

Multi-Paradigm Approach for Teaching Programming

Laxmi P Gewali* and John T Minor
School of Computer Science
University of Nevada, Las Vegas
4505 Maryland Parkway, Las Vegas Nevada 89154

Abstract: Selecting an appropriate programming paradigm in which to teach the first programming and problem solving course in a Computer Science undergraduate program has been discussed extensively. Procedural programming, functional programming, and object oriented programming are the most widely-used programming paradigms both in institutes of higher learning and in the high-tech industry. In recent years, a number of authors have suggested object oriented programming as the first programming paradigm for undergraduate curriculum. We argue the critical role of using more than one programming paradigm in the typical undergraduate program in Computer Science and Information Technology. We address the issue of integrating multi-paradigm programming in undergraduate curricula. We identify core computer science courses where appropriate programming paradigms could be embedded. We then propose an outline of multi-paradigm programming topics and their mapping to appropriate core courses in an undergraduate computer science curriculum.

Keywords: First programming language, programming paradigms, undergraduate curriculum.

1. Introduction to Programming Paradigms

A number of different programming paradigms have been developed over the years. Most of the early programming languages (machine, assembly, and high level) were based on the imperative paradigm. A computer program written using the imperative programming paradigm consists of a sequence of commands/statements which operate on the data stored in memory. One of the most remarkable aspects of imperative programming is the mechanism of *side effect* done by the assignment statement. The assignment statement changes the state of the program by altering the content of memory locations. The development of imperative programming is heavily influenced by the von Neumann model of stored programs. Because of the continued use of the imperative programming paradigm from the advent of early computers to the present time, its role in computer science education is vital. While large-sized programs written using imperative constructs with side-effects are difficult to comprehend, still we should be aware that this is the programming paradigm introduced in most first courses in programming and problem solving.

* Corresponding author

Almost all high level programming languages of the present time, including C/C++, C#, Java, Perl, and Python, provide a complete set of statements and command constructs to facilitate imperative. Over the years, a couple of organizational paradigms have been added to the imperative model in order to make program construction easier. The first of these additions is the procedural paradigm, in which programs are divided into a set of executable code blocks called procedures. This form of abstraction allows the programmer to use a divide-and-conquer approach to the design of the control flow of a program.

A newer organizational abstraction that has been introduced to imperative programming is the object oriented paradigm. In object oriented programming, data and operations are wrapped in entities called objects. Essentially, an object consists of data members and method members, and methods operate on objects to achieve a computation. In formal object oriented terminology, objects receive message requests from methods (functions) to perform a computation. While objects are viewed as servers, the code fragment(s) requesting the application of methods are the clients. This is why object oriented programming is also referred to as the programming approach based on the communication between clients and servers. In fact, the terms “object oriented programming” and “client server programming” are evolving as synonyms.

A paradigm based on the recursive function theory of computation, instead of the Von Neumann model, is the functional programming paradigm. The main aspect of functional programming is expression evaluation. Functional programming asserts that any computation can be expressed in terms of a sequence of expression evaluations without any side effect. One can come up with simple examples where it is very easy to write a program in an imperative setting but is very difficult and non-intuitive in a functional programming language, and vice versa.

Other programming paradigms, like logic programming and rule-based (production) systems, will not be discussed in this paper because they are not widely used in industry. They should be introduced in a “principles of programming languages” course or in elective courses like artificial intelligence or expert systems, where their use is applicable.

2. Multi-paradigm Programming

Students in Computer Science should be taught more than one programming paradigm during their first year of programming instruction. They need to see a variety of options that can be used in the design of their programs. It should be emphasized that not all problems fit easily into one and only one paradigm. That is, certain problems are easier to think of and solve in one model versus the other models, and in fact, some larger problems may require multiple models used together.

Let us consider the problem of writing a program for the implementation of a compiler or an interpreter for a high level language such as C++ or Java. The implementation may require the following:

- A procedural process should be used to form the basic organization of the compiler. It is easiest to think of the translation process as a sequence of transformations on the original source language: first the lexical analysis, second the syntactic analysis, and finally the semantic analysis and object code generation. The easiest way to implement this kind of design is with the procedural programming paradigm.

- A functional model should be used to implement a recursive descent parser. This is the easiest way to handle the syntactic analysis section of the compiler, if one decides to use a top-down construction.
- An object oriented model should be used to design the symbol table and its corresponding accessing and storage routines. The object oriented paradigm is perfect for encapsulating and isolating the data and operators together, no matter what kind of storage/retrieval mechanisms are used for the symbol table of the compiler.

Notice that there is a natural way to think about these sub-problems, and that each paradigm has its use in the design solution of the entire compiler. To force the entire compiler to be designed and implemented using only one programming model would be unnecessarily difficult, unnatural, and pedagogical unfriendly.

For these reasons, it is important that students see and be able to program in a variety of paradigms. This can be done either by teaching one programming language that incorporates multiple paradigms (like Leda [1], C++) or by teaching a set of programming languages, one for each model, which can be used together in the implementation of the total project.

3. Integration in Undergraduate Curriculum

It may not be feasible to completely include all programming paradigms, even the widely practiced ones (procedural, object oriented, and functional), as core courses in an undergraduate curriculum. It is possible to integrate enough introductory threshold materials of the three paradigms by indirectly embedding them in the core computer science courses. The most widely used core courses in undergraduate computer science that require some use of programming and that follow the ACM guidelines are: (a) Computer Science I, (b) Computer Science II, (c) Systems Programming, (d) Data Structures, (e) Principles of Programming Languages, (f) Operating Systems, and (g) Compiler Construction. The first two courses, Computer Science I and Computer Science II, are essentially the beginning courses where elements of programming and problem solving are introduced. Over the last sixty years these courses have been taught by using a number of different programming languages that include Fortran, Algol, PL/I, Scheme, Pascal, C/C++, and Java. While problem solving is an important issue in both these courses, the main ingredient is the art and science of programming. In most colleges and universities, these courses have been taught by using the procedural programming paradigm. A few universities at the present time are beginning to teach these courses by using the object oriented paradigm. Some are even spearheading a crusade to push for an “object first” approach in introductory computing courses [2,3]. Interestingly, a programming environment based on Java called Blue J [3] has been introduced with the sole purpose of emphasizing the object-first approach.

Some eminent authors are having strong reservations about the object first approach [1, 5]. We should not look at different programming paradigms as mutually exclusive. One cannot be a complete substitute for the other. Only teaching one paradigm and excluding others would be a mistake in preparing mature software engineers and computer programmers. In the algorithm analysis course, all important techniques (i.e., divide and conquer, greedy, dynamic programming, etc.) are delivered. Only emphasizing the divide-and-conquer approach in the algorithm analysis course would be considered unacceptable. Similarly, emphasizing only object oriented programming at the expense of the other paradigms would lead to an incomplete teaching job. For some problems the object oriented paradigm becomes too complicated. One

such example is the implementation of relational data bases. Similarly the object oriented paradigm is not natural for implementing table driven recognizers. On the other hand, a large variety of problems fit very well in the object oriented paradigm. One can similarly obtain example problems where the functional paradigm would be more appropriate. It is thus very important to incorporate a certain minimum exposure to all the important programming paradigms.

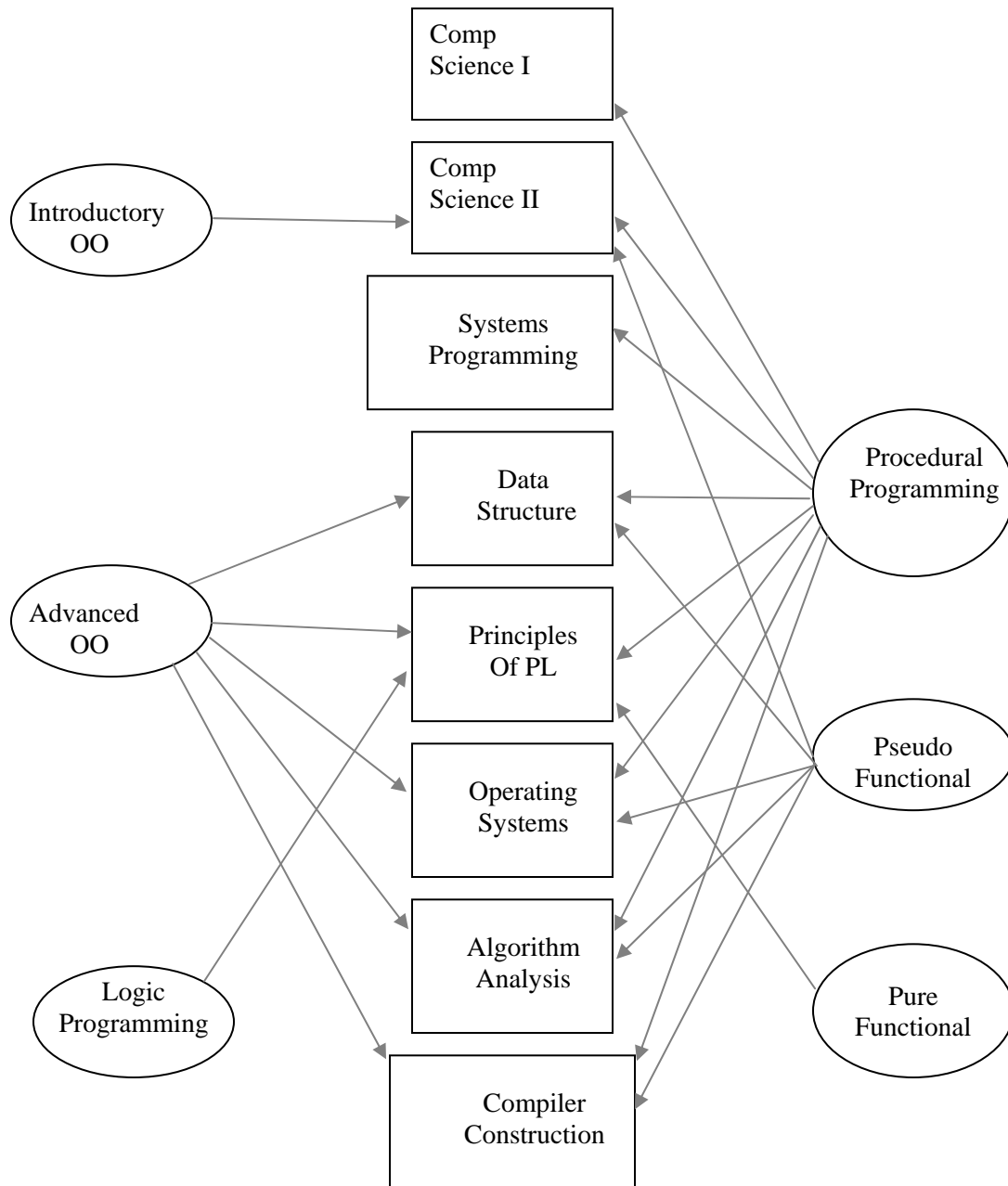


Figure 1: Mapping Paradigm Components to Courses

One critical question is where each programming paradigm should be included in a typical undergraduate curriculum. Since the art and science of imperative programming is needed in almost all courses (that need a certain amount of programming) it is obvious that the first two introductory programming courses be taught mostly by using the procedural programming paradigm. In the second programming course (Computer Science II), it would be appropriate to introduce some element of object oriented programming – maybe twenty five percent of the total course content. In order to map appropriate portions of programming paradigms to core courses, we categorize object oriented (OO) programming paradigms into two parts – *introductory OO* and *advanced OO*. Similarly, it is convenient to view functional programming as *pseudo functional programming* and *pure functional programming*. While side effects are forbidden in pure functional programming, in the pseudo functional paradigm a certain amount of side effect is allowed even though emphasis is put on expression evaluation. Pseudo functional programming would be an appropriate interface for transition from the imperative paradigm to the pure functional paradigm.

In the data structure course there are some problems where the course material can be better presented by using functional programming constructs. One example in this context is binary tree traversal. It would be pedagogically appropriate to start introducing the pseudo functional programming paradigm in the data structure course, and the students would have a better preparation when they proceed to the principles of programming languages course. The introductory OO can be introduced earlier than the functional paradigm. The right spot would be the second course on programming (Computer Science II). In the introductory OO paradigm, elementary concepts that include class, objects, data abstraction, encapsulation, information hiding, accessing privileges (private, public, protected), mutating methods, read-only methods, constructors, and UML diagrams could be covered. The advanced OO paradigm addresses topics such as single inheritance, multiple inheritance, polymorphism, and abstract interfaces. The advanced OO paradigm should be deferred to the data structure and the principles of programming languages course. Figure 1 displays the proposed mapping of programming paradigm components to computer science undergraduate core courses.

4. Discussion

We presented a brief outline of the integration of programming paradigm components in a typical computer science core curriculum. Similar mapping can be designed for elective courses such as software engineering, data base, computer graphics, computational geometry, computer networks, and computer security. The main issue emphasized here is that when students complete an undergraduate course in computer science they should have a pre-determined minimal amount of exposure to the widely used programming paradigms. It would be useful from a pedagogical perspective to prepare a list of problems which are more suitable for particular programming paradigms. Similarly, it would be very useful to have a list of interesting problems which fit almost equally well in all three paradigms.

References

- [1] Timothy A. Budd, “Multiparadigm Programming in Leda,” Addison Wesley Longman, 1994.

[2] Michael Kolling and John Rosenberg, "Guidelines for Teaching Object Orientation with Java," Proceedings of Sixth Conference on Information Technology in Computer Science, 2001, pp. 33-36.

[3] David J. Barnes and Michael Kollins, "Object First with Java – A Practical Introduction Using BlueJ," Second Edition, Prentice Hall, 2004.

[4] Chris Reade, "Elements of Functional Programming," International Computer Science Series, Addison Wesley, 1989.

[5] "Object Oriented Programming Oversold", <http://www.geocities.com/tablizer/>