposium on Computer science education*. (Phoenix, Arkansas, United States, March 10 - 12, 1994). ACM Press, New York, NY, USA, 1994, 51 - 55.

[9] Hadjerrouit, S., *Java as first programming language: a critical evaluation.* ACM SIGCSE Bulletin, **30**(2), 1998, 43 - 47.

[10] Hadjerrouit, S. A constructivist approach to object-oriented design and programming. In *Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*. (Cracow, Poland, June 27 - July 1, 1999), 1999, 171 - 174.

[11] Hitz, M. and Hudec, M. Modula-2 versus C++ as a first programming language--some empirical results. In *Papers of the 26th SISCSE technical symposium on Computer science education*. (Nashville, TN USA, March 2 - 4, 1995). ACM Press, New York, NY, USA, 1995, 317-321.

[12] Kölling, M., Koch, B., and Rosenberg, J. Requirements for a first year object-oriented teaching language. In *Papers of the 26th SISCSE technical symposium on Computer science education*. (Nashville, TN USA, March 2 - 4, 1995). ACM Press, New York, NY, USA, 1995, 173-177.

[13] Mannila, L., Peltomäki, M., and Salakoski, T., *What About a Simple Language? Analyzing the Difficulties in Learning to Program.* Computer Science Education, **16**(3), 2006, 211 - 228.

[14] Mayer, R.E., Dyck, J.L., and Vilberg, W., *Learning to Program and Learning to Think: What's the Connection?* Communications of the ACM, **29**(7), 1986, 605-610.

[15] McIver, L. and Conway, D. Seven Deadly Sins of Introductory Programming Language Design. In *Proceedings of the 1996 international Conference on Software Engineering: Education and Practice (SE:EP '96)*. (Dunedin, New Zealand, January 24 - 27, 1996). IEEE Computer Society, 1996, 309 - 316.

[16] Meyer, B. Towards an Object-Oriented Curriculum. In *Proceedings of 11th international TOOLS conference*. (Santa Barbara, United States, August 1993). Prentice Hall 1993, 1993, 585 - 594.

[17] Papert, S., *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., USA, 1980.

[18] Parker, K.R., Chao, J.T., Ottaway, T.A., and J.Chang, *A Formal Language Selection Process for Introductory Programming Courses.* Journal of Information Technology Education, **5**, 2006, 133 - 151.

[19] Ramalingam, V. and Wiedenbeck, S. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Papers presented at the seventh workshop on Empirical studies of programmers*. (October 24 - 26, 1997, Alexandria, VA USA). ACM Press, New York, NY, USA, 1997, 124-139.

[20] Stephenson, C. *A report on high school computer science education in five U.S. states.* 2000 [cited 31st August, 2006]; Available from: www.holtsoft.com/chris/HSSurveyArt.pdf.

[21] Stroustrup, B., *Learning Standard C++ as a New Language.* The C/C++ Users Journal, **May**, 1999.

[22] Tharp, A.L. Selecting the "right" programming language. In *Proceedings of the thirteenth SIGCSE technical symposium on Computer science education*. (Indianapolis, Indiana, United States). ACM Press, 1982, 151 - 155.

[23] van Rossum, G. *"Computer Programming for Everybody." Proposal to the Corporation for National Research Initiatives*. 1999 [cited 25th October, 2006]; Available from: http://www.python.org/doc/essays/cp4e.html.

[24] Wallingford, E. Toward a first course based on object-oriented patterns. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*. (Philadelphia, PA USA, February 15 - 17, 1996). ACM Press, New York, NY, USA, 1996, 27-31.

[25] Wirth, N., *The programming language Pascal.* Acta Informatica, **1**, 1971, 35 - 63.