White Paper

Why Bother to Unit Test?

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, by presenting and examining some of the common arguments made against unit testing.

Contents

1 Introduction 3

2 What is Unit Testing? 3

3 Some Popular Misconceptions 3

3.1 It takes too much time / costs too much 4

3.2 It is slower than de-bugging the code 5

3.3 It only proves the code does what the code does 5

3.4 "I'm too good a programmer to need unit tests" 6

3.5 Integration tests will catch all the bugs anyway 6

3.6 The client pays to develop code, not write unit tests 7

3.7 I don't know how to write good unit tests 7

3.8 I am working with a legacy application without unit tests so I can't unit test 8

3.9 I am only making a small change, I can 'see' it is correct, so why bother to test? 9

3.10 I cannot unit test my code because it is not testable 9

3.11 I'm not doing Test Driven Development 9

3.12 Maintaining unit tests is pointless 10

3.13 Testing is a job for testers / QA 10

3.14 Interfaces are best tested during integration testing 11

3.15 Our competition don't unit test, so it would make us uncompetitive 11

4 What do I gain from unit testing? 12

4.1 Unit tests enable collective ownership 12

4.2 Unit tests enable safe refactoring 12

4.3 Problems are found early in the development cycle 12

4.4 Unit tests document the code 13

4.5 Unit tests demonstrate progress 13

4.6 Unit testing improves coupling 13

5 Conclusion 14

6 References 14

Copyright Notice

1 Introduction

The quality and reliability of software is often seen as the weak link in a company's attempt to develop new products and services. The last decade has seen the issue of software quality and reliability being 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 that should have been eliminated by thorough and effective software testing.

This paper addresses a question often posed by developers: 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 invalid.

2 What is Unit Testing?

*A **Unit** is the smallest testable part of an application. In procedural programming a unit may be an individual program, function, procedure, etc., while in object-oriented programming, the smallest unit is a class, which may belong to a base/super class, abstract class or derived/child class.* [1]

The term *unit testing* is interpreted very differently by different organizations, the confusion stems from the term *unit* being interchangeably used with *module*, *component, object* and other terms. A better term for unit testing would be developer testing. The terminology used does not really matter. What does matter is that it is the smallest practicably testable bit of code without the need for the rest of the application. That practicability may involve tightly coupled clusters of code or single items in isolation.

The essence of a unit test is that it checks that the unit, whatever its definition, meets its purpose. Unit tests are not intended to verify the high level system or business requirements. They verify that each unit which makes up the system behaves only as expected, according to low-level functional requirements of the application and meets other test objectives such as robustness testing or code coverage. Without confidence in the units, it is much harder to have confidence that the software as a whole will work.

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 popular arguments made against unit testing are presented, together with reasoned cases showing how these arguments are invalid.

3.1 It takes too much time / costs too much

This is the most common complaint made about unit testing. Producing unit tests is time-consuming and does slow down the code production phase, but that is not the whole picture. Working on the assumption that all code has bugs, (Code Complete [2] cites the industry average as 15-50 errors per 1000 lines of delivered code), the challenge is when to find them and keep the code as bug free as practicable.

The earlier in the development cycle that bugs can be identified and removed the cheaper it will be over the entire application development. After code review and static analysis, dynamic unit testing is the earliest and therefore most cost effective means available for doing this because:

- defects in objects are identified nearly immediately, so there is less re-work and re-test dependency on other code;
- testing as soon as code is written can be done with isolation unit testing, not waiting for a full system build;
- complex applications have just too many variables to thoroughly test in a practicable way, so breaking them down into manageable unit tests is cheaper to develop and maintain;
- Defensive programming and makes verifying this behavior very difficult and expensive to system test, but simple and cost effective during unit testing;
- complex applications will always be more difficult and expensive to integrate and system test when the component units do not work correctly, so testing the units reduces these downstream costs and makes them more predictable;
- unit tests pinpoint failures more precisely than larger system tests on complex applications;
- when run in a regression suite or under continuous integration, unit tests act as the most efficient and automated safety net against downstream changes breaking existing code.

The whole picture comprises not just the initial code production phase, but all the later stages of integration and verification. Over the lifecycle unit testing as a technique can bring overall development and maintenance costs down.

There are many ways to reduce the costs of unit testing. Using automation framework tools available will inevitably save money. A good unit testing tool will provide a common consistent easy to use suite minimizing test generation and maintenance effort while maximizing diagnostic analysis for the developer.

CANTATA

> Automated test script generation parsing the source code to derive a test framework, and multiple test vector generation options makes test creation faster

> Efficient graphical development of tests written in C/C++ code

> Works in developers IDE on host and target

> Automated regression tests

> Well supported COTS solution with a 20+ year pedigree of worldwide use

3.2 It is slower than de-bugging the code

This objection is based on a simple but widespread misconception, that de-bugging is the same as testing. The objective of a test is to verify that the code does what it should do correctly and nothing else. The objective of de-bugging code is to identify the root cause of and to remove a bug. The reason for a test failure may be immediately apparent, it may not and the code / test require de-bugging as a result.

While de-bugging can of course find and identify some bugs, it may not find an error of omission in code, or identify that the software while executing successfully does something not as expected. De-bugging can therefore be a faster activity than unit testing but it does have some significant downsides because it not a comparable activity but a complimentary one.

Unit testing is a structured activity with the defined objective of proving the code unit behaves only as expected. De-bugging with its different investigatory objective can often be a much less structured, grazing of the code.

More significantly, de-bugging is a manually intensive activity without automated generation support, and is not designed to be consistently repeated. Whereas re-execution of unit tests is highly if not completely automatic, with the same code and tests producing the same persistent test results.

CANTATA

> Tests are structured and repeatable & can also be run under a de-bugger.

3.3 It only proves the code does what the code does

This is a common complaint of developers who jump straight into writing code, without first designing the unit and understanding its contribution to the overall system. 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.

Any tests you write will only ever be as good as your knowledge of what the code should do. If you derive the tests only from the code then, yes, they will only prove that the code does what the code does. If the tests are based on something other than the code (a requirement, design, model or other knowledge of the expected functionality) then tests are abstracted from source code, and when supplemented by techniques such as error guess, error seeding etc. are of much greater benefit.

However, a test that verifies that the code does what it does is not necessarily useless. It is of little benefit on its own, but can be very useful to help ensure that future modifications do not break existing correct behavior. Code changes happen both during production and maintenance phases, so having that safety-net or baseline of unit tests as part of a regression test strategy can pinpoint failures which may otherwise be hard to identify.

CANTATA

> Tests can be tagged to requirements, design or test specs, for traceability to the definition of what the code should do.

> AutoTest creates complete passing unit test scripts from C source code.

3.4 "I'm too good a programmer to need unit tests"

There is at least one developer in every organization who is seen as a guru, and is perceived to be so good at programming that their software will always work first time, and consequently does not need to be tested. How often have you heard this?

In the real world, everyone makes mistakes. Even if a developer can muddle through with this attitude for a while, this will not be the case when they come to develop real software systems which are much more complex containing many lines of code using sophisticated algorithms and interactions. In embedded systems, there are additional space and performance constraints that will increase the complexity of the code further. The code that is initially produced may well not be perfect.

Programmers cannot identify all of the bugs in the code during a review or walkthrough. The only way to be sure of the integrity of all the code is to test all the code.

The best route to good, tested code is through structured unit testing and not relying on programmer over confidence.

CANTATA

> Robustness tests can exercise the edge and error conditions of complex processing logic, which may not have been considered by the programmer.

> Code coverage diagnoses both missing constructs and code which is not required.

3.5 Integration tests will catch all the bugs anyway

Time has been allocated in the plan for testing the system once it has been developed so there is no need to perform unit testing during the development.

Large integrations of code are more complex than single units. If unit building blocks have not been tested first, the development team may spend a lot of time simply getting the integrated software to run, prior to executing any integration test cases. Starting such integration testing on an embedded target platform will be particularly difficult as debugging and resetting the software on target is often time consuming.

Once the software is running, the developer is then faced with the problem of how to thoroughly test the software functionality within units as well as the interactions between them. It can be quite difficult to create scenarios where all of the units are called, let alone adequately exercised once they are called. Thorough testing of unit level functionality during integration is more complex, and consequentially time-consuming and expensive, than testing units in isolation.

Error conditions and defensive programming scenarios are particularly difficult to test at the integration or system levels, for instance: forcing memory allocation to fail, or simulating hardware errors. At the unit test level, there are testing tools which make it easier to simulate memory failure and verify the desired behavior.

Due to these complexities it will usually take longer and cost more to thoroughly test code at the integration/system level compared to unit testing, even when it is possible.

Of course unit testing tools which support integration testing provide the greatest flexibility here. Thus providing the capability to test internal unit functionality and their use of interfaces and accessible data can be undertaken at unit level and integration level, in combination with the testing the combined functionality and the interaction between individual units. This becomes even more beneficial for embedded software with firmware and hardware in the loop testing.

CANTATA

> Can be used for both unit isolation and integration testing

> Supports for both host and target testing

> Stubbing and Wrapping allow simulation and controlled interception of all interfaces

3.6 The client pays to develop code, not write unit tests

Most clients are not experts in developing software systems and would not be expected to understand the best way to develop software or the importance of unit tests.

What the client is paying for is code that has an acceptably low residual error rate, it is likely that code that hasn't been unit tested will either be rejected by the client, be delayed due to extended integration and system testing phases or the client is left feeling the product doesn't really meet expectations. It is easy to see that a few trivial bugs in a delivered product can seriously affect the client's perception of its quality.

Being upfront early on and explaining why unit testing will reduce the overall cost of the software and reduce the residual bug rate in the delivered system will inspire the client to have confidence in the product they will receive and avoid recriminations later.

Fixing faults during integration and system testing, particularly on embedded targets, can be a time consuming and costly exercise.

Using a testing tool which supports test framework generation, test scripting in the source code language rather than a separate test language and which is integrated with the developers IDE will increase productivity and reduce the cost to the project of unit testing further.

The client pays you to develop *working* code and a good way to achieve that is to use unit tests.

CANTATA

> Automated unit and integration tests in C/C++ provide demonstrable results of code working only as expected.

> Test results can be used for device software certification in safety related systems

3.7 I don't know how to write good unit tests

This is a concern of many software engineers who are not familiar with applying unit testing to their code. The reality is that unit tests can be written in the source code programming languages with which engineers are already familiar.

There are many resources available to learn the craft of software unit testing, including free web based tutorials, books and courses taught by professional instructors. Not knowing how to write good unit tests should be seen as an opportunity to learn a new skill not a barrier.

Using test tools provides a structured environment for the unit testing. Ideally, the tool should provide a high degree of automation, be well integrated with the other software development tools used, to keep the learning curve shallow and make the tool as easy to use as possible.

So what makes a good unit test?

A complete and thorough test

There is no point executing the software under test if you do not have some means of verifying that it has behaved only as expected. Thoroughness can be measured by:

- how many of these positive requirements are tested
- robustness testing for the negative requirements
- code coverage measuring how much of the code was tested.

A simple, maintainable test

A simple test is easier to write in the first place. Where test cases are independent of each other it will also be more maintainable in the long term.

Use suitable tools and keep your tests as simple as possible to help you efficiently write good, maintainable tests.

CANTATA

> Automated test script generation parsing the source code to derive a test framework, and multiple test vector generation options makes creating more complete test scripts in C/C++ efficient and maintainable.

> Integrated code coverage supplements requirements tests.

3.8 I am working with a legacy application without unit tests so I can't unit test

A team with a large legacy code base will be daunted by the amount of effort required to unit test it. The code is already 'working' and a lot of time has been invested in integration and system testing to allow the software to be deployed in the field and is considered 'proven in use'.

The absence of existing unit tests does not prevent unit testing, it just makes it harder to get a complete set of tests for a large code base.

Unit tests could be added using an incremental approach such as initially producing unit tests for bug fixes, then for changed items only and then for new features added. This will help ensure that not only does the new and modified code work correctly, but there have been no unforeseen modifications to the existing behavior of the code. Before long a useful set of unit tests will have been produced and the remaining set of untested units will seem less daunting.

Whilst the above approach works for changes to a source file, consider what happens when a header file is modified. Every source file that includes that header file has in effect been modified and so should be considered for testing.

Almost any situation can benefit from unit tests, the earlier they are created and the more complete, the better. Of course the more a test tool provides for automated test generation the better too.

CANTATA

> Cantata's AutoTest feature generates a set of passing unit tests for existing C code

3.9 I am only making a small change, I can ‘see’ it is correct, so why bother to test?

A common assumption is that small changes to code can be verified by inspection and that all of the side-effects will be apparent.

Any change no matter how small may introduce unintended behavior elsewhere. A simple change could have unforeseen side-effects that only become apparent during later integration and system testing. A unit test will not only verify the small change but also possible side-effects of the change on the rest of the unit.

Any software change, no matter how small, could introduce bugs and hence is risky. A unit test removes this risk.

CANTATA

> Tests are maintainable and extensible.

3.10 I cannot unit test my code because it is not testable

There are many different issues that may make code difficult to test including:

- complex interactions with low level calls which cannot be simulated by stubs;
- non-returning functions;
- software which uses a well-established suite of libraries performing complex functions;

Some of these can be overcome by design/code changes early in the process. Most of these issues can be overcome with suitable testing tools. None of them should prevent unit testing.

It should also be noted that if the code is difficult to unit test, it is likely to be difficult or even impossible to test at the integration or system level and hence the code is likely to be difficult to maintain.

Code that seems to be ‘un-testable’ requires a different unit testing approach which can be made simpler by using appropriate test tools.

CANTATA

> Wrapping provides controlled interceptions of calls not possible to stub

> Expected Calls functionality checks what functions are called and when

3.11 I’m not doing Test Driven Development

Test Driven Development (TDD) is a development method that emphasizes very short development cycles with automated test cases for the each development cycle. These test cases can be unit tests.

The benefits of unit testing are agnostic of development methodology. Any development methodology can benefit from the addition of unit testing as they lead to a reduction in the number of residual bugs in the software that is supplied to integration and system testing.

CANTATA

> Provides a framework supporting both testing existing code and TDD methodologies.

> Automatically creates required test artifacts (e.g. makefiles and test scripts) that allow the tests to be executed.

3.12 Maintaining unit tests is pointless

An argument for not unit testing is that once the code has been developed, the unit tests will be discarded and the code maintenance will continue without unit testing as changes can be visually inspected.

A complete set of up-to-date unit tests is a valuable tool for regression testing. Once the software has been developed and initially tested, the likelihood over the development and maintenance lifecycles of is that code will change, as functional enhancements are implemented and residual bugs are identified and fixed.

Some of the existing unit tests may need to be updated in line with the changes to processing in the source code. However, having a working set of unit tests provides a powerful tool to pin-point individual previously working units now failing tests and to identify unwanted side-effects elsewhere.

CANTATA

> Cantata provides automatic full regression testing following code changes

3.13 Testing is a job for testers / QA

Separate testers or a Quality Assurance function are not best placed for code level testing. Understanding the detailed design / low level requirements of a software unit and designing a set of suitable tests requires programming skills and knowledge not usually present in a separated team. Even when a separate team is employed to test the application, the style of system level tests is not well suited to uncovering the types of bugs which can more efficiently be found in developer level unit and integration testing. The Quality Assurance function is to verify that quality control activities are being performed to an agreed set of standards and processes. Developers can and are best placed to take structured quality control responsibility for the code they produce before it is passed 'over the wall'.

Unit tests which are developed using an automated tool provide a formal test record that can be checked by QA to ensure that the testing has been performed to an agreed set of the quality exit criteria and can be used as evidence in device software certification if required.

CANTATA

> Tests scripts are written in the C/C++ language under test.

> All test artifacts (scripts, options and results) are available as quality control records for QA, can be used for device software certification in safety related systems.

3.14 Interfaces are best tested during integration testing

Integration testing will generally exercise the correct use of an interface. However, unit testing can exercise the interface with both expected and unexpected values.

Many interfaces, particularly in the embedded systems, will be programmed to ensure that any parameters passed across the interface will be handled correctly to avoid data corruption or system failures. It is important to know that this defensive programming works correctly and produces the expected results.

CANTATA

> Advanced stubbing and wrapping allows both simulation and control of all interfaces.

> Robustness testing and table driven test cases can exhaustively test interfaces.

3.15 Our competition don't unit test, so it would make us uncompetitive

This is a common concern both for software houses that are bidding for work or companies producing a product. A customer will not be impressed with a product that is faulty or doesn't meet their requirements. Successful businesses are ones that keep customers happy and gain repeat business from them.

Faulty products will have to be fixed, which could work out very expensive, and negate any profit made from delivering a sub-standard initial product.

CANTATA

> Is used in every safety related industry and many business critical sectors.

4 What do I gain from unit testing?

Unit testing is often seen as a tool to ensure the code contains no simple code errors. When this is seen as the only benefit then a cost/benefit analysis may show unit testing to be unjustified. However, there are many benefits to unit testing beyond checking a coder has not made a simple coding error.

4.1 Unit tests enable collective ownership

The concept of collective ownership is discussed at ExtremeProgramming.org [3]. This states that your unit tests guard against other developers breaking your functionality. Developers are required to ensure all unit tests still pass before releasing any new code, thus ensuring that changes being made do not break another part of the system. Faults introduced by multiple developers working on common functionality can easily be missed, resulting in bugs being delivered to customers.

Thus, unit testing leads to reduced costs for future releases of the system. Implicitly a new revision will involve change, a complete set of unit tests will help ensure that existing features are not broken when new ones are added.

4.2 Unit tests enable safe refactoring

Refactoring is a very common activity in modern OO development, and is an activity where it is very easy to introduce bugs. A complete set of unit tests will ensure that a refactoring does not introduce an unexpected bug.

4.3 Problems are found early in the development cycle

The cost of fixing a bug increases the later it is found in the development cycle. Consider the cost of a developer discovering a bug in his code at unit test time. At this stage, the only changes will be to the code, and possibly the design document. Now imagine the costs if this bug were to get to system acceptance. At this point, the code and design will still need updating, but now integration and system level tests will need to be rerun, with further updates possibly being required. It should also be noted that it may be very hard to identify and then isolate a simple coding error from any other unexpected failure in the system

Unit testing cannot eliminate this problem but should significantly reduce the number of iterations required. A code base with a large number of simple errors can be very difficult to debug as some bugs may corrupt a resource used by a different area of the code. In this case, fault finding can be a very arduous task. This type of bug is very easily found in a unit test using negative testing.

4.4 Unit tests document the code

The tests can be used as an effective way to document the code's intended behavior. The tests embody the characteristics of the unit from a functional perspective. It is then possible to look at the tests to gain an understanding of the characteristics of the unit.

In "*Beautiful Code*" [5] Alberto Savoia states "*With a few lines of JUnit code, run automatically with every build, you can document the code's intended behavior and boundaries, and ensure that both of them are preserved as the code evolves.*"

Traditional narrative documentation generally tends to drift out of sync with the code over time. With this methodology, the tests should provide examples of how the code is to be used. A user coming to a new piece of code should be able to use the tests to understand how they interact with the code.

4.5 Unit tests demonstrate progress

Unit tests provide a definitive way of measuring progress on a project. Measuring progress during the development phase can be very difficult. How much credit does a coded unit get if it has never been run? A unit with a completed set of unit tests can legitimately be considered to be finished.

4.6 Unit testing improves coupling

The lower the coupling between units, the easier it is to isolation test them. While not always possible, generally low coupling between units aids understanding and maintainability as:

- a change in one unit does not normally ripple through to other units;
- assembly of the system normally requires less effort;
- code re-use is easier as the unit is not as dependent upon other units.

Unit testing can reveal a high degree of control or data coupling between units. Without unit testing there is nothing enforcing units to have low coupling and therefore designs will lead to more tightly coupled code.

5 Conclusion

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 software as early in the lifecycle as practical, with automated regression testing when 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 maximized by selection of a testing strategy which includes thorough unit testing, good management of the testing process, and appropriate use of tools to support the testing process.

We hope that this paper has convinced you that unit testing is highly desirable. However if you are still uncertain then hopefully it has at least made you considering to giving it a go. Once started we are confident you will never look back!

6 References

[1] Anonymous, Wikipedia, the free online encyclopaedia. Available from: http://en.wikipedia.org/wiki/Unit_testing

[2] S McConnell, *Code Complete*, MICROSOFT PRESS,ISBN 978-0735619678

[3] Unit Tests – Lessons Learned, http://www.extremeprogramming.org/rules/unittests.html.

[4] ExtremeProgramming.org, http://www.extremeprogramming.org/ .

[5] Beautiful Code, O'Reilly Media, ISBN 0-596-15843-2

CANTATA

Cantata is a unit and integration software testing tool, enabling developers to verify standard compliant or business critical C/C++ code on embedded target and host native platforms. Cantata is integrated with an extensive set of embedded development toolchains, from cross-compilers to requirements management and continuous integration tools. The Eclipse GUI, tight tool integrations, highly automated C/C++ test cases generation, all make Cantata easy to use.

If you want to find out more on how Cantata can accelerate your unit and integration testing, why not start a free trial?

This can be requested from the QA Systems website or by emailing sales@qa-systems.com directly.