Volume 5, Issue 11, November 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Importance and Role of Programming Languages in Software Quality Improvement

Deepak Kumar Verma*, H. S. Shukla
Department of Computer Science, D.D.U. Gorakhpur University,
Gorakhpur, Uttar Pradesh, India

Abstract— The quality of a software product depends upon the features of the product which meets the need of customers and thereby provide product satisfaction. The success and failure of a software product is dependent on the satisfaction of the customer/user. If the user is able to use and modify the things easily for a long time and remains satisfied then only a software succeeds. And a Software fails- When its users are dissatisfied, When it is error prone, When it is difficult to change and even harder to use. We believe that the programming tool plays an important role in the success and failure of a software product. When we plan for a software solution, we have many different programming languages to choose from, and it's easy to get lost in the intricacies of each one. The choice of a programming language therefore constitutes a strategic decision in achieving software quality. Here in this paper we have analysed various programming languages based on various characteristics and shown their impact on software quality.

Keywords— Language, Language attributes, Software Product, Software Quality, Quality attributes.

I. INTRODUCTION

In modern era of computerization the need of software is increasing day by day very quickly. It's a very long time since the term "software engineering" was introduced. Till date we have progressed very much in the area of software development but we are still struggling with the same type of problems, for example, cost overrun and software defects. It is true that we are developing more complex systems today than for 40 years ago, but the basic problems are the same. New technologies and methodologies will not solve the problem; there is no silver bullet [1]. The fact still remains that the key asset is the people developing the software. This was made perfectly clear almost 30 years ago in the book by Boehm [2]. The most important factor is the people! Other studies have also reported in the large differences in performance between individuals; see for example [3, 4, 5].

Before turning our attention to the people, we would like to highlight some general success factors, which in retrospect determine whether a specific software project has failed or become a success. The following four factors are important success factors:

- Cost: primarily related to effort, i.e. person-hours, and productivity,
- Cycle time: time from development starts to delivery,
- Quality: this is a complex attribute where one important Part is defects,
- Predictability: the ability to predict the other attributes.

The bottom-line is that to achieve quality in the software, it is necessary to understand which factors that affect software quality. To shed some light on one of the most important factors, namely people issues, and in particular the importance of specific language knowledge, In this paper, we have studies various languages & their impact on developing high quality software. The objective of the study is to investigate the choice of programming language for a particular application is a difficult one and is often decided as the result of factors outside the immediate constraints that one might be considering for quality of software such as - reliability, reusability, security, portability efficiency and maintainability.

II. ISSUES AFFECTING SOFTWARE QUALITY

There seems to be three main aspects of a programming language and its implementation which have software quality implications.

- The Design of the Programming Language.
- The Specifications of the Programming Language.
- The Quality of the Compiler Implementation.

Because of the nature of such systems, and their inherent complexity, there are certain additional criteria that must be addressed. Not all of these criteria are applicable to all languages, but all of them apply to real time languages. The comparison of general and technical information for selection of commonly used programming languages shown in Table1:

Table-1

Language	Intended use	Paradigm(s)	Standardized
Ada	Application, embedded, real-time, system	Concurrent, Distributed.Generic, Imperative, Object oriented	1983, ANSI, ISO
C	System	Imperative procedural	1989,
C++	Application, System	Generic, imperative, object oriented, procedural	2011, ISO
C#	Application, business, client-side, General, server-side, Web	Functional, Generic, imperative Object oriented, Reflective	2000,ISO
Fortran	Application, numerical computing	Generic, Imperative, procedural	ISO 2003
Java	Application, business, client-side, General, server-side, Web	Generic, imperative, Object oriented, Reflective	De facto standard
JavaScript	client-side, Web	functional, imperative, prototype-based, Reflective	1997, ECMA
Visual Basic	Application, education	Component-oriented, event-driven, imperative	No
Visual Basic.NET	Application, Education, Web	Event-driven, Imperative, Object oriented	No

Here we have discussed various requirements of the real time languages which affect the quality of the software product are:

Simplicity: Perhaps the overriding requirement for any language is that it be inherently simple. Achieving this leads to a number of advantages when one considers training, maintainability and portability. A good example of such a language is visual basic studio 6.0, Java and C# was developed for a specific need but was designed with simplicity as the uppermost requirement.

Security: A secure computer language is one which is able to deal with errors made during programming or which occurs at run time. Generally real time software needs to operate as reliably as possible and this places an added constraint that the language intrinsically allows one to create reliable programs. The first stage is clearly that errors made during programming be detected. The cost of correcting such errors at this stage is obviously less than during testing or, worse still, during service. As an example, if a language strongly typed (as shown in table 2) this helps prevent erroneous use of variables in expressions and forces the programmer to think more clearly about the way data is to be handled and transformed. The second state is the detection of run time errors. These are faults which were not detected during compilation. C++ and Java, for example, provides what it calls exception handling. That is, if some error condition occurs outside the predicted scope of the program, in other words an exception, facilities are provided to deal with the condition, which might otherwise lead to failure of the program.

Adaptability: The language must allow the programmer sufficient flexibility to deal with the external environment since, in the real time systems, it is most often the case that exotic peripherals need to be embraced by the system. Having to resort to machine code greatly jeopardizes the integrity of the system and language should, ideally, be adaptable enough to allow the sort of operations, which are necessary.

Readability: One often-overlooked aspect of languages design is the provision of constructs and facilities, which allow the programmer to produce an easily read, and thus easily understood, program. Readability takes on great importance when modification is necessary by a programmer who may not have been involved in the original development. A language such as FORTRAN does not enforce readability on the programmer, whereas C++, java [7], and visual basic [8,12] lends itself to better structuring and thus enhanced readability be virtue of its block structure/class, object structure.

Portability: The idea of language portability is one that has been around for a long time but its realization in an actual language has taken many years to effect. There is inevitably a compromise here since, very often, a language implementation is mapped strongly on to the underlying hardware, thus making machine portability almost impossible. However, by separating out implementation -dependent aspects it is possible to attain portability and thus amortize the cost of the system development over a number of different hardware environments. An example of this is the ISO definition of C, C++ (De jury standard) and java (De facto standard), which has allowed developers of name-based/object-oriented systems as, mentioned in table-1 to be confident of the portability between systems, which support this standard.

Efficiency: Efficiency in a language has a number of distinct and often conflicting aspects. Until the recent fall in the cost of hardware, one of the most important considerations was the efficient use of available memory. It was often necessary to go to extraordinary lengths to squeeze a program into the available memory, often with the consequence that some of the requirements listed above could not be met. A further aspect of efficiency is to provide a language which can achieve the execution speed necessary to react to the stimuli being monitored. With the relative decrease in hardware costs such considerations are becoming secondary and thus the onus of maintaining efficient programs falls on the language designer rather than the programmer [6].

III. DESIGN OF PROGRAMMING LANGUAGES

One of the main topics for consideration when designing a real time language is that of data typing. This is the feature, in a language, whereby each variable has to be bound explicitly to a specified data type. This has been specified in table - 2.

Table-2

Language	Type strength	Type safety	Type Expression	Type Compatibility among Composites	Type checking
Ada	strong	Safe	Explicit	Name-based	Partially Dynamic
C	strong	Unsafe	Explicit	Name-based	Static
C++	strong	Unsafe	Explicit	Name-based	Static
C#	strong	safe	Explicit with optional	Name-based	Static
Fortran	strong	Safe	Explicit	Name-based	Static
Java	strong	Safe	Explicit	Name-based	Static
JavaScript	weak	Safe	Implicit	Name-based	Dynamic
Visual Basic	Strong	Safe	Implicit with optional Explicit typing	Name-based	static
Visual Basic.NET	strong	Unsafe	explicit		static

Specified data type affects the security aspects of the language and its flexibility which is important parameter of software quality product. The readability will also be strongly affected by the language typing system. Whilst weak typing will allow more flexibility, for example manipulating bits, it is inherently insecure and wherever possible strong typing should be designed in with additional language features to make up for any possible loss in flexibility.

Program structuring can be divided into two levels. At the basic level, control structures are needed to specify the sequence in which basic program actions are executed. At the higher level, structures are needed to group sets of selected actions into single units. Together these two levels provide a mechanism of structured programming.

In multi-program systems the up-to-date approach is to introduce specific constructs for task specification, task communication and synchronization into the language itself. This leads to much higher security as greater checking can be done when compiling. It does, however, put greater emphasis on production of higher-quality compilers.

Low level or assembler programming cannot always be tackled in a high level language; indeed, it may be desirable to avoid doing so, unless a high level language can allow bit-level manipulation, it will always be necessary to resort to low level programming. This need tends to be recognized `and thus mechanisms are usually provided to facilitate the production of device drivers and the like.

One useful feature is that of separate compilation. Having to re -compile all programs each time a fault is corrected is both tedious and time-consuming.

Other features which need to be considered are the initialization of variables and features of input and output. It can be case that a programmer may omit to initialize a variable before use and it is often considered desirable to make a compile time decision to initialize all variable. Within real time systems, the provision of input/output facilities is particularly difficult. Since each system is likely to be specific to location, or a system in which it is embedded, such facilities will tend to become implementation-specific. One approach is to provide high level I/O facilities and leave it to the system implementer to provide the low level, system-specific part. The greater emphasis placed on the use of high level languages, together with efforts to standardize such languages, has led to the production of validation suites for languages.

The use of high level language, whilst conferring many benefits on the development and maintenance of software, has one important drawback. This is that the software developer becomes dependent on the skill and accuracy of the compiler writer to generate machine code which represents his actual intentions. Also, because of the demands! of portability, it is important that software need not be rewritten purely because of transfer from on machine to another. The idea behind the validation suites is not just to exercise a language for conformance to standard, but to provide deviance checks. Some suites look at quality and performance factors (e.g. speed of compilation). The development of such validation suites is difficult and they need to be under constant review and extension in order to be effective and meet the needs of users.

IV. SPECIFICATION OF PROGRAMMING STRUCTURES

The use of top -down, bottom-up and modular approach design methods leads to considering the necessity of providing some means of mapping that design on to the language. That is, the programming language must be capable of representing a successive refinement of the design and thereby continuing the bottom-up process. Perhaps the most important construct which begins this is the class. A class is a collection of objects and their operators which is encapsulated in such a way that access from outside the module is controlled. In software a class is a module. An attribute is a characteristics or property of an object.

Within this module the language just provides some forms of control statement as, for example

Class module:

```
Class student
{
   private:
          char * name;
           int age;
  public:
          void enterdata(void);
          void displaydata(void);
}
```

Conditional statement:

```
                IF (condition)
                THEN
                        statement
                ELSE
                        statement
```

Or more specifically:

```
        IF (X==0) THEN M=M+1 ELSE RETURN
```

Another example is WHILE statement:

```
WHILE condition
     {
        statement;
     }
DO
   {
        statement;
    } WHILE condition;
```

Such statements allow one to construct efficient and readable programs. The design of control statements is still a contentious area. Much attention has been paid to the use of the GOTO statement with various arguments presented as to its desirability or otherwise.

Once one is satisfied with the basic building blocks it is possible to consider the way in which they can be assembled within a module. One common approach is to assemble the program actions into some sort of block structure, perhaps delineated by the use of braces { }, private, public, local, global, implicit, explicit this makes it possible to define the scope of variables, which helps to improve efficiency in the program. Procedures and functions may be employed which allow the computation of frequently repeated tasks. Names can be given to these tasks such that they can be repeated as often as is necessary. Procedures also allow a better representation of the top-down structures. A function is a special form of procedure which has the specific task of computing a single value.

Therefore we come to the conclusion that the structure of the various programming language or program structure play crucial role to achieve the quality of the software product.

V. QUALITY OF COMPILER IMPLEMENTATION

A good compiler and user interface is essential if a programming language is going to be accepted by its users. One important aspect of this, which relates directly to software quality is the correctness of the implementation. This has two main aspects. The first relates to the conformance of the compiler to the language standard, which has been discussed in the previous section. The other issue is the correctness of the results produced by the compiler, which is concerned mainly with the accuracy of the generated code.

There are two main approaches to achieving a high degree of confidence in a compiler, that is, compiler validation and the automated testing of compilers. These two methods are complementary and have different strengths and weaknesses. Compiler validation involves the compiler correctly processing a carefully chosen set of manually constructed test programs which attempt to cover all the different features in a language and often include specific tests for areas which compilers are known, from past experience, to frequently get wrong. The first major validation suite, as opposed to a set of benchmarks, was produced for Pascal [15], but a number of others have been produced since, e.g. the Ada Validation Suite. These go a very long way to ensuring a good compiler but there are two related problems. The first is that no test suite, or any other method of testing large software, is anywhere near being exhaustive, and many combinations of constructions will not be tested. The other problem is that users tend to view the term validated as being synonymous to correct, but with the current state of the art, there is a large gap between validation and any proof or guarantee of correctness.

The automatic generation of test cases for compilers attempts to narrow this gap by automatically generating a large number of different test programs, often driven by a random-number generator, using some explicit or built-in representation of the language and its semantics. This approach has a number of drawbacks in terms of the total coverage possible but can be usefully used to complement the validation process based on manually constructed test suites.

VI. CONCLUSIONS

In this paper we have highlighted some of the major factors relating to the choice of a programming language for a software product that relate directly to the quality of the finished product, and the software development process. It has shown how the design of the language, the form of its specification, and the quality of the implementation, all have a significant effect on software quality. All these three are the major issues of programming languages that affects the quality of the software product. Programming languages play an important role in quality of a software product. So it is essential to choose the right programming language for a particular domain to achieve the quality of software product.

REFERENCES

[1] Balaguruswami, E.,"Programming in C",Tata McGraw Hills, 2006.
[2] Boehm, B. W., *Software Engineering Economics*, Prentice- Hall, 1981.
[3] F. Brooks, "Studying Programmer Behavior Experimentally: the Problems of Proper Methodology", *Communications of the ACM*, April 1980.
[4] B. Curtis, "Measurement and Experimentation in Software Engineering", Proceedings of IEEE, September 1980.
[5] Claes W, "Is prior knowledge of a Programming Language Important for Software Quality", Published in Proceedings of the first IEEE International Symposium on Empirical Software Engineeringpp27-36, Nara, Japan oct.- 2002.
[6] Smith, David j., Wood, Kenneth B., "Engineering Quality Software", Elsevier Science Publishers Ltd Second edition, 1989.
[7] Java ™ 2 Complete", Bpb Publications B-14, Connaught place, New Delhi 110001, first Indian edition:1999.
[8] Brian Siler, "Special edition using Visual Basic 6.0", Prentice Hall (2000)
[9] Balaguruswami, E.," Object-Oriented Programming with C++", Tata McGraw Hills, Third edition: 2006.
[10] Singh Brijendra, "Quality Control & Reliability Analysis", Khanna Pub. Delhi, Third edition: 2011.
[11] Holzner S.,"Visual Basic 6 Programming Black Book", Dreamtech Press, New Delhi, Aug.2007.

