A Framework for Teaching Introductory Software Development

ZAIGHAM MAHMOOD
School of Computing, University of Derby
UNITED KINGDOM
z.mahmood@derby.ac.uk

Abstract:- Software development (SD) refers to design and development of software applications. Most educational institutions teach programming using a procedural paradigm and an imperative language where the emphasis is often on learning a computer language and not on problem solving or the modeling of realistic computational problems. Thus, the teaching is dependent on the chosen language, which is not entirely appropriate for teaching principles of programming or SD as an engineering activity. This paper discusses the traditional method of teaching programming and suggests an objects-first approach where students adopt a top-down method of learning to develop software. Our model introduces functions and modules as basic building blocks for producing software. Thus, students' first programs are written as sequences, selections and iterations of given functions and it is in the later stages of the course, that they learn the basic constructs of the language. This paper outlines a complete framework for teaching a first course in programming. It also discusses the characteristics of a good teaching language to help academics to choose an appropriate first programming language.

Key-Words:- Software engineering, Software development, Programming, Computer languages, Teaching

1. Introduction

Software Development (SD) refers to design and development of software applications. It includes analysis of user requirements, production of requirements document, design and construction of applications and software testing using test plans to ensure that the final products conform to clients' requirements [1, 2]. It may also involve formal specifications and formal methods if the application is large and complex and where the user requirements are to be precisely written. University courses in Software Engineering (SE) and other programming-based subjects normally start with programming-in-the-small where students begin to learn a simple design method, basics of a computer language and methods of testing. The teaching team provides the program specifications, so there is normally not much emphasis on requirements elicitation or systems analysis. Since the programs are usually small, students do not necessarily understand the importance and significance of proper testing.

A primary objective of such courses is to enable its students to gain the necessary knowledge of a computer language and acquire some experience of developing small programs. Whereas, teaching of the required theory is generally not too difficult, ensuring that students gain the necessary experience is far from easy. If the course is well designed, properly delivered and students show the required commitment and work as they should – as the software development is a technical subject and requires hard work, then the task becomes somewhat easier. However, being proficient in a skill is a function of time and the time available on a three or four-year degree or diploma programme is simply not enough. Thus, it is essential that the course team designs the practical elements of an SE and other programming-based programmes of study with great care and ensure an effective delivery.

Programming-based courses at educational institutions normally start with an *introductory programming* or an *introductory software development* module, right at the beginning of the course, normally in the first semester of the first year. This paper focuses on the teaching of such a programming module.

In this paper, we first discuss the traditional procedural paradigm and then highlighting the weaknesses of this method, we outline an objects-based approach to teaching software development. Finally, we discuss the characteristics and issues concerning the choice of first programming languages and present our conclusions.

In the following sections, the terms ‘programming’ and ‘software development’ and the abbreviation ‘SD’ will be used interchangeably.

2. Software Development Paradigms

There are several SD paradigms including imperative, object-oriented, functional and logic all of which are taught on a computing programme at different levels. For the teaching of programming in the first year, however, it is the procedural or imperative model that is most popular, even when the language used is an object-oriented or object-based. Advocates for this approach suggest that:

- This approach is highly compatible with von Neumann architecture of today’s computers.
- The fourth generation languages are highly suitable for implementing procedural programming.
- Students find it more natural to break down a programming project as a set of *procedures* rather than *objects*.

Although, numerous object-oriented, object-based and scripting languages have been developed over the years including the scripting languages, and institutions are beginning to adopt newer approaches to teaching programming including even the functional and logic programming methods [3], a majority of them are still teaching *console applications* as opposed to *windows applications*, at least at the beginning of a programming module.

2.1 Procedural Programming

Although, the procedural approach may be considered as suitable for programming-in-the-small, it is not entirely appropriate for software engineering. It is a bottom-up approach highly dependent on the chosen language. The emphasis is on learning a computer language and not on problem solving or the modeling of realistic computational problems. In most cases, teaching of the language follows a scheme that requires the teaching of the following normally in the order as presented here:

- General program structure
- Declarations and Variables
- Input/output and Assignments
- Iterations and Selections
- Arrays and Records
- Functions, Procedures and Parameters
- Other features of the language.

In addition to learning the syntax of a computer language, students also need to master the workings of the language environment, often called *integrated development environment* (IDE). To successfully execute even a very small program, students need to have a reasonable knowledge of the following:

- Use of the IDE
- Use of the editor (as part of the IDE) to input and edit source code
- Use of the compiler system to compile, link, build and execute their programs.

Furthermore, they need to have an understanding of the system and other diagnostics including syntax and runtime errors, all of which adds another layer of difficulty to the learning process. If a formal design method is also taught at the same time then the teaching and learning becomes even more difficult. However, as Dehnadi and Bornat [4] put it: *Learning to program is notoriously difficult*.

To summarize, in order to satisfactorily complete even a trivial programming exercise, students need to have the sufficient knowledge of all the following:

- Basics of a design method
- Syntax of necessary declarative and other functional statements of a language to convert design into program code
- Procedures for entering, editing, compiling, linking and executing the program
- All other procedures such as starting the IDE, getting into the language system and closing the project.

Design is often taught using function-oriented methods. Teaching *functional decomposition* or *stepwise refinement* is relatively easy, however, teaching a well established formalized method requires students to master an additional skill at the same time as learning a language and its environment. Additionally, if the module requires students to learn a formal design method then learning software development becomes a two stage process: leaning the design first and learning the program development later [11]. In this case, the time set aside for SD generally reduces to half, which is highly unsatisfactory.

Thus, there is much to do in the first few weeks of a programme of study. Although, it is essential that all this is well understood, learning it all at the beginning of the course can be so daunting that there is a real danger that novice programmers may loose their confidence and get so disappointed with the experience that some may even give up never to get to the next stage or transfer to another programme of study.

Not withstanding the difficulties mentioned above, a careful observation reveals that the process, that is usually followed, does not teach construction of software as an engineering activity. Although, applications are being designed and developed, the emphasis is not on building software based on accepted engineering principles (modularity, reuse, information hiding, functional independence, etc), which students will need to employ when they engage in the development of complex software. Also, we find that the course team is so busy teaching the essentials that the approach does not leave much time for teaching or learning documentation, quality, professionalism and elements of good practice.

An *objects-first* approach, as described in the following sections, will resolve some of the issues as mentioned above.

2.2 An Objects-First Approach

An *objects-first* approach to teaching a first course in programming suggests that:

- Modules and functions (also called *methods*) are regarded as basic building blocks
- Software applications consist of interacting modules
- New modules are built using existing ones, whenever possible.

This forces a structured approach to *modular programming* where use of modules and functions (also called *methods*) establishes the principles of code *reuse* and *functional independence*. Clearly, the emphasis is on modularization, encapsulation, recursion and reuse. Programming thus becomes an activity based on engineering principles. This contrasts sharply from the traditional procedural approach where modularity, functions and recursion do form part of the curriculum but attract much less emphasis as they are taught much later in the module.

3. Software Development Using An Objects-First Approach

To implement an engineering approach to software development, we propose an Objects-First Approach to Procedural Paradigm. Our model is loosely based on the work of Richard Bornat [5] and regards the construction of software as an engineering activity where modules and functions are the fundamental building blocks. The method helps to produce properly structured and good quality modular software. It is a top-down approach where the important concepts of object technology and principles of engineering are introduced right at the beginning: in the first semester of the first year of the programme of study. Although an engineering activity also has an emphasis on teamwork [13, 14], which SE necessarily requires, it is recognized that in an introductory SD course, there may not be enough time for students to work in groups.

Our method requires the establishment of a library of functions on a suitable topic (e.g. graphics) prior to the delivery of the programming module. When students begin to learn SD, their programs will be initially written as sequences of given functions where students will regard these as black boxes and consider only the external behavior of these functions. Building programs in terms of functions will help students to understand modularization, reuse and encapsulation mechanisms, without knowing the intricacies of the computer language being used.

It is important that students are able to successfully execute their programs early on in the course. This provides a sense of achievement and increases students' confidence. After successfully running a number of simple programs and understanding the basics of the language environment, programs are written as selections and repetitions of the same functions. It is at this point that students learn the general syntax of selection and iteration statements of the language. Now, the importance of modularity and code reuse can be re-emphasized and mechanisms for reducing software complexity, incremental development, polymorphism and overloading of functions explained which can be practiced in later sessions when students produce their own libraries of functions.

Students who come this far are now ready to learn the syntax of input, output, assignment and other basic statements of the language for producing more realistic programs. This is the time to practice functional independence, quality, code readability, maintainability and other elements of good programming style.

Now that students have the necessary practice and knowledge of the language, they can begin to 'problem solve', design and build their own programs from given specifications.

In an introductory programming module, the focus should be on problem description and problem solving strategies. We suggest that the design technique be a simple one so that students do not feel that they are learning an additional method - *Stepwise Refinement* [7] is a perfectly acceptable approach. Although, students will learn by producing their own programs, use of good quality, well structured and properly documented worked examples will greatly advance the learning process – all taking place in a computer laboratory so that teaching and practice can be usefully combined.

3.1 Teaching Strategy

We now outline a teaching plan based on the above model [19]. We assume the module duration to be twelve weeks and suggest four hours per week of contact time including lectures, tutorials and practical sessions.

Before Week 1:
The teaching team is required to create a library of functions on an appropriate topic e.g. graphics. This requires a certain amount of investment of time and effort but there is no need to create an extensive library. As an example, two such methods might be LINE and CIRCLE:

- LINE - to draw a straight line of a given length in a given direction starting at a given point. This will require 4 parameters to be supplied: length, direction, x-coordinate and y-coordinate
- CIRCLE - to draw a circle of a given radius at a given point. This will require three parameters: radius, x-coordinate and y-coordinate

Initially, just the two methods may be sufficient to produce simple programs. In the following sections, the terms ‘methods’, ‘functions’, ‘routine’ and ‘modules’ will be used interchangeably.

Weeks 1-2:
Today’s languages work within an *Integrated Development Environment* (IDE), which adds a certain amount of additional learning. So, introduce the IDE and get students to familiarize with the toolkit, especially the editor within the IDE. So:

- Give students a simple working program with a complete set of instructions.
- Ask them to follow the instructions to enter and execute the program to understand the compiling and execution process.
- Introduce the general program structure in the chosen language.
- Give students more working programs, which they enter and execute.

It is important that examples given to students are well constructed and properly commented to illustrate good programming practice and style. It is important that students understand the essentials of IDE well.

Week 3:
Teaching of programming concepts – e.g. *sequences* of actions - can now begin:

- Introduce the concept of functions (also called modules and *methods*) as components which, when put together in the right manner, result in the required software system.
- Explain the purpose and use of the library methods introduced e.g. LINE and CIRCLE.
- Explain the significance and meaning of required parameters and use of argument (i.e. parameter) lists.

- Provide examples of the use of library methods such as LINE and CIRCLE.
- Ask students to write simple programs calling (i.e invoking) the given methods in a sequence.
- Give students more examples of well-designed similar programs.

In the first exercise, they can be asked to draw a chair (consisting on a number of straight lines) or a table (consisting of a circle and a number of straight lines). Refer to Figures 1 and 2.

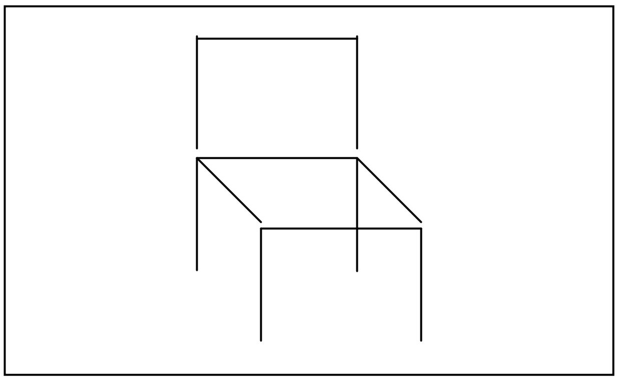
Figure 1: Sequence of LINE method

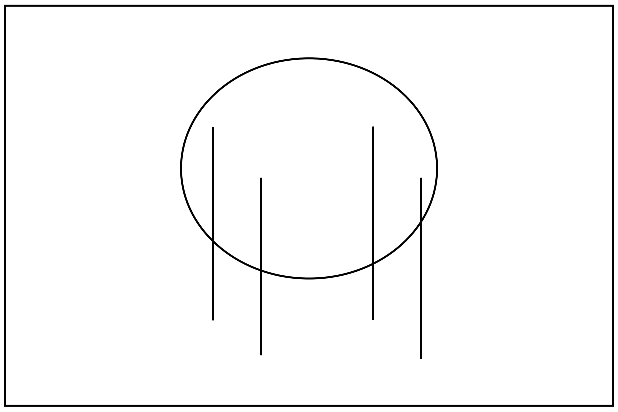
Figure 2: CIRCLE and Sequence of LINE method

Week 4:
Now that students understand the sequences of actions (or programming statements), *iterations* (i.e. *repetitions*) can be introduced. At the same time, students can learn the construction of new methods as sequences of existing methods:

- Provide examples of new methods e.g. a method called CHAIR, which consists of sequences of LINE method.
- Ask students to construct new similar methods and invoke them to draw a row of chairs (Figure 3).
- Ask students to construct a new method called, say, TABLE and write programs to draw a row of tables.

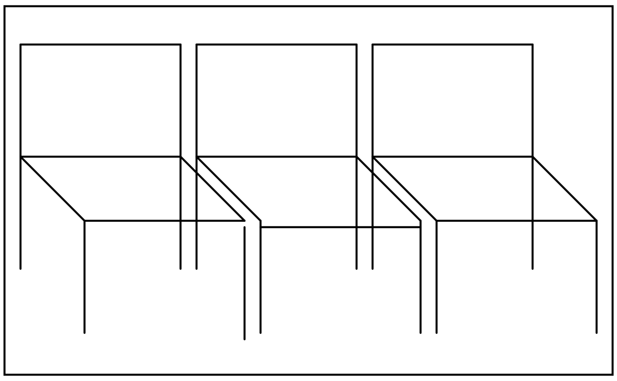
Figure 3: Sequence of CHAIR method

Weeks 5-6:
Now that students have written a few programs, it is time to introduce program design as well as further explanation of the programming constructs and concepts they have studied so far. Also explain the *selection* mechanism:

- Explain what ‘problem solving’ is.
- Explain a simple design method e.g. pseudo-code or Stepwise Refinement [7]
- Provide simple program specifications and ask students to produce pseudo-code based on Stepwise Refinement.
- Explain the selection (and iteration) statements of the language being used.
- Ask students to design (by writing pseudo-code) and write programs using newly learnt constructs e.g. a program that draws, depending on an option, either a row of chairs or a row of tables.
- As the next example, ask students to design and develop a program that draws, depending on an option, either a row of chairs or a table surrounded by a number of chairs. Refer to Figure 4.
- Give students more examples of pseudo code using Stepwise Refinement.

At the end of week 6, it is hoped that students have learnt the following:

- The basics of an IDE.
- The basic programming constructs: sequences, repetitions and selections.
- The syntax of basic language statements.
- Concepts of methods and parameters.
- A simple design method.

If this is the case, the initial difficulty of learning programming has been overcome.

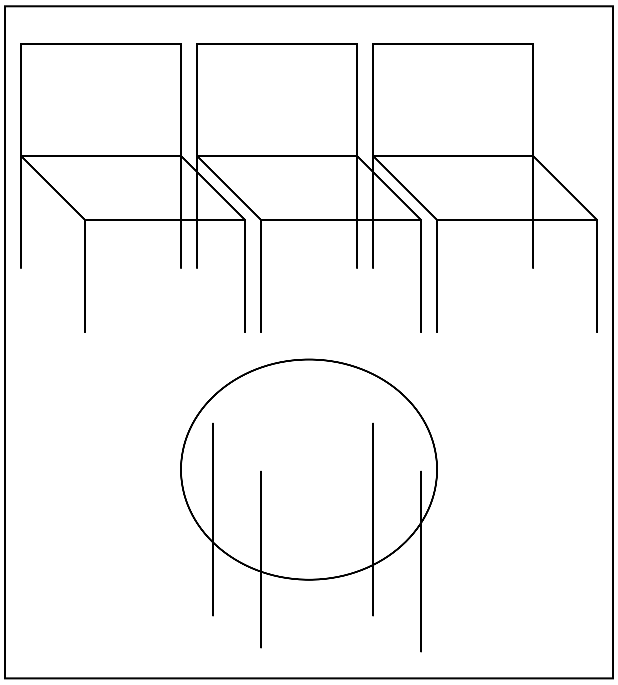

Figure 4: CIRCLE and Sequence of CHAIR method

Weeks 7-8:
From now on, the remaining teaching will become easier and now other aspects of SD such as documentation, programming style, quality, program testing, etc, can be introduced and practiced.

- Explain the significance and importance of good programming style: use of comments, indentation, blank lines etc.
- Explain modularity, code reuse, functional independence and other engineering principles – emphasizing that SD is an engineering activity.
- Explain the need to design first even for trivial programs.
- Explain the advantages of incremental development (often referred to as *growing* software) i.e. building the main functionality first and then adding the new features [14].
- Explain the benefits of producing proper designs and test plans.
- Explain how to produce test plans and emphasize importance in relation to proper testing of software.
- Discuss the benefits of using appropriate standards, developing quality software, keeping accurate records and producing proper documentation.
- Explain, now, that modules students used in earlier programs (e.g. LINE, CIRCLE, CHAIR, TABLE) represented objects and classes and that they were using these to create new objects (chairs and tables).
- Explain basic concepts of object technology noting that the purpose is not to teach object-oriented approach but merely to introduce the essential terminology.

The above will provide a welcome break from technicalities of design and development of programs. Certain basic language statements can now be explained and exemplified to develop non-graphical programs e.g. programs, that manipulate numbers and characters:

- Teach the syntax and use of input, output and assignment statements of the language.
- Explain types of variables and numbers available in the language.
- Give students examples of programs using these statements e.g. a program to read a series of numbers and calculate the average value.

Week 9:
Students are now, hopefully, reasonably confidant to design and write their own functions and objects. Next set of exercises can require them to design and produce methods called, say:

- ANOTHERCHAIR to draw a chair of a different style.
- ANOTHERTABLE to draw a different style of table.
- CHAIRSANDTABLES to draw a number of chairs and tables arranged differently and using existing methods such as CHAIR, TABLE, ANOTHERCHAIR, ANOTHERTABLE, etc.
- AVERAGE to read a series of numbers and output their average.

- Give students examples of well-written programs.

It is important that designs produced at this point are correct and the programs that students write exhibit good programming style.

Weeks 10-12:
Students can now explore further. Windows-oriented programming, using forms and controls, can now be explained and examples provided – especially if using a language such as VBasic, C# and Java [8-10]:

- Ask students to produce more complicated programs and practice what they have learnt.
- Ask students to produce programs to handle numbers and character variables.
- Provide more examples.
- Explain the concept of arrays and records and ask students to develop programs using these.
- Explain advanced features of the language and other engineering concepts.

Make students aware of the difference between programming-in-the-small and programming-in-the-large. Re-emphasize the advantages of incremental development and benefits of producing proper designs and test plans.

3.2 Summary

The strategy for teaching introductory programming presented in this paper suggests that teaching of a language should follow the sequence, as given below:

- Language IDE
- Program structure and program layout
- Use of libraries and sequences of statements
- Functions, methods and parameters
- Selection and repetition statements
- Input, output and assignment statements
- Data structures: arrays and records
- Advanced features of the language.

In the current scheme, the teaching of other elements of program development will follow the sequence, as given below:

- Development of a set of methods for students to use e.g. LINE and CIRCLE.
- Development of example programs for students to learn by examples.
- Problem solving.
- Design method e.g. stepwise refinement.
- Engineering principles such as modularity, reuse, functional independence, etc.
- Incremental development.
- Test planning and program testing.
- Quality.
- Documentation.

4. Programming Languages

The primary objective of an introductory programming module should be to teach the principles of programming. In this respect the choice of a language becomes irrelevant [6]. However, the teaching team needs a language to illustrate the principles and provide practice of SD. Choice of the language, then, depends on the programming paradigm employed. Since, procedural programming is the most favored approach, first languages tend to be mainly procedural. However, object-oriented, object-based and visual languages (e.g. C++ and Visual Basic) can also be used for procedural programming (i.e. for *console applications*). Some institutions are using declarative languages where the teaching is based on logic and functional programming paradigms [3].

It is often suggested that a first language should be well structured, available (in the sense of staff expertise) and easy to teach, learn and use [6]. Whereas, this may be acceptable for programming-in-the-small, when teaching principles of engineering and elements of good practice with a view to producing complex software, the criteria is not sufficient. Since choice of a language depends, also, on the programming and design methods used, the above criteria need to be extended. We suggest that a first language should possess at least the following characteristics:

- Small and simple but powerful.
- Strongly typed and block structured.
- Procedural but offering extensions to implement object technology.
- Features allowing implementation of engineering principles and concepts.
- Industrially relevant.

Simplicity and smallness imply ease of use as well as ease of learning and debugging. It is important that students can produce and execute simple programs quickly. Power of a language is its ability to deal with complex problems as well as simple ones. Strong typing reduces debugging problems and block structuring helps to produce structured and modular software. Features to implement object technology and engineering principles are essential when implementing an object-oriented approach or a model similar to the one suggested in this paper. Industrial relevance is important for the reasons of students' employability after the completion of their studies.

TIOBE [16] publish a programming community index each month that indicates the popularity of programming languages in the industry. According to their latest report in July 2009, Java and C have been the most popular languages since 2002, followed by PHP and VB as the next two favorite languages in the last two years (2008 and 2009).

Fischer [3] suggests the following criteria for the choice of first programming languages:

- Powerful enough to demonstrate the programming concepts.
- Easy to learn.
- Not error-prone i.e. get running fast.
- Easy to use development tools.
- Well supported by ways of availability of library functions.

Britton [12] suggests the following technical characteristics for a good programming language, not just for industry but for academia as well:

- Ease of learning.
- Ease of understanding.
- Speed of development.
- Help with enforcement of correct code.
- Performance of compiled code.
- Supported platform environments.
- Portability.
- Fit-for-purpose.

Ease of learning and understanding requires the language to be as close to a natural language as possible so that the written code can be easily understood for maintenance purposes. In this respect, Java is much easier to learn than C++. These two characteristics will also help with the enforcement of correct code. Speed of development and performance refer to agility and efficiency. Platform support and portability are useful feature when programming is conducted on different platforms using different operating systems. This feature is also important in today's age of open source projects.

Joseph [15] conducted a survey in 2008, of 17 UK universities, chosen at random, to determine what languages were being taught and the reason for the choice of first languages at these institutions. Java was the most popular language, VB.Net was the 2nd most popular language followed by PHP and C. Four key factors, generally considered by the 17 universities, when deciding on a first language to teach introductory programming important, were:

- Ease of learning.
- Easy to teach.
- Academic soundness – teach good programming principles.
- Relevance to industry – for students' employability.

Currently available languages such as Visual Basic, Java, and C# [8-10] are all highly suitable first languages for an introductory course in programming or SD:

- VB.Net is a good tool for building windows and web applications. Its key features include simplified development, power and flexibility, platform independence and interoperability.
- Java is an object-oriented structured general-purpose language. It is simple, robust, high performance and interpreted language.
- C# is Microsoft's answer to Java and is very similar to Java. It is a general-purpose object oriented language, which is simple, secure and excellent for screen handling and developing applications rapidly.

C is another popular language that is highly suitable for teaching introductory SD. It is a general-purpose language widely used on many different software platforms for building portable applications software [17].

According to TIOBE [16], Java and C have been consistently the most popular languages in the industry since 2002, followed by PHP [18] and VB [8] as the next two most popular languages for software development in the last two years (2008 and 2009).

As mentioned by Joseph [15], PHP [18] is also becoming popular in academic circles. It is simple and easy to learn but a powerful general-purpose scripting language that includes an easy to use command line interface. It supports object-orientation and is platform independent.

5. Conclusion

Procedural paradigm is the traditional and most favored approach for teaching a first course in software development. It is a bottom-up and syntax driven approach, which is highly dependent on an imperative language. Students learn not only the syntax of a language but its environment as well - as the current languages are generally available for use through an IDE. If a formal design method is also taught at the same time as the language then students can get so overwhelmed by the amount of learning that some may loose their confidence and get disappointed with the learning experience. Also, the traditional approach teaches programming in the sense of producing code and does not teach SD as an engineering activity.

To resolve the inherent issues in the traditional approach to teaching programming, this paper suggests an objects-first approach to procedural paradigm. This is a top-down approach, which regards functions and modules as the fundamental building elements for the construction of software. The emphasis is on modularity, code reuse, practice of engineering principles as well as quality, standards and professionalism right from the start. A teaching scheme is also presented which can be used as a basis to construct a first course in teaching software development.

A discussion on the characteristics of a good teaching language as well as a brief introduction to most popular languages, have also been presented to help decide a computer language for teaching introductory programming. Java, C, VB.NET and PHP are highly suitable languages for teaching an introductory course in software development.

References

[1] Sommerville I, “Software Engineering”, 8th Edition, Addison Wesley, 2006
[2] Bell D, “Software Engineering for students”, 4th Edition., Addison Wesley, 2005
[3] Fischer P, “Teaching programming to beginners”, IMM, DTU, Available at: www2.imm.dtu.dk/~tb/ fischer.pdf
[4] Dehnadi S and Bornat R, 2006, “The camel has two humps”, Available at: www.cs.mdx.ac.uk/research/PhDArea/ saeed/paper1.pdf
[5] Bornat R. “Programming from first principles”, Prentice Hall.
[6] Busschots B, “Teaching programming – why the choice of first language is irrelevant”, Bloggs dated 8 Jan 2008, Available at: http://www.bartbusschots.ie/ blog/?p=634
[7] Wirth N, “Program development by stepwise refinement”, Comm. ACM, Vol 14, No 4, pp 221-227, 1971
[8] Balena F, “Programming microsoft Visual Basic”, 2nd Ed, Microsoft Press, 2006
[9] Murach J, “C# 2008”, by dissection, M Murach & Associates, March 2008
[10] Deitel and Deitel, “Java: How to program”, 7th ed, Prentice Hall, 2007.
[11] Kambhampaty S, et al, “ Architecting for Next Generation Business Applications”, WSEAS Trans. on Business and Economics, Issue 4, Vol. 3, April 2006
[12] Britton C, “Choosing a Programming Language”, January 2008, Available at: http://msdn.microsoft.com/en-us/library/ cc168615.aspx
[13] Kumlander D, “On Using Software Engineering Projects as an Additional personal Motivating Factor”, WSEAS Trans. on Business and Economics, Issue 4, Vol. 3, April 2006
[14] Kumlander D, “Supporting Software Engineering”, WSEAS Trans. on Business and Economics, Issue 4, Vol. 3, April 2006
[15] Joseph M, “Choice of Language for Teaching First Year Students”, BSc final year project, School of Computing, University of Derby, UK, Oct 2008

[16] TIOBE Software, “TIOBE Programming Community Index for July 2009”, July 2009, Available at: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[17] Harbison H P, “C: A Reference Manual”, 5th Ed, Prentice Hall, 2002
[18] Converse T, “PHP Bible”, 2nd Ed, John Wiley, Sept 2002
[19] Mahmood Z, “An Objects-First Approach to Teaching Introductory Software Development”, Proc. 6th WSEAS Int. Conf on Engineering Education, Rhodes, Greece, July 2009.