Transactions on Information and Communications Technologies vol 11, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

Software quality issues when choosing a programming language

C.J. Burgess
Department of Computer Science, University of Bristol, BS8 1TR, UK

Abstract

For high quality software, an important part of the project is the choice of the programming language and compiler to be used. This paper examines some of the issues affecting software quality that should be considered when making this choice, including the extent to which the programming language encourages the writing of good quality programs; the degree to which the precise effect of the programs written in the language have been formally defined; and the degree of confidence that can be placed in the compiler implementation. As an important part of this, the paper will include reference to the role of formal definitions of programming languages, programming language standards and compiler validation techniques.

Introduction

For high quality software, an important part of the project is the choice of the programming language and compiler to be used. There are likely to be a large number of different factors to take into account when making this choice, such as the past experience of the programmers; the re-use of existing modules; the compilers available, including the speed of execution of their generated code; and the portability of the programs. These are undoubtedly important aspects to take into account, but these should be balanced against any adverse effects on the software quality, either in terms of the final product or on the software development process.

This paper concentrates solely on the software quality issues affecting the choice of language, which must then be given the appropriate weight when combined with all the other factors that influence the final choice.

Issues affecting software quality

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

These are:

1. The Design of the Programming Language.
2. The Specification of the Programming Language, and it's incorporation into a Programming Language Standard.
3. The Quality of the Compiler Implementation.

Each of these three aspects will be considered separately, although inevitably they are not completely independent of one another.

The Design of the Programming Language

Many of the major issues in programming language design have been extensively discussed over the years, probably ever since the term 'Structured Programming' was first used, if not earlier. It is fairly well established that the use of a language which actively encourages Structured Programming, such as Modula-2 or Ada, certainly helps to develop good habits when teaching programming, e.g. a recent paper by Jones and Burgess [1], but the same features that help a student learn good programming practice are also likely to help programmers produce good quality software during the software development process.

All the good aspects of program design relating to program structure, such as modularisation, abstract data types, data hiding, variety of sensible control structures, should be implementable in clear, explicit ways using the language in order to maintain software quality. In addition, there is a growing view that a language that supports strong-typing, which enables all the variables and data-structures used to have their types checked at compile-time, helps to avoid many problems, e.g. the problems caused by the many implicit coercions performed in PL/I, some of which the user may not have intended. Yet a very large amount of software is still written in languages, such as Fortran, or C, where either all these concepts are not possible to implement, or at the very best, they are difficult to make explicit in the final code without the extensive use of comments. This is not meant to be particularly critical of the C programming language as a whole, but only from the software quality viewpoint of the language design. A similar situation exists with object-orientated programming, where probably C^{++} is one of the most popular of the current implementation languages and yet it does not implement many of the features that you might like, e.g. multiple inheritance, and suffers from the same structural problems as for C. It probably owes its success to its close resemblance to C, and also what seems to be an unavoidable inefficiency in implementations of other object-orientated languages, such as Smalltalk [2, 3]. Whether this situation will change by the wider use of newer Object Orientated languages, such as Oberon [4, 5] or Ada9X [6], it is not clear but it certainly seems unlikely to change in the near future.

The conflict between a good language design from the point of view of style and the speed of execution of the object programs is not new, although with modern compilers and computer architectures there is probably little difference in the execution speeds possible for the mainstream types of imperative language. However, Fortran is still used a great deal for the more mathematical types of problem, not only because of a huge investment in existing software routines, but also due to the comparative ease by which very efficient execution times can be achieved and a long history of usage for these types of problem.

Some of the problems affecting software quality can be solved by imposing additional in-house restrictions on the use of the programming languages. One example of this is avoiding the use of dynamic storage allocation and pointer variables where very high-integrity software is required. Another example is insisting on the use of explicit declarations for all variables in languages which allow implicit declarations, so as to improve the checking that a compiler can do against accidental misspelling of identifiers.

Programming Language Standards

A good paper discussing the effects of programming language standards on software quality has recently been published by Wichmann [7]. There are a number of major decisions that the program standards committees have to make which can have a considerable impact on the type and form of standard produced.

Formal or Informal Standard

The first major decision is whether the standard should be in some form of structured English or whether a more more formal approach is more appropriate, possibly complemented by an English description. The structured English document provides a very much more readable document, which can be used by both programmers and compiler writers alike as a reference document but lacks any possibility of mathematical rigour. The arguments for and against using a formal form for the standard are very similar to the current debate about the use of formal methods for program specification when compared with more informal methods based on English descriptions. One of the earliest good attempts at producing a standard based on structured English and including a formalising of the syntax of the language was the Algol60 report [8]. This technique has been developed and refined over the years and is still used by most, if not all of the current ANSI standards, e.g. [9, 10]. At the present time this is probably the most widely accepted form of standard but since it is very hard to make such standards both complete and unambiguous, then it can have implications where very high levels of software integrity are required.

The other major approach is to attempt to formally define the whole language standard. As far as the author is aware, the first published language definition of this type was produced by IBM for programming language PL/I, and was based on the Vienna Definition Language. However, whilst this was a brave

attempt in its day, the idea was not followed up for any other major languages. Other serious attempts along this path have been the Algol68 report [11]. This used a two-level grammar, which enabled formal definition of the semantics as well as the syntax of the programming language, so that the effect of any program was completely defined. Unfortunately the report, when published, was over 200 pages long and could only be fully understood by a comparatively few experts. In addition, the complexity of the report made it difficult to ensure that even the formal definition was complete and did not contradict itself. A more recent approach along the same lines, but hopefully a lot more readable, is an attempt to include as part of the new standard for Modula-2, a definition based on the use of the formal specification language VDM. However it is to early to judge whether this will be any more successful, as none of this work seems to have been published yet.

Coverage of the standard

A closely related issue is whether the standard should cover only the language and its effect or also specify implementation issues. There are a number of different examples of implementation issues which would fall into this category. These issues have also been enumerated by Wichmann [7]:

1. Capacity limits for compilers

   This is normally specifically excluded from standards with the ANSI standard for C being an exception [10]. The lack of this can produce a very wide range of limits, some of which are not always easy to determine. An example of this for Modula-2 can be found in a paper by Pronk [12].

2. Floating-point arithmetic

   Although there are many applications where floating-point arithmetic is not required, none of the standards, except Ada [13], state much about the semantics of floating-point operations. This gap is gradually being closed by a proposed new separate standard on computer arithmetic [14].

3. Uniformity

   This concerns the degree to which two different implementations of the same language can differ from one another, raising questions for portability of software. A closely related issue, is whether the standard for the language explicitly includes or excludes subsets of the language.

4. Error detection and handling

   There is normally a complete absence of any reference as to how compile time or execution time errors should be handled by the compiler, with the exception of exception-handlers for the few languages that allow users to specify their own exception handling.

   At compile-time this relates to the form of the error message, the accuracy of the message and any pointer to the offending source code. Also whether

Transactions on Information and Communications Technologies vol 11, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

warnings should be given to indicate code that does not conform to the standard, but might be common past usage, or code highly likely to imply logical errors during execution even though the program is valid and would compile, e.g. declaration of identifiers that are never used.

At execution-time, whilst the standard often indicates errors that can occur, there is no obligation on the compiler to always check for them in the implementation. A very common omission, even as a user option, is any check for occurrences of integer overflow, as this can have a very considerable impact on the execution speed. Another is the checking for the use of values of unassigned variables.

These issues will often have an impact on the software quality of the final product either directly, or indirectly by affecting the software development process, in particular, the testing stage.

Quality of the 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 and has been covered 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.

Unfortunately, like most other sizable software, it is still impossible to prove the correctness of a compiler for a reasonable size language and a practical computer. In the meantime, 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. A review of this approach has recently been published by the author [16], and another paper by Boujarwah and Saleh [17] evaluates the effectiveness, in terms of the coverage achieved, of some of these types of automatically generated test suite. 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.

Conclusions

This paper has attempted to highlight some of the major factors relating to the choice of a programming language for a project that relate directly to the quality of the final product, and the software development process. It has shown how the design of the language, the form of its specification, the type of standard available, and the quality of the implementation, all have a significant effect on software quality. At the present time, there are probably a large number of other factors that have a more significant affect on software quality and there is still a limited choice in terms of the languages, standards and good implementations available, but the situation may well improve, particularly when the customer can demand a far higher quality that at present in the product, and still at an economic price.

References

[1] Jones, B.F. and Burgess, C.J., Training the next generation of software engineers: Is software quality being taught in British Degrees?, *SQM94 Conference*, published in *Software Quality Management II*, Vol. 1, pp. 637-650, Computational Mechanics Publications, 1994.

[2] Goldberg, A. and Robson. D., *Smalltalk-80 : the language and its implementation*, Addison-Wesley, 1983.

[3] Goldberg, A., *Smalltalk-80 : the interactive programming environment.* Addison-Wesley, 1984.

[4] Wirth, N., *The programming language Oberon*, Software - Practice and Experience, Vol. 18, pp. 671-690, 1988.

[5] Reiser, M., *The Oberon System, User Guide and Programmers Manual*, ACM Press, Addison-Wesley, 1991.

[6] Barnes, J.G.P., Introducing Ada9X, *ACM Ada Letters*, Vol. 13, No. 6, pp. 61-132, 1993.

[7] Wichmann, B.A., Contribution of standard programming languages to software quality, *Software Engineering Journal*, Vol. 9 No. 1, 3-12, 1994.

Transactions on Information and Communications Technologies vol 11, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

[8] Naur, P. (Ed.), The revised report on the Algorithmic Language Algol60, *C.A.C.M.*, Vol 6, pp. 1-17, 1963.

[9] American National Standard Programming Language Fortran, ANSI X3.9-1978, (Revision of ANSI X3.9-1966), *American National Standards Institute*, 1978.

[10] American National Standard Programming Language C, ANSI X3.159, 1989.

[11] Van Wijngaarden, A. (Ed.), Mailloux, B., Peck, J. and Koster, C., Revised report on the Algorithmic Language ALGOL68, *Acta Informatica*, Vol. 5, pp. 1-236, 1975.

[12] Pronk, C., Stress Testing of Compilers for Modula-2, *Software - Practice and Experience*, Vol. 22, No. 10, pp. 885-89, 1992.

[13] Ichbiah, J.D. et el., *Reference manual for the Ada programming language*, ANSI/MIL-STD 1815A, US Department of Defense, Feb. 1983, (and ISO-8652:1987).

[14] Information technology - language independent arithmetic, Part I: Integer and floating point arithmetic, ISO/IEC 10967-1:1993, 1993.

[15] Wichmann, B.A. and Ciechanowicz, Z.J., *Pascal Compiler Validation*, John Wiley, 1983.

[16] Burgess, C.J., The Automated Generation of Test Cases for Compilers, *Software Testing, Verification and Reliability*, Vol. 4, No. 2, 81-99, 1994.

[17] Boujarwah, A. and Saleh, K., Compiler test suite: evaluation and use in an automated test environment, *Information and Software Technology*, Vol. 36, No. 10, pp. 607-614. 1994.