SOFTWARE TESTING AS AN AID TO QUALITY SOFTWARE PRODUCTION

Alfred Vella[1], Hazel Curtis[2], Diana Burkhardt[1] and Mark Broadbent[2]

[1]Department of Computing and Information Systems, The University of Luton, UK
[2] Allstor Software Limited, Whiting Way, Melbourn, Herts, UK

Abstract

The software industry has been in crisis almost since it began. This crisis reflects the wide gap between the expectations of users and commissioners of software products and what the software producer actually delivers. This gap can be expressed in terms of timescales, costs and functionality, as well as in terms of reliability.

In this paper we take a close look at the ways in which software can be tested in order to keep the gap between the user and the producer as small as is possible.

Such a close look reveals that new forms of testing could be developed so that the impact of the crisis can be reduced to a more bearable level.

1 INTRODUCTION

The software development process is notoriously difficult to control. Every week the computing press bemoans another long list of software projects that have failed to meet the expectations of their customers. Why should this be the case, given the enormous efforts that have been made to control software development?

The answer seems to lie in the very nature of software development. In this paper we model the process as a sequence of steps, each of which involves the translation of a 'representation' of the required system into another representation. The translations may be performed by humans, in which case they may be flawed, or they may be performed by the computer itself, in which case they are less likely to be flawed.

In this paper we develop our model as an aid to software testing. It helps us to think about the sorts of testing that can be undertaken. As we shall see, the representation shows us that many opportunities exist for generalising the testing process.

Software development, like any design process has three components: Goals, constraints and criteria. We need to understand these before we can look at the process and how it may be improved.

For a detailed look at the process of software development from a more traditional point of view see [Pressman 1997].

1.1 Goals

As in more traditional design tasks, the goals of a software development project are often ill defined. The client, customer, user, call them what you will, often does not know what they want. The design is a truly creative partnership between the customer and the producer. Unfortunately, customers and sometimes producers seem to forget this, and the project turns sour as each blames the other for escalating costs and missed deadlines.

1.2 Constraints

The constraints on a software project consist of the abilities of the producers, the capabilities of the hardware and software to be used, and the constraints imposed by customers. These

latter are often in the form of hardware/software utilisation and costs and performance requirements.

Strictly speaking the goals above could be taken as the overall goals and the full requirements, written and unwritten, taken as constraints. Constraints are absolute requirements: if a constraint is broken, the system fails to meet its requirements.

1.3 Criteria

Some criteria for success and factors which make one system better than another are easy to quantify. If we need a fast system, measurable improvements in performance would be a possible criterion. We may need to run the system on the cheapest hardware possible and again, this may be quantifiable.

Some issues are much less quantifiable. How does one measure the usability of a system? Many attempts have been made but as yet none are especially good.

Criteria are used to compare systems that already meet all of the constraints placed upon them. Criteria specify those desirable characteristics that we would like to be achieved by the system, but which are not essential. Most often they take the form of some measurement being better if it is larger or smaller.

2 SOFTWARE QUALITY

As with quality in general, the quality of software is surprisingly difficult to determine. Quality is variously defined for example as ‘fitness of purpose’ or ‘conformance to requirements’. However these both suffer from a tendency to be easier to use when finding fault with software once something is known to be wrong, rather than a tool for improving ‘quality’ as a part of the development process.

Much of the difficulty in software production stems from our inability to define, up front, the purpose (as in ‘fitness of purpose’) and the requirements (as in ‘conformance to requirements’). Often the user does not know the requirements and purpose fully, so how can one expect a process which starts from such an uncertain position to produce acceptable results?

A related problem is that defining measures of quality that can be used during the development process is still an open issue. If we could do this, we might be able to keep the process on track. In manufacturing quality control we need some measurement on the *product* that varies smoothly and that we can check to be within acceptable limits. As soon as the limits are breached we say that the *process* is out of control and take immediate action to rectify the situation. We are far from this state of affairs in software quality.

A number of different ‘programming styles’ can be observed, starting from the traditional ‘procedural’ style of programming to the more modern ‘event driven’ and ‘object oriented’ styles. An excellent introduction to comparative programming languages is contained in [Wilson 1993].

One of the easiest styles for non-programmers to understand is the procedural style. This is after all used for sets of instructions in DIY manuals, for recipes and for knitting patterns.

The declarative style is harder to explain. In it we state what is required and not how it is to be achieved. This style has become popular with academics trying to teach novice programmers how to think about good programming. Languages such as Prolog, Hope and ML use this style.

Prolog requires its programmers to declare properties of a number of predicates which together enable the computer to solve the problem. The actual, 'how' part of the program is determined by the system itself using mathematically sound reasoning procedures. Prolog is thus called a logic programming language.

In contrast to this ML and Hope require the programmer to define a set of functions that together have the required behaviour. By using pattern matching and function evaluation the system produces the required results.

Object oriented programming (OOP) was made popular by Xerox's Smalltalk language. When it first arrived, personal computers did not have the power to run it effectively and so it did not become as popular as it should have. The main idea in OOP is to separate objects from each other and include with them not only their properties but also the code that makes their actions (or methods). This separation is said to aid reusability of code. Procedural languages such as Pascal and C have been modified to give them OOP-like capabilities and this has resulted in languages such as C++, an object oriented language that is more often than not used like a procedural language. Using an OOP type of development encourages cohesive programs with little coupling between different parts. This in turn aids the maintainability of the system.

The event driven programming technique requires a different style of development, although it is often associated with OOP. In it we think of events (such as the mouse moving over an object for example) and actions that objects take in response to the events. Thinking of events in isolation in this way limits the amount of information that the programmer needs to recall at one time and this results in more reliable programming.

Besides these programming styles we also have to contend with 'newer' computing devices where the program, its data and its mechanisms are intertwined. Specifically here we are thinking of the increasing use of Expert Systems, Artificial Neural Networks and Genetic Algorithms. At present these are very hard to test satisfactorily. A detailed look at some of the issues involved in programming the first two of these is contained in [Giarratano 1989].

Expert Systems provide an interesting contrast with object oriented programming. In Expert System development using rule based programming, for example, we separate the program control from the data or knowledge as it is called in this case. The 'rule base' (or 'knowledge base') contains the data whilst a standard 'inference engine' determines all of the control. This is diametrically opposed to the OOP method where, as we have seen, control and data are collected together for each object.

Artificial Neural Networks (ANNs) represent one of the most successful development techniques that use machine learning rather than programming. In it we start with a network of neurons and go through a training process that should result in an ANN that can match inputs with their appropriate outputs. Not only this but one hopes that if the system is given an input that it has not yet seen, it will produce a sensible output. This form of 'programming', which relies on the system 'generalising' to inputs it has not seen, presents many novel problems for the software tester.

Our final class of programming is represented by the genetic algorithm, genetic program and evolutionary programming styles. Here an initial population of candidate 'programs' is randomly generated. After this, new programs are produced by a process analogous to evolution and natural selection. The fittest of the resulting programs survive to the next generation.

Unlike ANNs the resulting ‘best’ program can be tested using fairly conventional testing techniques.

This brief tour through the styles and types of programming currently in use shows the diversity of the problems that face the software tester. Each has its own particular problems but they all have in common the ability to surprise the tester with unexpected results.

3 THE DEVELOPMENT OF SOFTWARE

There are many competing methodologies for software development and models abound. However for our purposes the actual process can easily be modelled using a simple pictorial model and notation. Whatever the process involved we can look at each step as one of communication, transformation or translation between two entities. A representation of this is shown in figure 1.

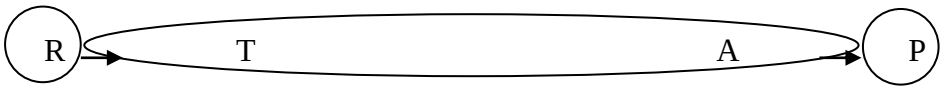

Figure 1 The basic process

In this figure, we introduce the following notation. We use a circle to represent the state of the process at the beginning of the step and another for the state at the end of the step. An oval is used for the process. Our notation uses the letter R to represent the system before the step. We say that R stands for the requirements of the step. We use the letter P for the system after the step and say that P stands for the resulting program. Letter T in the oval represents the Transformation involved in moving from requirements to program. The letter A also in the oval represents the Actor, that entity which performs the translation. The arrows in the diagram point from the start to the end of the step. As we shall see later, the Actor could be a human or the computer itself. The notation enables us to remember that the input to a step in some way represents the requirements for the translation. The output is in some sense a program produced at this step and the requirements needed for the next step. Clearly, in the traditional view only the input to the whole process is thought of as requirements and the output of the last step thought of as a program.

If however we use an analogy with the way that the word ‘customer’ is used in a quality context, we see that this generalisation of customer to include both internal and external customers has many benefits. Similarly we will see that the extension of the concepts of requirements and program to be the start and end points of translations enables us to think about testing in a more powerful way.

Using this notation we can model the whole process as just one step! We can refine the model into one containing an arbitrary number of steps when we wish.

The simplest example of this entails viewing the whole software development process as one step. Often a customer, with the help of computing professionals, will produce a specification of the required system: this is then developed into a working program. It is easy to see how Figure 1 represents this.

Less trivial examples involve the translation of an idea in the customer’s head into a specification, the translation of a specification into a data flow diagram (DFD), and finally the translation of the DFD into a program in a target programming language such as C++ or Delphi.

Less obvious, but none the less useful, examples include the translation of a program by an interpreter into actions and the translation of a program into machine code by a compiler.

In general development of a piece of software will involve a long sequence of translations and can be represented by a diagram similar to figure 2. We also note that often (at least initially) there is no 'specification', just an idea in someone's head.

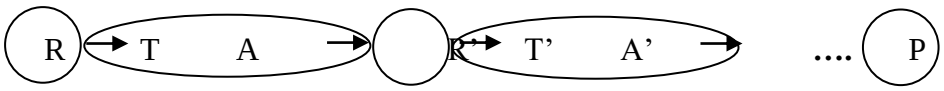

Figure 2 The software development process as many steps

For the purposes of this paper we thus look at the software development process as a sequence of translations between one representation of the system and another. We do not assume that the translation is faithful for a number of reasons, some good and some bad.

On the positive side, as a software project develops so does the specification, implicit and explicit. Allowing this to happen is an important feature of any mature development process. Only naive process models exclude negotiated changes to specification.

On the negative side, no matter how carefully the specification is written and the translation process is attempted errors do arise. A good process will attempt to discover such errors at the earliest opportunity. This is the main objective of testing.

In fact this area is the key to our problems. If, as we do here, one looks at software development as a sequence of translations starting with an idea in someone's head and ending with an artefact commonly called a computer program, the problems stem from faulty translation along the way.

It seems that humans and computers find different languages easier to handle. The development of formal methods was an attempt to get humans to use mathematical languages to specify programs. Once a specification in these languages was produced, checking any derived program was easier than for those specified using less formal systems. Unfortunately formal methods suffer from two major problems. Firstly humans do not like programming in these languages and so they have never caught on. Secondly the effort required to write a program correctly using such methods is of the same order as that required to write the same program correctly in a less formal language. Thus little if any benefit was delivered by these methods.

Some faults are introduced during translation from one human's representation to another's. People translating from human language to computer language introduce other errors; this is the normal 'programming step'. Further errors are possible when the computer itself does some translation between different computer understandable representations. However these are less likely than those introduced by human translators.

Before always blaming humans, one must, however remember the infamous hardware bug that occurred in early Pentium processors. Many thousands of these processors were sold and in use before someone noticed that under some very special circumstances erroneous results were obtained during calculations.

4 TESTING SOFTWARE

Given the notation that we introduced earlier, we can describe testing as taking two nodes, not necessarily adjacent, and comparing them. Testing should satisfy us that the one representing

the system further down stream from the other is ‘better’ in some sense. The word better here is in quotes, as we need to have either clarified the requirements or agreed a variation with our customers. It is also expected that the better system is represented in a language nearer to that of the target computer.

Over the years many methods of software testing have been developed. One popular method of classifying testing methods is that of black vs white box methods. An excellent source on these is contained in [Marick 1995].

In black box methods we use only knowledge of what is to be done. Thus these methods use declarative knowledge. We think of the transformation as described as a single arc in figure 1. Thus with black box testing we may only make use of properties of the overall transformation.

With white box testing, on the other hand, during testing we also make use of the way in which the actions are to be done. We can then use intermediate results as part of our tests. We can also analyse more carefully the areas where errors are likely to occur and intensify our testing towards these points. Thus, for example, if we know that some special inputs require handling in different way than other inputs we should ensure that our tests cover all cases.

For the following we would like to distinguish between forward and backward testing strategies. In forward testing we compare the results of tests in the forward direction from specification to program. This will be illustrated using output analysis, independent development and complexity as our tools. We could however make a translation in the reverse direction, obtaining a system that is nearer human language, and compare results. We shall see how this works when we look at reverse engineering.

4.1 Output analysis

The basic idea here is to look at output produced by the program and compare it with expected results. In general if we do have a good way of producing the required output given a known input then we should produce three sets of these input output pairs, before the program is written. The first set can be used during program development so that the implementers have some independent check on their work. The producer uses a second set just before the system is handed over to convince them that the system is working to specification. A final set could be used by the customer to check for themselves that the system meets the specification.

Analysing a program’s output by comparing it with expected results presupposes that we have a firm view of what those results actually are! This often presents us with a major problem, how do we get the expected results?

The answer is of course that we cannot always do this! However we may be able to find a property of the output that is easy to calculate. This idea is similar to the use of parity and check digits for memory and bar code checking.

These processes can be represented as in figure 3.

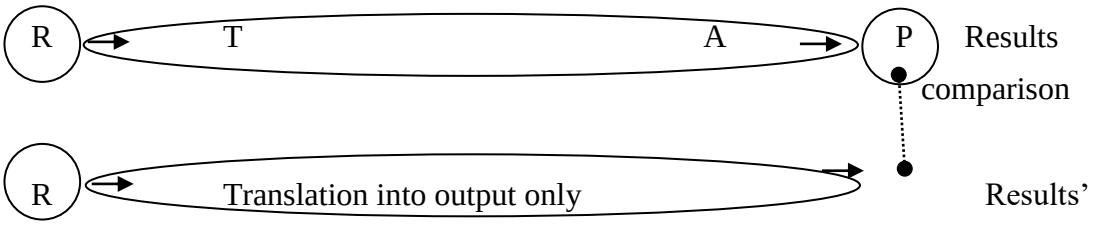

Figure 3 Output analysis

4.2 Complexity

Two types of complexity are looked at below. They are time and space complexity. Both of these require us to monitor the program's use of resources as the program's parameters are varied.

A simple example is that of sort routine. If we have implemented a sort strategy that we can show has complexity of *o(nlogn)* say, but when we plot a graph of time vs number of sorted items we do not get the right picture, we should then investigate what is going wrong. Similarly if we monitor memory usage as a function of *n* and we get anomalous behaviour we can try to explain this discrepancy or look for a bug.

4.3 Other forward testing techniques

One development method used for safety critical systems is that of independent development. This is illustrated in figure 4.

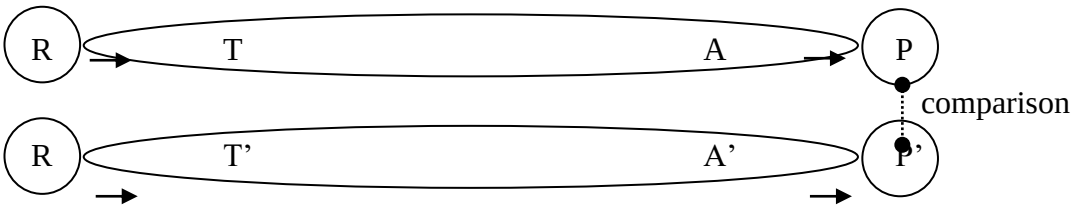

Figure 4 Independent development

In independent development we produce two versions of the system using two independent teams working from the same specification. By comparing the resulting systems we hope to obtain either a better system than either taken individually, or some confidence in the resulting system should the two turn out to be functionally identical.

The diagram clearly shows the similarity between this case and that of output analysis. In a sense the two are very close, they differ only in that with independent development the process of finding the required output is of the same order of difficulty as the original programming task.

It is worth noting here that independent development has a number of uses other than in safety critical systems. For example there are often many ways of developing a software system. The use of rapid prototyping can significantly reduce development time though the performance of the resulting system may be poor. If rapid prototyping is used for one of our independent development processes we may obtain a system that does not meet constraints such as speed

although its output may be ‘correct’. We can use this output to test that produced by a more costly development process that has met constraints failed by the rapid prototyping.

Similarly a constraint may be that the system should run on such and such a processor using so much memory, hard disk space etc. If we can cheaply produce a functionally similar system which fails many constraints, although we cannot use the resulting system as our final product, we can use it to test other more compliant systems.

4.4 Reverse engineering

The idea behind reverse engineering is that, for humans at least, programs are harder to understand than their specification. If we could take the output of one of the steps and produce an approximation of what the input to the step (or a previous step) was then a human (or possibly a computer) could compare the two.

This is illustrated in figure 5 where we show a single step being reverse engineered.

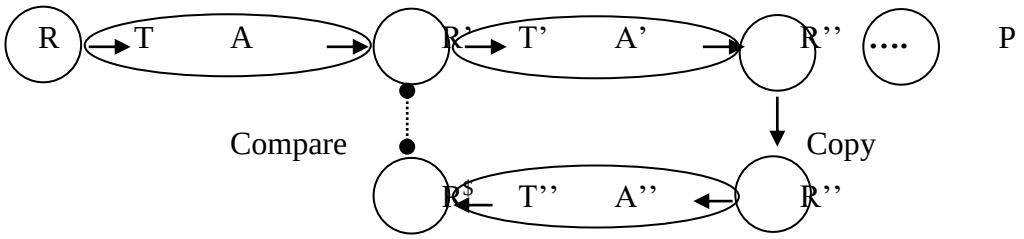

Figure 5 Reverse engineering a single step

It should be stressed that $R^{\$}$ and R’ need not be in the same language, but they must be comparable. So for example $R^{\$}$ could be a diagrammatic representation of R’’ (a data flow diagram for example) and one might be able to see if it is possible that this data flow diagram could represent the system. If not, we know that we have a problem.

We could also reverse engineer a number of steps rather than just one, as shown in the diagram. We give reverse engineering a much closer look in [Curtis 1998].

5 AUTOMATING TEST SET GENERATION

As we argued in the section on output analysis the generation of test sets can be one of the most difficult aspects of testing. In this section we look briefly at the possibility that the generation of such sets being automated or at least aided by the computer. As before, we can look at both forward and backward generation.

5.1 Risk analysis

The more knowledge of our processes we have, the more effective the test sets can become. Thus past experiences, possibly from databases of past mistakes that have been made, can be used to assign risks to each aspect of our transformation and these risks can then be used to allocate numbers of test cases to testing different aspects of the transformation.

For example, we know that *C* programmers often have difficulty in avoiding using memory just beyond the end of an array. This is caused by the language feature that defines the last element of an array with twelve elements (say) as having index 11! As we know that this is an insecurity in *C*, we can look out for it.

Risk analysis can only be used to its fullest if statistics on errors found are religiously kept. Unfortunately because of the nature of the software development process and the touchiness of programmers (or their bosses) much useful data is often discarded (or disguised).

5.2 Forward generation

The aim of forward generation is to use knowledge gained from the requirements to design effective test sets. If we have a reasonably tight specification of the requirements of the current step and an idea of weaknesses in the process of the step, we might well be able to look for tests focused upon these weaknesses. For example in white box testing we know that errors occur at the boundaries where one feature in the specification meets another or where there is a natural boundary caused by the properties of the program that we are writing. For example if we are writing a program that solves equations, we would naturally look for conditions where there were no, one or many solutions. Mathematics would in these cases tell us where the boundaries between these cases are and we would be foolish not to test our transformation at these boundaries.

5.3 Backward generation

Because the results of a step tend to be in a more formal language that the requirements of the same steps the backward generation of test cases is much easier to automate than forward generation. The basic idea here is to look at the output of the step and try to find inputs to the program that cause it problems, or at least exercise it fully.

For example if we have the following Java fragment which we have adapted (word for word but with slight changes in formatting) from [Ince 1997]. It forms the following exercise:

‘Exercise

A class UserTable has two instance variables: an integer variable noInTable which contains the number of items in a table and an integer array arrTable which represents the table. Write down a method which checks the equality of two objects defined by UserTable. Assume that two arrays are equal if they contain the same number of elements in the same order.

```
Public boolean equals(UserTable u) {
Int j = 0, count = 0;
this.arrTable[noInTable] = 1; u.arrTable[noInTable] = 2;
while (this.arrTable[j] == u.arrTable[j]) j++
return (this.noInTable == u.noInTable && j = u.noInTable);
}
‘
```

An automatic test generator would be able to report that:

a) *noInTable* must be within bounds for both *this.arrTable* and *u.arrTable*

b) both *this.arrTable[noInTable]* and *u.arrTable[noInTable]* should not contain useful information before entry.

And possibly

c) both *this.arrTable[j]* and *u.arrTable[j]* should be legal for all values of *j* from 0 to *noInTable*

From the specification of the fragment a human might realise that b) above cannot be taken for granted and so a potential bug has been found. How it is dealt with thereafter would depend upon the details of the required system.

Obviously much more can be said about this code fragment depending upon the system's knowledge of Java. If some idea of the purpose of the step were also known to the system it might be able to use this knowledge to analyse the fragment further.

6 CONCLUSIONS

In this paper we have introduced a notation for describing the software development process as a sequence of translation steps from an idea in someone's head to program code or even machine code in the computer. Looking at the process in such a simple way enables us to treat software testing in a similarly simple fashion. We saw how traditional testing can be thought of as a forward testing process and how reverse engineering is just as simple to represent.

Fully automating the testing process is still some way off but techniques are being developed which will aid the human tester in their work.

REFERENCES

Curtis, H., Vella, A., Burkhardt, D. and Broadbent, M. [1998] Automated test suites from reverse engineering and Planguage.

Giarratano, J. and Riley, G. [1989] Expert Systems, principles and programming. PWS Kent.

Ince, D. and Freeman, A. [1997] Programming the Internet with Java. (p164) Addison Wesley.

Marick, B. [1995] The craft of software testing. Prentice hall. 0-13-177411-5.

Pressman, R. S. [1997] Software Engineering. McGraw Hill.

Wilson, L. B. and Clarke, R. G. [1993] Comparative programming languages. Addison-Wesley.