I. In S/W Engineering for Microprocessor System ed. P. Depledge. Peter Peregrinus: London, 1984. 6

1983b

Chapter 1

Why software engineering?

I. Sommerville

1. INTRODUCTION

Like many other terms associated with information technology, the term 'software engineering' has been subject to a number of different interpretations. In some cases, it is applied to the construction of systems which require knowledge of both computer hardware and software. In other instances, it is simply another name for computer programming, and in yet others it is applied exclusively to the production of very large software systems.

However, if we take a dictionary definition of engineering and apply that to the term 'software engineering' we get a definition as follows:

> The profession of applying scientific principles to the design, construction, and maintenance of computer software systems.

This definition emphasises that software engineering is not just computer programming - the implementation of software systems - but includes all aspects of software production from initial conception through to maintenance after delivery of the completed software system.

The problems of developing software systems are akin to the problems which arise in other engineering disciplines - costs and complexity must be controlled and people must be managed and motivated. Software engineering embraces such non-technical considerations as documentation, user psychology, and software project management as well as the technical aspects of software design and production.

Software engineering is becoming of increasing importance as programmable microelectronic systems are used in more and more applications. Before such systems, the electronic systems engineer may have used computers to help him in his work but had no real need to develop well-engineered software systems. He may have been an amateur programmer writing 'cheap and nasty' programs but it was not cost effective for him to spend a lot of time developing professional quality software systems.

Now, software and hardware are so closely interlinked that it is essential that the electronic engineer has an understanding of the principles of software engineering and an appreciation of the true costs associated with software development. In the remainder of these notes, the stages of development of a software system, their associated costs, and the characteristics of well-engineered software are described.

2. THE SOFTWARE LIFE CYCLE

In the development and use of a software system, a number of distinct, interacting phases can be identified. These can be compared with the phases passed through by an insect or an animal as it is born, reaches maturity and ultimately dies. Hence, the term 'software life cycle' has been coined to describe these phases of a software system.

The stages of the software life cycle are:

(1) Specification
Before any software is actually produced, the functions of the software must be established and the operational constraints on that software defined.

(2) Design
The software specification must be analysed and a software design established. A software design is a machine independent statement of how individual programs units must interact to implement the software specification.

(3) Implementation
The software design is realised in a computer programming language which can be executed by the target computer destined to run the software system.

(4) Validation
This phase of the software life cycle is intended to validate that the implemented software meets the needs of the user. Sometimes, this stage is called 'testing and debugging' but this implies that it is concerned solely with validating the implementation phase of the life cycle. In fact, during this validation phase, it is common to detect errors and oversights in all preceding life cycle stages.

(5) Operation and Maintenance
The software system is put into use. As it is used, errors which have been missed by the validation phases are often detected and must be corrected. Furthermore, as the software system becomes an essential feature of the user's environment, the user's perception of what the software ought to do for him will change and the software must be modified to meet these changing user needs. Lehman (2) has called this process 'software evolution'.

As set out above, the software life cycle seems to be a straightforward linear progression from phase to phase. In

fact, it is a
phase interac
Usually, work
redone as probl
the design
discovered requ
parts of the
implementors,
programming lan
and possibly e
after the impl
throughout the
software proble
the life cycle

Because c
stages, it is di
individual phas
as Boehm (1), ha
life cycle
simplification
between the c
producing the s
software mainte
modifying the s
changing require
software develop

Specificat
for about 20% o
the remainder ta
system costs $1
of that developm
cost about $400
gone into operat
$5 000 000.

Typically,
times development
such as complex
art hardware and
the system maint
the development
estimated that fo
cost was $30 per
$4000 per instruc

3. THE CHARACT

Whilst it is all
engineering, the
software engineer
well-engineered a
There are a numbe
software - sp
implementation, re
given equal impor
engineered or not.

Bearing in

fact, it is a cyclic rather than a linear process where each phase interacts with preceeding and succeeding phases. Usually, work done in early stages of the life cycle must be redone as problems arise in the later life cycle stages. As the design progresses, specification errors will be discovered requiring the specification to be changed. Some parts of the design may place constraints on the implementors, others may be unimplementable in the programming language used for the project. Thus, the design and possibly even the specification may have to be changed after the implementation phase has begun. Changes occur throughout the software development process and many software problems have come about because the complexity of the life cycle model has not been recognised.

Because of the interacting nature of the life cycle stages, it is difficult to establish an accurate costing for individual phases of the life cycle. Most estimators, such as Boehm (1), have made the simplification that the software life cycle is a linear process. Taking this simplification into account, there is a gross imbalance between the costs of software development, that is, producing the software in the first place, and the costs of software maintenance. Maintenance costs, the costs of modifying the software to correct errors and to adapt it to changing requirements are orders of magnitude greater than software development costs.

Specification, design, and implementation each account for about 20% of the total software development costs with the remainder taken up by validation costs. Therefore, if a system costs $1 000 000 to develop, the most expensive stage of that development will be software validation which might cost about $400 000. Maintaining that software after it has gone into operation, however, is likely to cost at least $5 000 000.

Typically, software maintenance costs are about 5 times development costs. However, for some types of system such as complex real-time systems reliant on state-of-the art hardware and subject to tight efficiency constraints, the system maintenance costs may be several hundred times the development costs. An example of this, cited in (1), estimated that for one USAF avionics system, the development cost was $30 per instruction and the maintenance cost was $4000 per instruction.

3. THE CHARACTERISTICS OF WELL-ENGINEERED SOFTWARE

Whilst it is all very well to talk in glib terms of software engineering, the software life cycle, etc. the working software engineer is faced with the problem of recognising well-engineered as distinct from poorly-engineered software. There are a number of criteria which can be used to assess software - space and time efficiency, speed of implementation, readability, etc. not all of which should be given equal importance in deciding if the software is well-engineered or not.

Bearing in mind the distribution of costs over the software life cycle and the professional responsibility of the engineer, well-engineered software should exhibit three dominant characteristics:

(1) It should provide the facilities and operate within the constraints set out in the software specification.

(2) It should be reliable.

(3) It should be readily modifiable.

The first of these characteristics is, of course, a very general characteristic and places a great responsibility on the individuals responsible for drawing up the software specification. It is very common indeed for software systems to fail to meet the intentions of customers because of inadequate and ambiguous specifications and it is arguable whether such software should be considered well-engineered ot not. Our present notations for specifying software functions and constraints are grossly inadequate and until new notations are developed, it will remain very difficult to measure how well a software system meets its specifications.

As software systems become more diverse and are used in more and more application areas, it is becoming clear that reliability is the most important dynamic characteristic of a software system. This is partly due to the fact that software systems are now used as control systems in many larger systems whose failure could endanger life and partly due to cost considerations - unreliable software is very expensive indeed.

The immense costs of unreliable software can be illustrated by examples of situations where software system failure result in very high costs to the software customer.

Say a software system controls the drilling of an off-shore oil well. The cost of running an oil rig runs into thousands of dollars per hour and a software failure causing drilling to be suspended can result in costs which far exceed the original software costs.

Another type of situation where the costs of software failure may exceed the software development costs could arise in a situation where ROM based software controls the braking system in a car. Errors in the software, discovered after the car has gone into production, would result in all cars sold being recalled and the ROMs replaced. So far, this situation has not arisen but as ROM-based software is used in more and more products, it is likely that such a situation will eventually come about.

The third characteristic of well-engineered software - that it should be readily modifiable - arises directly from a consideration of life cycle costs. Because software maintenance, that is, the modification of existing software, is far and away the most expensive aspect of the software life cycle, the software development stage should be geared towards producing a readily maintained software system. This means that readability should take place over

writeability - prog
written, documenta
complete, and progr
be used so that
interacting system
closely interlinked

Notice that a
explicit feature of
This does not mean
cases efficiency in
the size of the
importance. Howeve
hardware becomes
complex and powerfu
likely to become le
software systems.

In those cas
precedence over co
ought to be clearl
The efficiency spe
to the microproces
system into as fe
construct a contro
is his responsi
specifications are
off between efficie
made clear to the s

To conclude,
professional softw
the real costs of s
problems, and on
the software engine
of and be able to
which allow him to
develops and mainta

The answer to
is straightforward.
production of reli
cannot afford to a
subject area which
our intellectual an

REFERENCES

1. Boehm, B.W..
'Practical Strate
Systems'. ed. Horo

2. Lehman, M.M.
of Software Evolut

writeability - programs are read more often than they are written, documentation should be clear, concise, and complete, and program and data structuring techniques should be used so that the software is built as a loosely interacting system of independent components rather than a closely interlinked monolithic system.

Notice that a software characteristic which is not an explicit feature of well-engineered software is efficiency. This does not mean that efficiency is unimportant - in some cases efficiency in terms of operational execution speed or the size of the object software system is of paramount importance. However, this is not universally true and, as hardware becomes cheaper and single chips become more complex and powerful, software efficiency considerations are likely to become less and less important in the majority of software systems.

In those cases where software efficiency must take precedence over considerations such as modifiability, this ought to be clearly stated in the software specifications. The efficiency specifications are of particular importance to the microprocessor software engineer who must fit his system into as few chips as possible or who may have to construct a control system with a given response time. It is his responsibility to ensure that the software specifications are clear on this point and that the trade-off between efficiency and other software characteristics is made clear to the software customer.

To conclude, the production of well-engineered, professional software must be based on an appreciation of the real costs of software systems, an understanding of user problems, and on the individual professional integrity of the software engineer. The software engineer must be aware of and be able to apply developments in software technology which allow him to improve the quality of the systems he develops and maintains.

The answer to the question 'why software engineering?' is straightforward. Future economic progress depends on the production of reliable, maintainable software systems. We cannot afford to adopt any less professional approach to a subject area which will consume an increasing proportion of our intellectual and economic resources.

REFERENCES

1. Boehm, B.W.. 1975. The High Cost of Software. In 'Practical Strategies for Developing Large Software Systems'. ed. Horowitz, E, Reading, Mass. : Addison Wesley.

2. Lehman, M.M. 1980. Programs, Life Cycles and the Laws of Software Evolution. Proc. IEEE. 68(9), 1060-1076