15 Ways to Maximize the Value of Unit Tests in Safety Critical Projects

In safety critical software development, unit testing is mandated by standards. However not all tests are built equal. This paper discusses approaches you can take with your unit testing to maximize their contribution to the development process.

www.qa-systems.com

Contents

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 software quality and reliability being addressed through a growing adoption of design methodologies and supporting CASE tools, today most software designers have had some training and experience in using 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.

In safety critical development, unit testing is mandated by standards. However, tests are often produced too late in the project development to offer any actual value other than a check box exercise. Effective and well considered unit tests can bring a multitude of benefits to your project beyond standards compliance. This paper offers approaches to help maximize the value of your unit testing efforts. We show how a professional unit testing tool, such as Cantata, can help your developers and testers focus on the content of the tests rather than the drudgery of building and maintaining a unit testing suite.

2 Why do safety critical software standards require unit tests?

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 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 can be exercised with no reliance on the rest of the application. That practicability may involve tightly coupled clusters of code or single items in isolation.

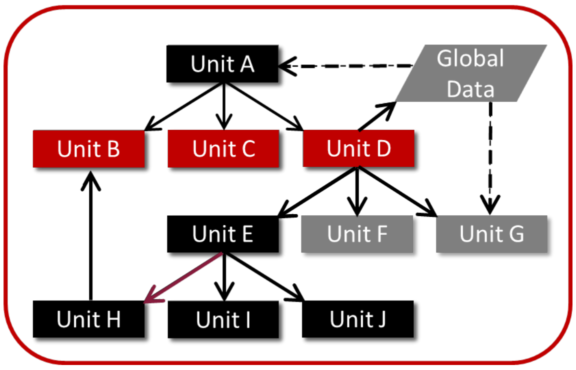

Illustration of multiple software units in a system

Safety-critical systems are those systems whose failure could cause loss of life, significant property damage, or damage to the environment. When developing software for such systems it is important to ensure that every component performs as expected under all conditions. We need to consider not only the expected inputs but also unlikely events. For instance, if a hardware sensor fails and starts feeding readings to software out of the normal operating conditions, we need to check that the software will handle the condition correctly. This testing is difficult to do at the system level, as the number of fault injection points is large to fully simulate all conditions. Not only that, but by waiting until we have a complete system before beginning to test, risks the discovery of bugs close to project completion. It is much more efficient to find and fix software problems as close as possible to the time the code is being written.

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.

Because unit testing has been proven to reduce software problems in the field, it is natural that the software safety standards for safety critical system mandate them. Going through running unit tests to satisfy the standards can be tedious work. By focusing on the value this testing can bring to the development team, beyond straight standards compliance, can help reduce coding time, speed up the system testing stages and produce a more robust and reliable overall product.

2.1 Certified Testing Tools

Using testing tools that have been pre-certified to the required standards can both remove some repetitive work in building and maintaining a test suite and reduce the burden of regulatory compliance. All the main safety related standards impose some means by which the quality of the test tools used is demonstrated to have achieved certification credit for the device software.

Open-source tools can be qualified or certified as suitable or useable in a safety related context. However, the costs associated with such an exercise (defining requirements for the tool, developing and running tests against those, and showing the tool operates correctly in the end users environment), are prohibitive. It is for that reason that developers of safety critical devices just do not do it.

Most commercial unit test vendors have sought and achieved some independent certification of their tools, or they support user’s qualification of their tools (where this is required e.g., by DO-178C).

CANTATA

SGS-TÜV Saar GmbH, an independent accredited third party certification body for functional safety, has certified Cantata as “usable in development of safety related software”, up to the highest safety integrity levels, for all the main safety related standards:

> IEC 61508:2010 (general industrial)
> ISO 26262:2018 (automotive)
> EN 50128:2011 (railway signaling)
> EN 50657:2017 (railway rolling stock)
> IEC 60880:2006 (nuclear power)
> IEC 62304:2006 (medical devices)

SGS TÜV SAAR
CERTIFICATE NO FS/71/220/19/0413 PAGE 1/1

LICENCE HOLDER
QA Systems GmbH
Roggenstraße 11
71334 Waiblingen
Germany

MANUFACTURING PLANT
QA Systems Ltd
2 Palace Yard Mews
Bath, BA1 2NH
United Kingdom

PROJECT NO/-ID: K3UX-AU09
LICENSED TEST MARK: TÜV SAAR [illegible]
CERT. REPORT NO.: K3UX0015

Tested according to: IEC 61508:2010, ISO 26262-8:2018, clause 11.4.9, EN 50128:2011, IEC 60880:2006, IEC 62304:2015
Certified product(s): Cantata, Core product
Model(s): Version: 9.0; Build_id: RELEASE_9_0.4
Technical Data and Parameter: Suitable for verification of safety-related software acc. to:
- ISO 26262 up to ASIL D
- IEC 61508 up to SIL 4, class T2 tool
- EN 50128 up to SIL 4, class T2 tool
- IEC 62304 up to SW safety Class C
- IEC 60880 for category A functions

Specific Requirements: The certificate is based on voluntarily tests. Any changes to the design, components or processing may require repetition of some parts of the pre-qualification in order to retain the certification. The certificate report is an integral part of this certificate. The safety related application guidelines shall be maintained.

Certification Body for Functional Safety SGS-TÜV Saar GmbH
Munich, 06.08.2019
[illegible]
Gudrun Neumann

CERTIFICATE

QA Systems believe a vendor should ease your route to device certification, so available free of charge for Cantata are comprehensive tool Certification Kits for all of the above standards including:

> Detailed standard briefing – mapping the requirements to Cantata capabilities,
> Cantata Safety manual – providing guidance for how to use the Cantata tool in a safety related context, and to meet the requirements of the independent tool certification,
> SGS-TÜV Certificate & Certification Report.

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

Cantata also has a comprehensive Tool Qualification kit for DO-178C / DO-330 available under an NDA and a one-off charge.

3 15 Ways to Maximize the value of unit test investment

Having established what unit testing is and why safety critical standards require you to do it, we can now explain 15 keys ways that you can maximize your return on investment in unit testing.

3.1 Consider all code units as unit testable

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

- complex interactions with low level calls which cannot be simulated by stubs
- non-returning functions and continuous loops
- use of libraries performing complex functions

We can overcome some of these by making design/code changes early. Most of these issues can be overcome with suitable testing tools. None of them should prevent unit testing.

We should also note it 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' often requires specialist test tool support for these special circumstances. Examples include accessing protected functions / data or intercepting calls to inject errors or get control of execution flow, and specialized code coverage for dealing with 'untestable' code contexts.

CANTATA

> Cantata Wrapping provides controlled interceptions of all function calls, which is not possible with stub simulations.
> Cantata White-box testing automates access to static/private functions and data.
> The Expected Calls functionality checks which functions are called and when.
> The Custom Code Insertion feature provides full test control anywhere in the source code.
> Cantata infeasible code coverage measures code execution in 'untestable' contexts.

3.2 Trust unit test proof, not over-confident coders

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

In the real world, everyone makes mistakes. Even if a software developer can muddle through with this attitude for a while, this will not be the case when they come to develop complex software systems 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 the bugs in the source code during a review or walkthrough. The only way to be sure of the correctness of all the code is to dynamically test all the code.

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

CANTATA

> Cantata is supplied with a comprehensive set of built in testing tutorials and a help system, enabling anybody to come-up to speed quickly and create effective tests.
> 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.
> The Cantata Trace feature links test cases with requirements, helping programmers identify missing or incomplete implementation of requirements.

3.3 Unit test early, so units are easier to test

Time is allocated in the plan for testing the system once it is developed, so why should we create thorough unit tests during the development phase?

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 getting the integrated software to run prior to executing any integration test cases. Starting such integration testing on an embedded target platform will be 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 and the interactions between them. It can be difficult to create scenarios where all the units are called, let alone adequately exercised once we call them. Thorough testing of unit level functionality during integration is more complex, and consequentially more time-consuming and expensive, than testing units in isolation.

Error conditions and defensive programming scenarios are 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.

Because of 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.

Unit testing tools which also support integration testing provide the greatest flexibility here. This becomes even more beneficial for embedded software with firmware and hardware in the loop testing.

Where this is available, the tester can choose whether to verify use of external function calls and external data during unit testing, or to verify these during integration testing. The expected behavior of the unit can be verified earlier by unit isolation testing using simulation of these external dependencies. However, the sequence of the interactions between units (e.g. call order and data read/write order) can be better verified using interceptions of the integrated software. There are no hard and fast rules on when to do unit isolation testing and when to do integration testing, but earlier unit testing (even if the scope is limited) is usually more effective.

CANTATA

> Cantata can be used for both unit isolation and integration testing.

> Supports both host and embedded target testing.

> Stubbing and Wrapping allow both simulation and controlled interception of all function call interfaces during both unit and integration testing.

> Call sequences and orders or data manipulation can be automatically verified.

> Potential code fixes for errors found at unit testing can be stored with Cantata Custom Code Insertions without modifying the production source code.

3.4 Unit test early, so bugs are cheaper to fix

Producing unit tests can be time consuming and when performed as code is written, it can slow down the code production phase, but that is not the complete 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 challenges are when to find bugs and to keep 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 after coding, 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 or integration 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.
- Verifying defensive programming can be very difficult and expensive to system test but is 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 later stages 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 the existing code.

We should consider the complete project, not only the initial code production phase, but all the later stages of integration and verification. Over the whole software development 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 test 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

> Cantata automated test script generation parsing the source code to derive a test framework with multiple test vector generation options, makes test creation faster.

> Efficient graphical development of tests written and editable directly in C/C++ code.

> Works within developers IDE on host and embedded targets.

> The Cantata AutoTest feature automatically generates complete, passing unit test. scripts to exercise C/C++ code, preventing future regression errors

3.5 Write unit tests in your programming language

A concern of many software engineers who are not familiar with applying unit testing to their code is that they will need to learn yet another test scripting language. Unit tests can be written in the programming languages (i.e. C/C++) 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 is 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 high 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 the functional and non-functional requirements are verified (requirements-based testing)
- how many scenarios of software doing anything it should not do are verified (robustness testing)
- how much of the code was tested (structural code coverage)

A simple, maintainable test

A simple test is easier to write. 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

> 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.

> The integrated code coverage helps identify missing tests.

> The Cantata tool includes an extensive library of built in tutorials. Hands-on training courses in testing with the tool are also available.

3.6 Unit tests should be created by developers

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 used to test the application, the style of system level tests is not well suited to uncovering the 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.

3.7 Keep unit tests synchronised with code

A common assumption is that minor changes to source code can be verified by visual inspection or static analysis, and that all the side-effects of change 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 slight change but also side-effects of the change on the rest of the unit.

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

CANTATA

> Cantata tests are maintainable and extensible and can reliably be re-executed without depending on a programmer's memory or understanding of possible regression errors.

> The Cantata Code Change Manager feature analyses changes in source code through impact analysis to automatically refactor and update the tests.

> Cantata automates regression testing, so tests are always reliable and repeatable.

3.8 Automate unit testing against requirements

Automatic test case generation technology from source code can deliver a significant reduction in time and cost when producing unit test cases. However, any tests will only ever be as good as your knowledge of what the code should do and should not do. Test cases auto-generated only from the source code will only prove that the code does what the code does. The pre-requisite of 'requirements-based' unit testing is to have well defined unit level requirements which define the expected behavior and absence of unexpected behavior.

For complicated unit code, test vectors are often very complex, making it hard to manually write test cases as well as to verify that requirements are sufficiently satisfied by the test cases. It can be difficult to manually calculate the necessary combinations of pre-conditions & inputs to drive the code down a path and produce the correct expected behaviors / outputs, and post conditions. Many separate complex test cases might be needed to test code even when atomic unit level requirements are available. These can be difficult to produce even when requirements are correct, complete, unambiguous and logically consistent.

If software requirements are the basis for requirements-based tests, any auto-generated test cases from source code need to satisfy the software requirements. To achieve this requires human insight and experience to assess the test cases. However, unit test tools can provide a framework to make tracing tests to software requirements a quicker and more logical task. For this it is helpful if all information required is presented clearly, with natural language descriptions of why the test cases were generated and full visibility of the test case details in a single view.

The most effective way to generate requirements-based unit tests is using a combination of automatic test case generation from source code, combined with tracing these tests to unit level requirements which define the expected behavior and absence of unexpected behavior.

CANTATA

> Cantata AutoTest creates passing unit test scripts from C and C++ source code with a description for each test case that explains why it was generated, so they can be easily traced to requirements.

> Cantata Trace provides full bi-directional traceability between requirements and test scripts for traceability to the definition of what the code should do.

> Use of Cantata AutoTest and Trace in combination for the Automotive and Avionics software safety standards is explained in more detail in the white papers [6]:

- Automating Requirements Based Testing for ISO 26262
- Automating Requirements Based Testing for DO-178C

3.9 Unit test and de-bug in parallel

It is a widespread misconception that de-bugging is the same as testing. The aim of a test is to verify that the code does what it should do correctly and nothing else. However, the aim of de-bugging code is to identify the root cause of an error and to remove a bug. A reason for a test failure may be immediately apparent and the code / test may require de-bugging.

While de-bugging can 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 has some significant downsides because it is not a comparable activity but a complimentary one.

Unit testing is a structured activity with the defined aim 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

> Cantata test scripts are automated, structured and repeatable.

> Cantata tests can also be run under a de-bugger.

> Cantata includes extensive out of the box support for de-bugging tools.

> Cantata Custom Code Insertion automates error injection for individual test cases in a faster and more repeatable manner than de-buggers.

3.10 Test function call interfaces at the unit level

Integration testing will exercise the correct use of a function / method interface. However, isolation unit testing can exercise the call interface with both expected and unexpected values through simulation.

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 in Cantata allow both simulation and control of all function interfaces.

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

> Cantata automatically generates all required link references to resolve daisy chains of dependencies at unit testing.

> The Custom Code Insertion feature provides full test control anywhere in the source code, making isolation testing of interfaces practical at unit testing.

3.11 Do not ignore unit testing your legacy code

The amount of effort required to unit test it will daunt a team with a large legacy code base. 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 additional features added. This will help ensure that not only does the new and changed 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. The more features a tool can provide for automated test generation, the more time is saved.

CANTATA

> The Cantata AutoTest feature generates a complete set of passing unit tests for existing C/C++ code, which can be easily linked to existing requirements (if available).

> The Cantata Test Architect add-on identifies the inter-dependencies in existing code bases and automatically creates Cantata unit test projects for selected code.

3.12 Run your unit tests on all your platforms

If you are limiting your unit tests by running them only in your development system, or in a simulated environment, then you could overlook how the code will perform once deployed onto a device. Embedded compilers can treat the same piece of code differently to host-based compilers. To ensure correct operation of your software in the field, it is important to re-run as much of your testing as possible in the execution environment it was designed for.

By baking different build and execution environments into your testing framework from the outset, you are simplifying the latter stages of testing. You are also likely to identify any obscure bugs earlier in your development, saving time and costs later in the project.

The challenge for deploying tests to multiple systems can be simplified by using automated tools to build and maintain a test framework that allows for switching the build and run environment.

CANTATA

> Generates a transparent and easily modifiable make file build system. This allows you to easily switch configurations and execute your tests with both host and target compilers.

> Cantata's Project Creator simplifies the process for matching your test projects with projects used from your existing IDE or build system, streamlining the process for building your unit tests.

3.13 Maintain your unit tests even after delivery

An argument for not unit testing is that once the code has been developed, we will discard the unit tests 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 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.

> The Cantata Code Change Manager feature analyses changes in source code test script dependencies through impact analysis to automatically refactor and update the tests for the code change.

> The same set of Cantata tests can be re-run across multiple embedded target platforms, and with different build variants making test maintenance easier.

3.14 Explain why unit tests deliver working code

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 which has not been unit tested will either be rejected by the client, be delayed for extended integration and system testing phases, or fails to meet the client's 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 e.g. ISO 26262 (Automotive), DO-178C (Civil Avionics).

> The Cantata Team Reporting add-on allows client monitoring of current and historical unit test status, over multiple codebases; and developers to demonstrate continual supply of *working* code.

3.15 Use your testing credentials as a selling point

A common concern both for software houses that are bidding for work or companies producing a product is that too much focus on testing will make them uncompetitive. A customer will not be impressed with a product that is faulty or does not 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.

For safety related industries, all international software safety standards (e.g. ISO 26262 for Automotive) require software to be unit tested. To even compete in these markets, companies need to show this testing complies with these standards, including the use of the use of a certified unit testing tool. The competitive advantage comes from using the most efficient tooling to achieve the required standard.

CANTATA

> Cantata has been successfully used in every safety related and business critical industry for over 30 years.

> Lower total cost of ownership effective and automated testing reduces the need to solve bugs in the field.

> Cantata encourages developers to adopt Shift Left testing – where more testing is done earlier in the software development lifecycle, so overall testing costs are reduced.

4 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 simplified. Developers will 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 use of tools to support the testing process.

We hope that this paper has convinced you that unit testing is not only highly desirable when working towards standards, but that they can add actual value to your development process.

5 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

[6] Automating Requirements Based Testing for ISO 26262 / DO-178C (QA Systems White Papers)

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.