Congo Sciences

Journal en ligne de l'ACASTI et du CEDESURK
ACASTI and CEDESURK Online Journal
ISSN: 2410-4299, an International Journal

How to select a programming language for teaching or for software development project.

Olamba K.P.*

Paper History
Received:
November 1st, 2017
Revised:
February 27, 2018
Accepted:
March 04, 2018
Published:
March 30, 2018

Keywords:
programming, language, criteria, selection, software, teaching

ABSTRACT

Selecting a high level programming language arises, each time, one has to start a software development project or to define a way to teach programming. The problem is complex and has been studied intensively. In this paper, we review what has been done about the programming language selection and suggest an improved programming language selection procedure. To this end, a particular attention has been paid to the different mathematical aspects of the problem. It has been shown that the problem is both ill posed, NP-complete problem, and exhibits characteristics of multi-criteria decision making problems.

*Faculté Polytechnique Département de Génie Electrique et Informatique, Université de Kinshasa B.P. 255 Kinshasa XI, R.D. Congo
* To whom correspondence should be addressed.: *paul.olamba@unikin.ac.cd*

INTRODUCTION

At the beginning of Computer Science, programs and machines were tightly coupled. So, on the first electronic calculator ENIAC (Electronic Numerical Integrator And Calculator), it was necessary to plug or unplug hundred of cables in order to move from one calculation to another. It is even why the ENIAC is not considered as the ancestor of the computer (current understanding of the word).

It is precisely by solving definitively, and in an elegant manner, the problem of moving from one program to another that the EDVAC (Electronic Discrete Variable Automatic Computer), the John Von Neumann machine, won the palm of computer ancestor. But even for the Von Neumann machine, programs were written in machine language and programming was not accessible to anyone and was reserved to only initiated professionals.

To solve that problem, assembler's languages appeared very quickly followed by high level programming languages. Nowadays, we count several hundreds of high level software programming languages and sometimes, when starting a project or when one has to teach programming, the question is which language to choose from that bunch.

An analogy to cars may help us to understand better the problem of choosing a programming language. Indeed, a tractor, a fifty tons truck, a four wheel drive car, a family city car are all vehicles and allow to move from one point to another. All these vehicles have each a steering wheel, a brake pedal, and some board instruments such as a velocity counter, a tour counter, etc.

When one wants to move between two towns as quickly as possible, one will use his Ferrari and not a tractor or a truck. But if one is moving in campaign across earth roads, he will use his four wheel drive car (choosing a Ferrari is an error here). In case one wants to transport forty tons goods from Kinshasa to Matadi (about three hundred fifty kilometers), he will then use his truck.

All vehicles allow us to move from one point to another but they do not provide the same services and have not the same efficiency according to the context in which we want to move. When one touches slightly the acceleration pedal of a Ferrari, it will jump from 0 to 300 km/h in a few seconds. On the contrary, when one pushes on the accelerator pedal of a tractor, he needs to insist to get the tractor to start moving and go from 0 to 30 km/h after several minutes. While the Ferrari will transport 2 or 3 persons, a truck will help to transport several tenths tons of goods.

The situation is the same for programming languages. Each programming language allows us to tell a computer about the job it has to do. But according to the fact that we need to manage a big stream of data or that we have to implement part of the automatic pilot of a plane, we will have to choose different programming languages.

Nowdays, there are tremendously large number of programming languages, so it is impossible to study them all or to use all of them in a project. Then which one do we choose in case we want to study a programming language or to develop a software project? Answering properly this question is a quite complicated problem.

The problem has been studied from different points of view by several authors [BRUGESS *et.*AL., 1995; HOWLAND, 1997; NAIDITCH, 1999; MCIVER, 2002; SMOLARSKI et al., 2003; PARKER et al., 2006; PARKER et al., 2006; HARPER, 2012; RABAI et al., 2014; RAY et al., 2014; ROUYENDGE et al., 2014; ELGMAL et al., 2014; HOWATT et al., 2015; JUHARTINI et al., 2015; BERNSTEIN et al., 2016].

For KRUGLYK et al., [2012]: "an educational language must contain all main programming concepts, while in general, programming languages are designed for industries and do not

match that criteria". So, for them, the only way to get a good educational programming language will be to choose an industry language and build some pedagogical, methodological, and technical complements around it.

From the industry prism, the selection of a programming language may be based on the project goals, as proposed by JOSHI et al., [2012]. From that point of view, it is suggested before starting a big project to test independent parts of code on small and independent pieces of the problem to solve. It is then possible from these experiments to select the best programming language and sometimes to use several programming languages, each addressing a specific part of the project.

The objectives of this paper are to surround the mathematical characteristics of the problem of selecting a programming language for any purpose and to propose a unified procedure that guides one in the selection procedure. To get to our objectives, we start by clarifying concepts of programming language, good programmer, what we want to teach in a programming language course and what is the impact of the programming language in a software development project. These clarifications are made with, as a central thread, the point in gout of all concepts, principles, problems and techniques of importance in a programming language selection procedure.

GENERALITIES

Programming languages are means of expressing computations in a form that is comprehensible to both machines and people. from that point of view, they are communication interfaces between machines and people. according to Herbert Meyer (see [AL-QAHTANI et al., 2010]): "no programming language is perfect. there is not even a single best language; there are only languages well suited or perhaps poorly suited for particular purposes". so how then can we choose a programming language for a given purpose? does it exist a dedicated selection procedure for each type of problem to solve (for example teaching and software development)?

Before trying to answer these questions, let us recall how a programming language is designed. for van Roy et al., [2004], each programming language supports and implements features in a certain way according to a computation model. a computation model is a formal system that defines how computations are done. among others, one definition of computation model which is important for programmers consists of data types, operations, and a programming language. such a computational model is also called a programming paradigm. more details about the programming paradigm concept are provided by van Roy et al., [2004] and by Nashas et al., [2012].

NANZ et al.,[2015] states that most of the time, compromises and arbitrations are made about which features to support and how to implement them. So according to the objectives of a school, it can choose to teach a given programming paradigm and then choose a programming language which implement that paradigm in the best way. But this is not the best selection criteria as to a given programming paradigm are attached sometimes tenths of programming languages.

Some programming languages, example LISP manage properly lists, some others are good for scientific computing (FORTRAN, C), some others are better for business applications (COBOL). So there exist classifications of programming languages according to the specific domain they address. That could be a useful criterion for programming language selection.

To make it short, a good programming language is an elegant interface for writing software. Here are essential pieces of such a programming language:

- Syntax: how to write valid programs using the considered language;
- Semantics: what does mean any word, any sentence of a valid program? For example, how expressions are evaluated;
- Idioms: what are common approaches to using the language features to express computations;
- Libraries: what has already been written for you?
- Tools: what is available for manipulating programs in the language (compilers, interpreters, debuggers, integrated development environment, etc.).

Libraries and tools, specific to any programming language, are essentials for being an effective programmer in a given programming language while syntax, semantics and idioms are fundamentals, concepts attached to any programming language, but implemented in a different way by each of them.

So programming languages could also be classified according to their syntax and semantics and language selection can be made on basis of how syntax and or semantic implementation fit with problems to solve.

From the paper of NANZ et al., [2015] it is clear that the design of a programming language is the result of multiple trade-offs that achieve certain desirable properties, such as speed, at the expense of others, such as simplicity.

However, technical aspects are not the only relevant concerns when it comes to choosing a programming language. Factors such as heterogeneity of supported domains, a strong supporting community, similarity to other widespread languages, or availability of libraries and efficient tools are often instrumental in deciding about a language popularity and how it is used.

From preceding paragraphs, it appears that it is not easy to come out with what the best programming language is. What is sure is that, we have pointed out some primitive principles, main characteristics of programming languages and some directions of programming languages taxonomies.

For JOSHI et al., [2012], and White [1990] a good programmer is someone who writes efficient programs in respect of specifications, time delay and care about quality and maintenance aspects for these programs. All informatics and related sciences and engineering schools where programming is taught, would like to make their students good programmers. That is not an easy task.

We shall, for example, know that programming is an art. To demonstrate the preceding assertion asks to a set of programmers, coming from the same school and exposed to almost the same experience, to write a small program for a given purpose, and you will get a set of programs of different qualities according to the intrinsic abilities of the programmers.

For VAN ROY et al., [2014], the most difficult work of programmers is not writing programs but rather designing abstractions, as programming a computer is essentially designing and using abstractions to achieve assigned goals. For them, an abstraction could be defined, in an "abstract way", as a tool or device that solves a particular problem. A given abstraction can be used to solve many different problems. Good programs must be readable and understandable by people and executable without errors by machines. So, good programmers may have the ability to visualize the consequences of the action under consideration as indicated by ABELSON et al., [1996].

We want that every time our students start a project, they master the programming language beside others aspects of the project. Mastering a programming language is not something else than mastering its essentials pieces we mentioned above in this section: syntax, semantics, idioms, libraries and tools.

Most of engineering schools have then as strategy to teach

programming in two steps. First, students will learn syntax, semantics, and idioms as fundamental concepts in a course such as "Programming languages". In such a course, all the concepts common to all programming languages are studied. The fact that designing a programming language is a project and that most of the time it is possible to get divergent requirements in specifications of such a project are discussed. The fact that the designed language is a result of compromises between several parameters (resources, speed, etc.) is explained.

Then in a second step, once the fundamentals and common concepts of programming languages are mastered, it is time to master tools and libraries of a specific language in order to become an efficient programmer. In case of a project, it is only when the project is set up and one or several programming languages identified that it becomes possible to be familiar with libraries and tools of these languages. Here the time to market, the maintenance and the quality of the product also come into account.

In schools, the two steps described here are sometimes reversed. And this doesn't really lead to any troubles. The important thing here is that the integration of fundamentals concepts knowledge and the mastering of libraries and tools should be done at a certain moment.

The analogy to vehicles we mentioned in section I can also help us understand what a good programmer is. Indeed, a good mechanical engineer knows how cars work, how to get the most out of them and how to design better ones. But he also needs to have a specialty, a type, a model of car on which he works on every day, a car that he can assemble and disassemble in a minimum delay without getting in trouble.

We can say for a good programmer exactly the same as we said for a good mechanical engineer. Indeed, beside the fact that he has to understand how to tell a computer about its job in general (semantics, idioms, syntax) , he also needs to master tools and libraries of one or a few programming languages in order to be able to write efficient programs and solve problems.

THE PROBLEM OF SELECTING A HIGH LEVEL PROGRAMMING LANGUAGE AND ITS COUNTLESS ASPECTS

As mentioned above, the problem of selecting a programming language among several hundreds of them for teaching or for a software development project has been studied by several authors. For HOWATT, [1995]: "Users do need to know the strengths and deficiencies inherent in a language, and how well a language applies to an application domain. But even within an application domain, requirements for two distinct projects may vary widely. One product may have to be highly reliable and portable, while another may have to be extremely efficient. Thus, knowing how well a language supports an application domain may not suffice; we also need to know how well a language supports needs of particular projects within a domain".

For NAIDTH,[1999] it is necessary to avoid that "... discussions about what language to select break down into emotional appeals for one's favorite language. Instead, a fair and rational process should be established that bases the choice of a programming language on issues related to customer needs and business goals...". For us the idea is even more general as customers can be the users of a software product or the users of schools products. PARKER et al., [2006] established that the most important was to derive a set of best selection criteria and to define a selection process that uses these criteria. In another paper PARKER et al., [2006] provided the same year a formal selection process for programming language selection.

Elgamel et al.,[2014], in their paper in the world Applied Science Journal, stands that the most important aspect of the problem is how to find the best selections criteria. An important question which comes to mind is: after a selection procedure, what do we get? The best programming language? The right programming language? Or just a programming language that fulfill a given problem requirements?

Sometimes it is necessary to consider directly user requirements as suggested by JOSHI et al., [2012]. So, if for example, the user expects a rapid implementation, then it could be interesting to choose a rapid application development (RAD) programming language. And even in such cases, according to the experience of the developer, the choice could be different.

WHITE, [1990], in his paper "The Comparison and Selection of Programming Languages for High Energy Physics Applications", reports that for industry, the programming language is a critical parameter of a software development project and it determines the rapidity of development, the easiest of maintenance, the portability of the software between plate-forms which are some of the most important aspects of a software development project. Even the history of the language plays a central role, compatibility with existing tools and libraries and assurance that the language is still under development and will continue growing with the software product for several years are of capital importance when making a choice.

In choosing a programming language, there are beside technical issues, the welcome industry, academics and individual reserves to languages. Indeed, several very good programming languages didn't survive because they were so academic, so scientific that the industry, the common programmer, didn't give them a chance to evolve. Some of these languages remain as academic treasures but with no impact on the industry and to the common programmer.

Nowadays, some classifications exist to try to measure the acceptance and or the popularity of a programming language. We can mention the TIOBE Quality Indicator, [2016], and The Redmonk Programming Language Rankings,[2015], indexes, etc. These indexes are sometimes used as language selection criteria.

Some of the reasonable questions to answer before choosing a programming language for any purpose are:

- what libraries are available for reuse?
- what tools are available and how easy to use are they?
- what is the ''de facto'' industry standard?
- how precise and easy to use is the syntax?
- what do I or my team know already?
- what about the portability?
- what features are implemented?
- what about the support in case of any problem?

According to the background of the persons in charge of the problem, the short list above can become a not ending one.

Among several papers describing empirically how to select a programming language from a given set of candidate languages, PARKER et al., [2006] introduced a more formal selection process. They summarize that process as follows:

i. Creating a language selection form based on a set of language selection criteria;

ii. Selecting a set of language evaluators. Each evaluator having to give a weight to each selection criteria according to the organization's needs (university, faculty or project); if necessary, average weights are computed and provided to all evaluators;

iii. Evaluators assign a score to each selection criteria for each candidate language. Then the score assigned to each criteria is multiplied by the weight of that criteria and these products are summed to obtain an overall rating for each language as follows:

$$\chi_i = \sum_{j=1}^{n} \omega_j \nu_{ij}$$

where:

- χ_i is the candidate language i. i=1 ...N;
- ω_j is the weight associated to the criterion j; j=1 ...M;
- ν_{ij} is the score of the language i for the criterion j.

The language with the highest score is then the optimal choice. It is always possible to get more than one optimal choice.

MATHEMATICAL ASPECTS OF THE PROBLEM

The selection procedure we propose is based on the one of PARKER et al.,[2006] mentioned in the section above. Let's say we have a set of programming language χ_i with $i= 1 \ldots N$, we can always identify a set of selection criteria C_j with $j=1 \ldots M$. Then it is possible to associate to each selection criteria C_j a weight ω_j $(j = 1 \ldots M)$. The selection process can be run by one or several evaluators. Let's consider the case of several evaluators. Each evaluator has then to attach to each selection criteria a score ν_{ij} $(i=1 \ldots N; j= 1 \ldots M)$. Then each evaluator come out with the following system of equations

$$\begin{pmatrix} \chi_1 \\ \chi_2 \\ \chi_3 \\ \vdots \\ \chi_N \end{pmatrix} = \begin{pmatrix} \nu_{11} & \cdots & \nu_{1M} \\ \vdots & \ddots & \vdots \\ \nu_{N1} & \cdots & \nu_{NM} \end{pmatrix} x \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \vdots \\ \omega_N \end{pmatrix}$$

We have then K such systems of equation in case of K evaluators. The programming language with the maximum sum of products of weights and scores is the selected language for the considered evaluator. For all the K evaluators, the mean sum of products of scores and weight for the language i is

$$\chi_i = \frac{\chi_{i1} + \chi_{i2} + \cdots + \chi_{ik}}{K}; \qquad k = 1, \ldots, K$$

Then the selected language is the one with the maximum χ_i. In case the maximum is note unique, then another run of the algorithm with more premises criteria's is necessary.

From our point of view, this way of doing could be improved. Indeed, the set of selection criteria depends on two main factors:

- The goals of schools or projects teams;
- The backgrounds and attempts of all persons involved in the selection procedure and the one of the users of the products to be developed (software application) or the organizations that will exploit the students considered as final products of their schools.

Where an instructor or a manager defines a set of selection criteria, another one will come with a slightly different set or a completely different one. Thus taking into consideration all the possible criteria, all persons involved in the procedure, all possible customers of project products and all the organizations that would use graduate students, the number of selection criteria can grow very quickly. It is even clear that once the number of candidates programming languages and persons involved in the selection procedure increases, the set of selection criteria becomes infinite. In these conditions, having an infinity list of selection criteria, we need to limit the problem to a given context, to the background and to the attempts of a limited number of persons involved in the selection procedure, every time we want to get a solution. Mathematically speaking, every time we want to get a solution, we need to make an approximation. We can then introduce an assertion in the problem analysis.

For each set of candidates programming languages $\{\chi_1,\chi_2,\chi_3, \ldots, \chi_N\}$, an associated set of selection criteria $\{C_1, C_2, C_3, \ldots, C_M\}$ and a list of K evaluators (N, M, and K ; For N, M, and K sufficiently big, it is always possible to find another set of criteria's $\{C_1, C_2, C_3, \ldots, C_M\} \cup \{C'_1, C'_2, C'_3, \ldots, C'_M\}$ which if associated to the set of programming languages expresses exactly the same problem and where associate to the same set of programming languages set expresses also the same problem.

In more general terms, the assertion here above stands that, for any programming language selection problem expressed by a set of programming languages χ, a set of evaluators K, and a set of selection criteria C it is always possible to replace C by another set of selection criteria C', or to enlarge it to a new set $C'' = C'UC$. This is achieved for example by changing a member of the evaluators set. This means that it is possible to extend without limit the set of selection criteria.

It is also clear that for small problems there are no complications and the problem can be solved without any computations. Persons involved in the selection procedures can make a decision just by speaking to each other. But, once the number of programming languages candidates and the number of persons and organizations involved in the selection procedure increase, then the number of elements of the selection criteria set become infinity. So the time necessary to compute the selected programming language become infinity.

In mathematics, there exist ill posed problems and NP-complete problems. An ill posed problem is a problem which may have more than one solution, or in which the solution depend discontinuously upon the initial data. According to KABANIKHIN [2008], most of the inverse problems are ill posed problems. Normally a programming language is designed to solve a given type of problem. So the problem of finding a programming language which can help to solve a given problem sounds like an inverse problem. It even satisfies the "almost rule" of KABANIKHIN [2008] according to which most of the inverse problems are ill posed problems. Indeed, the solution is not unique as in the programming language selection problem, there exist always at least 3 different solutions: "the best", "the right" and "the" programming language. Another aspect is the fact that the solution is tightly coupled to the persons responsible for the selection procedure. So for a given problem the solution could change each time you change the persons involved in the selection procedure. This sounds like the discontinuity of the solution according to the initial data attached to ill posed problems.

An NP-complete problem is a problem for which there is no existence of polynomial algorithm to solve it [CORMEN et al., 2009]. Or it is clear that for the programming language selection problem, once the number of persons and structures involved in the selection procedure increases, the set of possible selection criteria become a set of cardinal infinity and it is not possible to find a polynomial algorithm that can compute a solution for the problem. The tentative of making the problem more objective than subjective initiated by PARKER et al., [2006] seems to have success for small problems but once the number of persons and organizations involved in the selection procedure increases, the subjectivity of the problem becomes the dominant aspect making the selection criteria set an infinity set.

Fortunately, the problem fits to the class of Multi-Criteria Decision Making (MCDM) problems as already pointed out by

ROUYENDEGH et al.,[2014] and PARKER et al., [2006] in two different papers. Here the problem is defined as a problem of finding the best alternative for a decision maker, or finding a set of good alternatives or sorting or classifying alternatives. The number of alternatives could be infinite or discrete. Here the problem is expressed mathematically as the following:

$$(maximize)\chi(x)$$

where $\chi(x)$ is a vector of M criteria (objectives functions) and A is the feasible set. If A is defined explicitly, the problem is a multiple criteria evaluation problem. If A is defined implicitly, by a set of constraints, then the problem is a multiple criteria design problem. Getting a solution of the problem in these conditions requires then the use of specific Multi Criteria Decision Methods (MCDM) as did PARKER et al., [2006].

One may wonder if it is really necessary to go so deep in the programming language selection problem. Once again the answer depends on the context and people involved in the procedure. What is clear is that we have analyzed the problem from all its facets and it is up to working people to use one or another way in order to get a solution.

UNIFIED PROCEDURE FOR PROGRAMMING LANGUAGE SELECTION

In the previous section, we have analyzed the most important issues of the programming language selection problem. The problem can be understood and solved in different manners. There exist always more than one solution and the problem is a decision making one. The decision, the choice can be made after computations or without any computations. In the following, we present what we call a unified procedure for programming language selection. The procedure is the following:

0. Identify the problem
1. Draw up the list of candidates languages from the approximately 700 existing ones [WIKIPEDIA, 2016]);
2. Establish a finite list of selection criteria (an approximation is made here as in general the list of selection criteria is infinite for non empirical problems, see appendix I.);
3. In case less than 10 persons are involved in the selection procedure and the number of candidates programming languages is approximately the same, initiate discussions between concerned persons and select the appropriate language. In case it is not possible to make a decision, go to the next step; Selection could be done after a survey.
4. Express the problem in the formal form, see [PARKER et al., 2006];
5. Select a set of suitable evaluators;
6. Compute the problem manually, using excel or any specific tool and select the programming language (MCDM are used here). In case we end up with a solution made of several programming languages, add more selection criteria's and repeat 7. In case it is not possible to get a solution, go to next step;
7. Use another MCDM method to solve the problem.

Using the unified selection process requires lists of programming languages and a set of selection criteria. We present, as an illustration, a list of selections criteria in the appendix I, most of them from the paper of, PARKER et al., [2006] and one can get a list of candidates programming languages from the online encyclopedia [WIKIPEDIA, 2016]).

CONCLUSION

In this paper, we have examined the problem of selecting a programming language from a list of candidates languages for any purpose. After having examined the formal way of selecting a programming language introduced by PARKER et al., [2006], we have then showed that the programming languages selection problem has both characteristics of ill posed problems, NP-complete problems and multi-criteria decision making problems and this is the first important contribution of this paper.

Then we proposed what we called a unified procedure for programming language selection. That procedure is the second important contribution of this paper. A special effort was made to explain properly all the concepts and to express in mathematical form the problem. Then, we wondered if it was necessary to go so deep in solving the programming language selection problem. Once again, the answer to the question belongs to the persons responsible for the problem under consideration pointing out, if again necessary, the subjectivity of the problem.

It is also important to note that the attempt of PARKER et al., [2006] to make the problem more objective than subjective, was successful only for problems that could be solved without any computations or a very limited set of computations, as soon as the number of languages and persons involved in a selection problem grows, the problem become NP-complete and needs to be approximated.

Another lesson to be drawn from this paper is that research should be pursued, to classify and store programming languages selection criteria and to develop a global presentation format of all the existing high level programming languages if not already existing. In such a format, each language should be characterized by its purpose, existing tools and libraries, projects in which the language has been used and schools where the language is taught. Such a description could be very useful when selecting a programming and would help software developers and teachers to avoid developing new programming language when it is possible to use an existing one even if it is necessary to extend it.This lesson is the fourth important contribution of this paper.

RÉSUMÉ

La sélection d'un langage de programmation pour les besoins de l'enseignement ou pour la mise au point d'un logiciel est, contrairement aux apparences, un problème difficile. Dans ce papier, nous avons passé en revue les différentes analyses existantes du problème avant de proposer une méthode unifiée de sélection de langage de programmation pour tout type de problème. Pour atteindre cet objectif, nous avons dû mettre en lumière les différentes particularités mathématiques du problème. Ce qui nous a permis de montrer que ce problème porte les caractéristiques d'un problème mal posé, celles d'un problème NP-difficile et celles d'un problème de décision multicritères.

Mots-clés

Langage, programmation, choix, critères, sélection, logiciel, enseignement.

REFERENCES ET NOTES

ABELSON H., SUSSMAN G. J.[**1996**]. Structure *and* Interpretation of Computer Programs, MIT Press, second Edition.

AL-QAHTANI S. S., GUZMAN L. F., ARIF R., TEVOEDJRE A., PIETRZYNSKI P. [ANNEE]. *COMPARING SELECTED CRITERIA OF PROGRAMMING LANGUAGES JAVA, PHP, C++, PERL, HASKELL, ASPECTJ, RUBY, COBOL, BASH SCRIPTS and SCHEME REVISION 1.0,* [online] *arix.org/abs/*1008.3434.

BERNSTEIN G. L., KJOLSTAD F.[**2016**]. *Why New Programming Languages for Simulation?, ACM Transactions on Graphics, Vol.* 35, No. 2.

BURGESS C.J.[**1995**]. *SOFTWARE QUALITY ISSUES WHEN CHOOSING A PROGRAMMING LANGUAGE, WIT TRANSACTIONS ON INFORMATION and COMMUNICATION TECHNOLOGIES, VOL. 14.* [online] *www.witpress.com/elibrary/wit-transactions-on-information-and-communication-technologies/*14/10584.

CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C.[**2009**].

Introduction to Algorithms, third Ed., MIT Press.

ELGAMEL L., SARRAB M.[**2014**]. *Selection of Programming Languages for Developing Distributed Systems, World Applied Sciences Journal* 31.

HARPER R.[**2012**]. *PRACTICAL FOUNDATIONS FOR PROGRAMMING LANGUAGES, CREATIVES COMMONS LICENCE. [online] http://creativescommons.org/licences/by-nc-nd/3.0/us/*

HOWATT J.[**1995**]. *A project Based Approach to Programming Language Evaluation, ACM SIGPLAN Notices, pp* 17-10.

HOWLAND J. E.[**1997**]. IT'S ALL IN THE LANGUAGE (Yet Another Look at the Choice of Programming Language for Teaching Computer Science)", South Central Conference, Texas.

JOSHI O. D., GUNDALE V., JAGDALE S. M.[**2012**]. Guidlines in Selecting a Programming Language *and* a Database Management System , International Journal of Advances in Engineering & Technology, March.

KABANIKHINS.I.[**2008**]. *Definitions and examples of inverse and ill-posed problems, J. Inv. Ill-Posed Problems,* 16, pp 317-357.

KRUGLYK V., LVOV M.[**2012**]. *CHOOSING THE FIRST EDUCATIONAL PROGRAMMING LANGUAGE, [online] ceur-ws.org/Vol-*848/ICTERI-2012-CEUR-WS-paper-37-p-188-198.pdf.

MCIVERL.[**2002**]. *Evaluating Languages and Environments for Novice Programmers, In J. Kuljis, L. Baldwin & R Scoble(Eds). Proc. PPIG* 14, pp 100-110.

Naiditch D., "Selecting a Programming Language for Your Project", IEEE AES Systems Magazine, September 1999.

Nanz S., Furia C. A., "A Comparative Study of Programming Languages in Rosetta Code," arXiv:1409.0252 v4, January, 2015.

NASHAS M. *and* MAAITA A.[**2012**]. *CHOOSING APPROPRIATE PROGRAMMING LANGUAGE TO IMPLEMENT SOFTWARE FOR REAL-TIME RESOURCE- CONSTRAINED EMBEDDED SYSTEMS, EMBEDDED SYSTEMS - IN THEORY and DESIGN METHODOLOGY, TANADA K. (ED), ISBN: 978-953-51-0167-3, ,* [online] http://www.interchopen.com/books/embedded-systems-theory-design-methodology/.

PARKER K. R., OTTAWAY T. A.[**2006**]. Criteria for the selection of a programming language for introductory courses, International Journal of Knowledge *and* Learning, January .

PARKER K. R., CHAO J. T., OTTAWAY T. A., CHANG J.[**2006**]. *A Formal Language Selection Process for Introductory Programming Courses, Journal of Information Technology Education, 5*, .

RABAI L. B. A., COHEN B., MILI A.[**2014**]. *Programming Language Use in US Academia and Industry, Informatics in Education, Vol No 2*., pp 143-160.

RAY B., POSNETT D., FILKOV V., DEVANBU P. T.[**2014**]. A Large Scale Study of Programming Languages *and* Code Quality in Github, in FSE November , Hong Kong China.

"THE REDMONK PROGRAMMING LANGUAGE RANKINGS: JANUARY 2015, ", [online] *redmonk.com/sogrady/*2015/01/language-rankings-1-15/

ROUYENDEGH B. D., LESANI H.[**2014**]. Object Oriented Programming Language Selection using Fuzy AHP Method, International Journal of the Analytic Hierarchy Process, June-July.

"SEPR *and* Programming Language Selection", www.stc.hill.af.mil, February 2003.

SMOLARSKI D. C.[**2003**]. A FIRST COURSE IN COMPUTER SCIENCE LANGUAGES *and* GOALS, TEACHING MATHEMATICS *and* COMPUTER SCIENCE. [online] http://tmcs.math.klte.hu.

JUHARTINI, SUYANTO M. [**2015**].*The use of Programming Languages on Final Project Report by Using Analytical Hiearchy Process(AHP), International Journal of Advanced Computer Science and Application, Vol. 6*, No 9.

TIOBE QUALITY INDICATOR, "THE TIOBE QUALITY INDICATOR A PRAGMATIC WAY OF MEASURING CODE QUALITY", TIOBE SOFTWARE, [online] *www.tiobe.com/files/TIOBEQualityIndicator_v3*.11.pdf.

VAN ROY P., HARIDI S.[**2004**]. Concepts, Techniques, *and* Models of Computer Programming, MIT Press.

WIKIPEDIA [**2016**]. THE FREE ENCYCLOPEDIA. [online] en.wikipedia.org/wiki/list_of_programming_languages.

WHITE B.[**1990**]. The Comparison *and* Selection of Programming Languages for High Energy Physics Applications,, International Workshop on Software Engineering, Artificial Intelligence *and* Experts Systems for High Energy *and* Nuclear Physics, Lyon Villeurbanne, France.

Appendix I. Some Programming languages selection criteria

Selection criteria	Selection criteria
Supported platform	What are the hardware and the deployment costs
Elasticity of the language	What programming paradigm is supported
Time to production	Object orientation yes/no
Performance	Garbage collection functions
Support and community	Type integration
How efficiently one can write	Static errors handling
What are the cost of using the language	Dynamic errors handling
Multi-threading	Costs of training
Reflection	Cost of witting and testing code
Available libraries	How easy is the maintenance
Encapsulation	Simplicity
Inheritance	unambiguity
Polymorphism	Portability
Supported operations	Industry preference
Available tools	Available documentation
Interoperability	Availability of student/academic version
Academic acceptance	Availability of textbooks
Marketability of graduates	Operating system dependence
Proprietary/Open source	Development Environment
Integrated Development environment (IDE)	Debugging facilities
Ease of learning fundamentals concepts	Implemented programming languages features
Web development	Application domain
Available experience in the team or for students	Modularity
Reusability	Automatic Garbage
Securities features	Compiler
Interpreter	Parallel computing
Full support of object oriented programming	Full support of procedural programming
Full support of functional programming	Support of modern software engineering principles
Completeness of language features	Safety and reliability