Why Bother to Unit Test?

Executive Summary

This paper addresses a question often posed by developers who are new to the concept of thorough testing: Why bother to unit test? The question is answered by adopting the position of devil's advocate, presenting some of the common arguments made against unit testing, then proceeding to show how these arguments are worthless. The case for unit testing is supported by published data.

IPL is an independent software house founded in 1979 and based in Bath. IPL was accredited to ISO9001 in 1988, and gained TickIT accreditation in 1991. IPL has developed and supplies the AdaTEST and Cantata software verification products. AdaTEST and Cantata have been produced to these standards.

Copyright

This document is the copyright of IPL Information Processing Ltd. It may not be copied or distributed in any form, in whole or in part, without the prior written consent of IPL.

IPL
Eveleigh House
Grove Street
Bath
BA1 5LR
UK
Phone: +44 (0) 1225 444888
Fax: +44 (0) 1225 444400
email ipl@iplbath.com

Certificate Number FM 01589

Last Update:03/07/1997 08:32:00
File: WHY_UT.DOC

1. Introduction

The quality and reliability of software is often seen as the weak link in industry's attempts to develop new products and services.

The last decade has seen the issue of software quality and reliability addressed through a growing adoption of design methodologies and supporting CASE tools, to the extent that most software designers have had some training and experience in the use of formalised software design methods.

Unfortunately, the same cannot be said of software testing. Many developments applying such design methodologies are still failing to bring the quality and reliability of software under control. It is not unusual for 50% of software maintenance costs to be attributed to fixing bugs left by the initial software development; bugs which should have been eliminated by thorough and effective software testing.

This paper addresses a question often posed by developers who are new to the concept of thorough testing: Why bother to unit test? The question is answered by adopting the position of devil's advocate, presenting some of the common arguments made against unit testing, then proceeding to show how these arguments are worthless. The case for unit testing is supported by published data.

2. What is Unit Testing?

The unit test is the lowest level of testing performed during software development, where individual units of software are tested in isolation from other parts of a program.

In a conventional structured programming language, such as C, the unit to be tested is traditionally the function or sub-routine. In object oriented languages such as C++, the basic unit to be tested is the class. With Ada, developers have the choice of unit testing individual procedures and functions, or unit testing at the Ada package level. The principle of unit testing also extends to 4GL development, where the basic unit would typically be a menu or display.

Unit level testing is not just intended for one-off development use, to aid bug free coding. Unit tests have to be repeated whenever software is modified or used in a new environment. Consequently, all tests have to be maintained throughout the life of a software system.

Other activities which are often associated with unit testing are code reviews, static analysis and dynamic analysis. Static analysis investigates the textual source of software, looking for problems and gathering metrics without actually compiling or executing it. Dynamic analysis looks at the behaviour of software while it is executing, to provide information such as execution traces, timing profiles, and test coverage information.

3. Some Popular Misconceptions

Having established what unit testing is, we can now proceed to play the devil's advocate. In the following subsections, some of the common arguments made against unit testing are presented, together with reasoned cases showing how these arguments are worthless.

©IPL Information Processing Ltd

3.1. It Consumes Too Much Time

Once code has been written, developers are often keen to get on with integrating the software, so that they can see the actual system starting to work. Activities such as unit testing may be seen to get in the way of this apparent progress, delaying the time when the real fun of debugging the overall system can start.

What really happens with this approach to development is that real progress is traded for apparent progress. There is little point in having a system which "sort of" works, but happens to be full of bugs. In practice, such an approach to development will often result in software which will not even run. The net result is that a lot of time will be spent tracking down relatively simple bugs which are wholly contained within particular units. Individually, such bugs may be trivial, but collectively they result in an excessive period of time integrating the software to produce a system which is unlikely to be reliable when it enters use.

In practice, properly planned unit tests consume approximately as much effort as writing the actual code. Once completed, many bugs will have been corrected and developers can proceed to a much more efficient integration, knowing that they have reliable components to begin with. Real progress has been made, so properly planned unit testing is a much more efficient use of time. Uncontrolled rambling with a debugger consumes a lot more time for less benefit.

Tool support using tools such as AdaTEST and Cantata can make unit testing more efficient and effective, but is not essential. Unit testing is a worthwhile activity even without tool support.

3.2. It Only Proves That the Code Does What the Code Does

This is a common complaint of developers who jump straight into writing code, without first writing a specification for the unit. Having written the code and confronted with the task of testing it, they read the code to find out what it actually does and base their tests upon the code they have written. Of course they will prove nothing. All that such a test will show is that the compiler works. Yes, they will catch the (hopefully) rare compiler bug; but they could be achieving so much more.

If they had first written a specification, then tests could be based upon the specification. The code could then be tested against its specification, not against itself. Such a test will continue to catch compiler bugs. It will also find a lot more coding errors and even some errors in the specification. Better specifications enable better testing, and the corollary is that better testing requires better specifications.

In practice, there will be situations where a developer is faced with the thankless task of testing a unit given only the code for the unit and no specification. How can you do more than just find compiler bugs? The first step is to understand what the unit is supposed to do - not what it actually does. In effect, reverse engineer an outline specification. The main input to this process is to read the code and the comments, for the unit, and the units which call it or it calls. This can be supported by drawing flowgraphs, either by hand or using a tool. The outline specification can then be

©IPL Information Processing Ltd

reviewed, to make sure that there are no fundamental flaws in the unit, and then used to design unit tests, with minimal further reference to the code.

3.3. “I'm too Good a Programmer to Need Unit Tests”

There is at least one developer in every organisation who is so good at programming that their software always works first time and consequently does not need to be tested. How often have you heard this excuse?

In the real world, everyone makes mistakes. Even if a developer can muddle through with this attitude for a few simple programs, real software systems are much more complex. Real software systems do not have a hope of working without extensive testing and consequent bug fixing.

Coding is not a one pass process. In the real world software has to be maintained to reflect changes in operational requirements and fix bugs left by the original development. Do you want to be dependent upon the original author to make these changes? The chances are that the “expert” programmer who hacked out the original code without testing it will have moved on to hacking out code elsewhere. With a repeatable unit test the developer making changes will be able to check that there are no undesirable side effects.

3.4. Integration Tests will Catch all the Bugs Anyway

We have already addressed this argument in part as a side issue from some of the preceding discussion. The reason why this will not work is that larger integrations of code are more complex. If units have not been tested first, a developer could easily spend a lot of time just getting the software to run, without actually executing any test cases.

Once the software is running, the developer is then faced with the problem of thoroughly testing each unit within the overall complexity of the software. It can be quite difficult to even create a situation where a unit is called, let alone thoroughly exercised once it is called. Thorough testing of unit level functionality during integration is much more complex than testing units in isolation.

The consequence is that testing will not be as thorough as it should be. Gaps will be left and bugs will slip through.

To create an analogy, try cleaning a fully assembled food processor! No matter how much water and detergent is sprayed around, little scraps of food will remain stuck in awkward corners, only to go rotten and surface in a later recipe. On the other hand, if it is disassembled, the awkward corners either disappear or become much more accessible, and each part can be cleaned without too much trouble.

3.5. It is not Cost Effective

The level of testing appropriate to a particular organisation and software application depends on the potential consequences of undetected bugs. Such consequences can range from a minor inconvenience of having to find a work-round for a bug to multiple deaths. Often overlooked by software developers (but not by customers), is the long term damage to the credibility of an organisation which delivers software to users with bugs

©IPL Information Processing Ltd

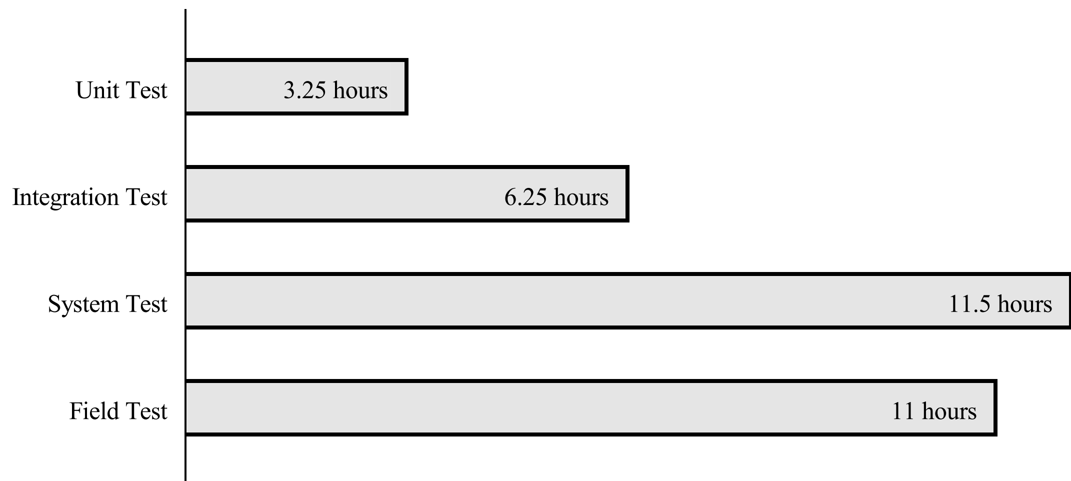
in it, and the resulting negative impact on future business. Conversely, a reputation for reliable software will help an organisation to obtain future business.

Many studies have shown that efficiency and quality are best served by testing software as early in the life cycle as practical, with full regression testing whenever changes are made. The later a bug is found, the higher the cost of fixing it, so it is sound economics to identify and fix bugs as early as possible. Unit testing is an opportunity to catch bugs early, before the cost of correction escalates too far.

Unit tests are simpler to create, easier to maintain and more convenient to repeat than later stages of testing. When all costs are considered, unit tests are cheap compared to the alternative of complex and drawn out integration testing, or unreliable software.

4. Some Figures

Figures from “Applied Software Measurement”, (Capers Jones, McGraw-Hill 1991), for the time taken to prepare tests, execute tests, and fix defects (normalised to one function point), show that unit testing is about twice as cost effective as integration testing and more than three times as cost effective as system testing (see bar chart).



(The term “field test” refers to any tests made in the field, once the software has entered use.)

This does not mean that developers should not perform the latter stages of testing, they are still necessary. What it does mean is that the expense of later stages of testing can be reduced by eliminating as many bugs as possible as early as possible.

Other figures show that up to 50% of maintenance effort is spent fixing bugs which have always been there. This effort could be saved if the bugs were eliminated during development. When it is considered that software maintenance costs can be many times the initial development cost, a potential saving of 50% on software maintenance can make a sizeable impact on overall lifecycle costs.

5. Conclusion

Experience has shown that a conscientious approach to unit testing will detect many bugs at a stage of the software development where they can be corrected economically. In later stages of software development, detection and correction of bugs is much more difficult, time consuming and costly. Efficiency and quality are best served by testing

©IPL Information Processing Ltd

software as early in the lifecycle as practical, with full regression testing whenever changes are made.

Given units which have been tested, the integration process is greatly simplified. Developers will be able to concentrate upon the interactions between units and the overall functionality without being swamped by lots of little bugs within the units.

The effectiveness of testing effort can be maximised by selection of a testing strategy which includes thorough unit testing, good management of the testing process, and appropriate use of tools such as AdaTEST or Cantata to support the testing process. The result will be more reliable software at a lower development cost, and there will be further benefits in simplified maintenance and reduced lifecycle costs. Effective unit testing is all part of developing an overall "quality" culture, which can only be beneficial to a software developers business.

©IPL Information Processing Ltd