Some comments on development testing

Les Hatton
Kingston University

January 5, 2012

Abstract

In line with the special theme of this edition, development testing is taken to mean testing done during the coding process. There follow some general comments on this process and the opportunities it presents.

0.1 Background

I feel I must start this article by repeating my mantra of the last twenty-five years. Design, development and testing are emphatically *not* phases of a project. They are states of mind which co-exist throughout a project. This merely reflects the essentially iterative nature of software development. I don't think I have ever embarked on a software project where I knew what I was doing at the beginning, (and in fortunately few but memorable cases, at the end either). Instead, requirements capture, implementation and testing must be woven together at each stage of the project. In the early days, requirements dominate but in all but the most trivial of projects, this will necessarily be accompanied by mini-prototypes to test feasibility and testability. In the later stages of the project, this balance will change to favour active testing and re-development.

In spite of this, we do seem to feel a need to compartmentalise software processes. This may simply be an offshoot of the bureaucratic urge to create mini-empires and I have certainly seen enough of these in my career to see me out. Separating software processes in this way is very damaging to the intellectual coherence of a software project and has had numerous side-effects in my view, for example,

- Testing is still often considered a low-level and in particular, a low-status activity.
- Implementation, (i.e. development) is considered almost a detail by managers who, after writing a few Excel macros think that the act of coding is just like writing 'a big macro'.
- Testers are still rarely involved in the design stage of a project. I co-edit the *Software Impact* column in *IEEE Software* with Michiel van Genuchten and we recently interviewed all the contributors so far to solicit comments about the design process. These are really major systems with millions of lines of code and many users. All the contributors stated that testers should be involved at this stage but none were. This seems astonishing to me but is the norm.
- The design process is considered a high status activity but in truth, we have almost no idea how to do it properly and there is precious little standardisation, (another thing which has emerged from our *IEEE Software* column).
- The influence of bureaucracy and the intrusion of management into software skills definition has meant that programming skills are in significant decline, (I make this observation partly as a part-time academic and partly through my involvement with commercial projects). For example, it is now possible to graduate with a degree in Computing with almost no programming experience whatsoever.

I didn't mean this article to be polemic so I will now focus on what we might do, with some examples.

0.2 Design IS Development IS Testing

Having caught your eye with a suitably catchy heading, let me put a little flesh on this. Consider the following computer program.

$$A1 = (S1 \cap S2 \cap \neg S3) \cup (S1 \cap S2 \cap S3) \quad (1)$$

$$A2 = (\neg S1 \cap \neg S2 \cap S3) \cup (S1 \cap \neg S2 \cap S3) \cup (\neg S1 \cap S2 \cap S3) \cup (S1 \cap S2 \cap S3) \quad (2)$$

$$A3 = (\neg S1 \cap S2 \cap \neg S3) \cup (S1 \cap \neg S2 \cap \neg S3) \cup (\neg S1 \cap \neg S2 \cap S3) \cup (S1 \cap \neg S2 \cap S3) \quad (3)$$

$$A4 = (\neg S1 \cap S2 \cap \neg S3) \cup (\neg S1 \cap S2 \cap S3) \quad (4)$$

It might not look it, but it is actually the specification for the treatment of *errno* in the ISO C programming language, the error number returned by the C run-time library. A1-A4 are the actions to be taken, S1-S3 are the three independent binary states which must be considered and $\cap, \cup$ are the logical operators AND and OR. You don't need to know what the states and actions are (and probably don't want to) but the important point is that this is a design specification written in the predicate calculus, more or less directly from the fairly dense verbal description of this in the ISO C standard itself[1]

However, it is also a program which can be translated into C as:-

```
if ( (S1 && S2 && !S3) || (S1 && S2 && S3) )
{
     A1;
}

...
```

(I've only done equation(1) here but the rest follow in an obvious way.) However, last but not least, it also shows the 12 test cases, which are:-

- Test case 1: (S1 true, S2 true, S3 false): results in A1.
- Test case 2: (S1 true, S2 true, S3 true): results in A1.
- ...

By using this specification language, it can be seen therefore that design, implementation and testing are all intimately inter-linked, so inter-linked in fact that separating them is artificial and in my view damaging to the clarity of the development. It may be that you are not familiar with specification languages, in which case you can use Warnier-Orr diagrams which are a graphical equivalent of this process as shown in Figure 1, (although they can't be refactored logically in any trivial way as I do below with the predicate calculus version.).

[1] I was going to use an airbag example but I didn't want to frighten anybody.

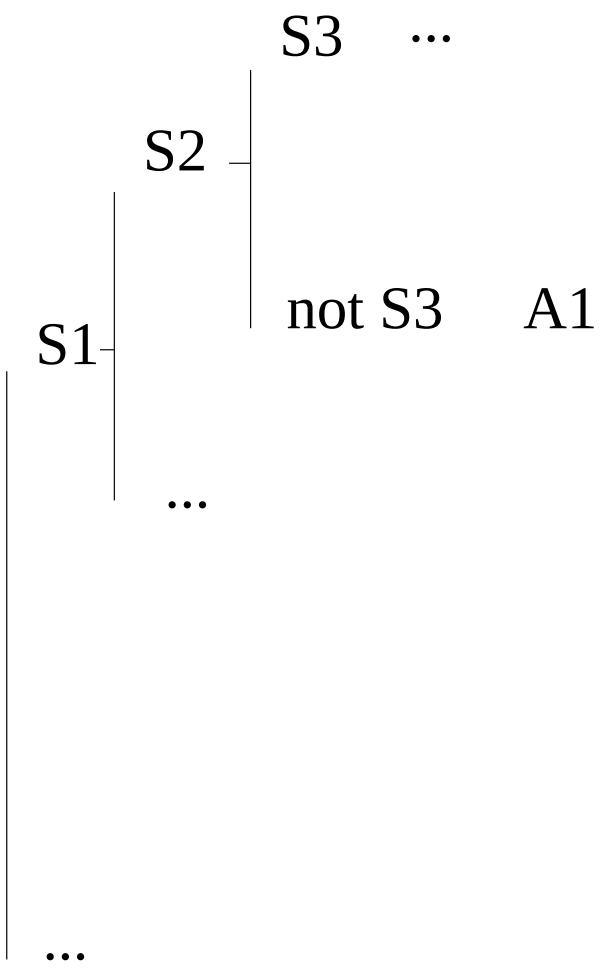

Figure 1: A Warnier-Orr diagram equivalent to the specification equation (1).

If as in this case, I am using a specification language with formal mathematical properties (the laws of predicate calculus). I can use these laws to simplify this design specification, whilst simultaneously simplifying the code and the testing. The design specification (1) - (4) then turns out to be identical to:-

$$A1 = (S1 \cap S2) \tag{5}$$

$$A2 = S3 \tag{6}$$

$$A3 = \neg S2 \tag{7}$$

$$A4 = (\neg S1 \cap S2) \tag{8}$$

This radically simpler but identical program has only 4 test cases.

You might well ask if this is the way I develop all programs. The answer is no, but you need some kind of similar representation when things are getting logically difficult, however the fact that I might carry the specification in my head informally whilst developing the code does not mean that I should forget about the test cases. As a developer, I should be developing the test cases in parallel so that the package I would pass onto an independent testing team is the code *and* the test cases, (and maybe the formal design specification as well, if I had to develop one). The independent test team can then judge the package as a whole.

I would go further. The independent test team should be able to swap places with the developers.

0.3 Development testing and honesty

One of the most important traits a developer should acquire is intellectual honesty. After years of developing myself and talking to many other developers, I know that experienced developers sense when they are in trouble. They know when their code is not coherent and they know when they have used a clumsy set of data structures for a particular algorithm. The question is what do you do about it ?

Over thirty years ago at Bell Labs, a paper on code rewriting appeared based on the well-known Unix diff (differential file comparator). The authors (McIlroy and Hunt) described the evolution of this program. In particular, they actually developed it three times, but they were only intellectually satisfied with the third version. In essence, the first two versions were good working prototypes but in each case, the authors felt they could do better. They did. The third version was astonishingly reliable and half the size of the first version. However, in order to be able to see this, they had to take the earlier versions through much of the normal software process, linking the various phases. Contemporaneously, Fred Brooks was writing in the eponymous *Mythical Man Month* that we should plan to throw (at least) one away, because we will anyway, and the third version was usually the best.

We seem to have forgotten much of this pioneering work. If managers allowed developers to prototype more and developers understood the need for it rather better than they do, we would have made a lot more progress. Successful prototyping is at the heart of development testing. It is at the heart of all successful software systems. It is really at the heart of all successful human creative achievement as I was reminded at the Leonardo da Vinci exhibition at the National Gallery in London in December 2011. Da Vinci endlessly prototyped parts of his great works, (such as the fall of cloth on an arm), until he could put them all together. Beethoven did the same with his symphonies.

0.4 Development testing and patterns

Coding itself is (at least for nerds like myself), a pleasurable process. I like making code look good and I like making code read transparently. Above all, I like thinking about how I would test it. I know I should I get out more, but its really for self-defence. I still maintain a lot of commercial and open source code (in C, Tcl, Perl and Fortran) and it rapidly gets out of hand if code is incoherent or rushed. Even throw-away code should be good because you never know when you will have to re-use it for more prototyping or to plunder a neat trick you have learned but temporarily forgotten the mind-bending syntax involved.

As time goes by, you begin to accumulate knowledge and patterns of failure. This knowledge can come from various sources. Here is an example in C:-

```
if ( a && b && c ... ) {
     ...
} else {
     /* There is about a 70% chance you will get this bit wrong */
}
```

This of course is an example of a complex decision with an *else* clause. However, we have known since the work of the psychologist T.R.G. Green that humans are very poor are reversing complex decisions. In other words we are more than likely to get the logical condition in which we enter the *else* clause above wrong.

Here is an example of one of my mistakes with this construct:-

```
 /*
 *      Check division operators for zero division.
 *      right_ok = TRUE, if the divisor has been defined correctly.
 */
        case OP_DIV:
        case OP_MOD:
                if ( right_ok  && (right_cval.cv_long == 0) )
                {
/*
 *                      Can't do the division, the divisor is zero.
 *                      Give a hard warning if the expression has to be
```

```
 *                      evaluated and a soft warning otherwise.
 */
                        ...
                }
                else
                {
/*
 *                      All OK, do the division with right_cval.cv_long.
 */
                        ...
                }
...
```

Is it OK to do the division ? Well actually no. I had forgotten the case when right_ok is not true but right_cval.cv_long IS zero. This emerged five months after releasing a product with a very aggressive set of regression tests, (not aggressive enough however).

0.5 Development testing and open source

Perhaps the most obvious place in which development testing has flourished is in the open source world from which we can learn many lessons because it is very common in open source projects for development to go hand in hand with testing. (It doesn't in all projects but it appears to do in all successful projects). Availability of source code means that any algorithm has its deepest secrets exposed to anybody who wants to look. Faults are flushed out and fixed perhaps before they have ever had the chance to fail. In very widely-used systems such as the Linux kernel, PHP, Perl and Apache, this has led to exceptionally reliable systems in a environment in which development and testing are inextricably inter-twined as they must be.

It is painfully difficult to impart much useful information in a couple of thousand words but hopefully these few comments will encourage readers to delve more.