LDRA
Software Technology

Technical White Paper

Software of Unknown Pedigree (SOUP) and the LDRA tool suite®

www.ldra.com

* Registration required to download the document

Contents

Introduction

Software test tools have been traditionally designed with the expectation that the code has been (or is being) designed and developed following an ideal development process which adheres to a best practise approach. Such an approach implies the existence of clearly defined requirements, an adherence to corporate or international coding standards, a well-controlled development process, and a coherent test regime.

Legacy code turns the ideal process on its head. Although such code is a valuable asset, proven in the field over many years, it was likely to have been developed on an experimental, ad hoc basis by a series of "gurus" – experts who prided themselves at getting things done and in knowing the application itself, but not necessarily expert at complying with modern development thinking and bored with providing complete documentation.

Outsourced code such as open source software libraries can also offer a tempting short-cut to development teams looking to work to a tight schedule or budget. But as for the legacy code example, such libraries are unlikely to have been developed in accordance with a known software development process or methodology, or one which has unknown or no safety- or security- related properties.

Irrespective of its source, this SOUP (Software Of Unknown Pedigree) presents its own set of unique challenges. If SOUP is to be leveraged as the basis for new developments, then a process is required to ensure that it meets modern functional safety, security, and coding standards, and can be seamlessly integrated with newly developed code to complete the functionality of the product.

The Dangers of SOUP

Many SOUP projects will have initially been subjected to functional system testing which usually offers very poor code coverage, leaving many code paths unexercised. When that code is applied in service, the different data and circumstances are likely to use many such paths for the first time and hence unexpected results may occur (Figure 1).

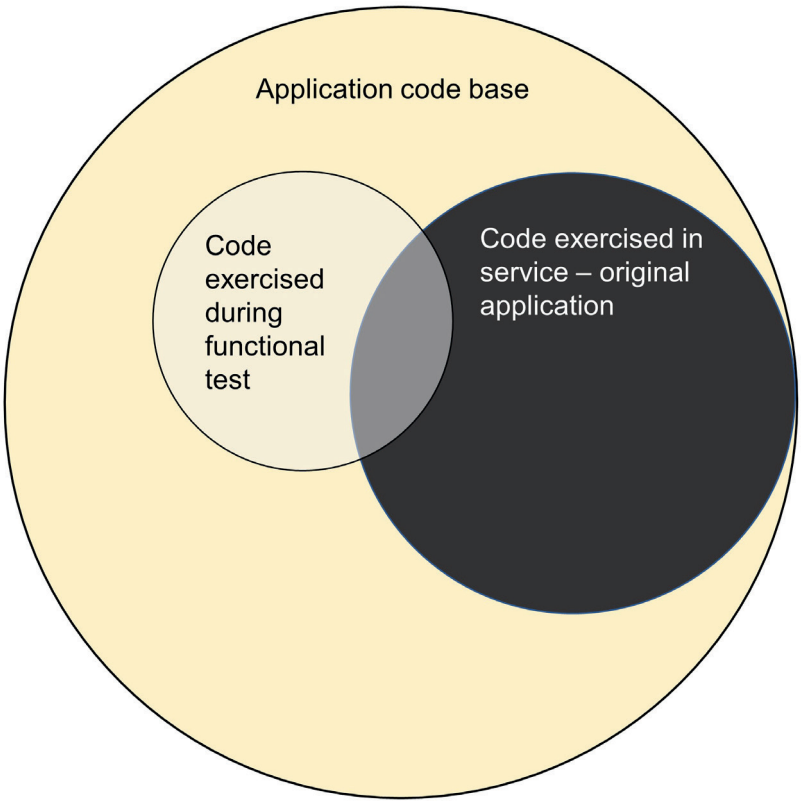


Figure 1: Traditional functional testing can leave many parts of the code unproven.

As long ago as 1996, the European System and Software Initiative "Prevention of Errors through Experience-driven Test Efforts"[1] (PET) investigated this phenomenon and established that deployed code often possesses many errors. Their findings demonstrated that the number of bugs that can be found by formalized static and dynamic analysis is substantial, even in code that has been through functional system testing and has subsequently been released.

This assertion is reinforced by functional safety standards used in the safety critical industries, such as DO-178C[2] (aerospace), ISO 26262[3] (automotive) and IEC 62304[4] (medical devices), each of which insists that such methods need to be applied before software can be deployed in their respective safety-critical domains. More recently, security guidelines such as SAE J3061[5] and MITRE's Common Weakness Enumeration (CWE) list[6] also place focus on similar validation and verification processes - no surprise given how many unsafe software constructs are also unsecure!

It is often argued that legacy code which forms the basis of new developments has been adequately tested just by being deployed in the field. However, even in the field, it is highly likely that the circumstances required to exercise some parts of the code have never (and possibly can never) occur. It follows that many unexercised paths are likely to remain in software tested only through functional testing and in the field, and such applications have therefore sustained little more than an extension of functional system testing by their in-field use.

When there is a requirement for ongoing development of legacy code for later revisions or new applications, previously unexercised code paths are likely to be called into use by combinations of data never previously encountered (Figure 2).

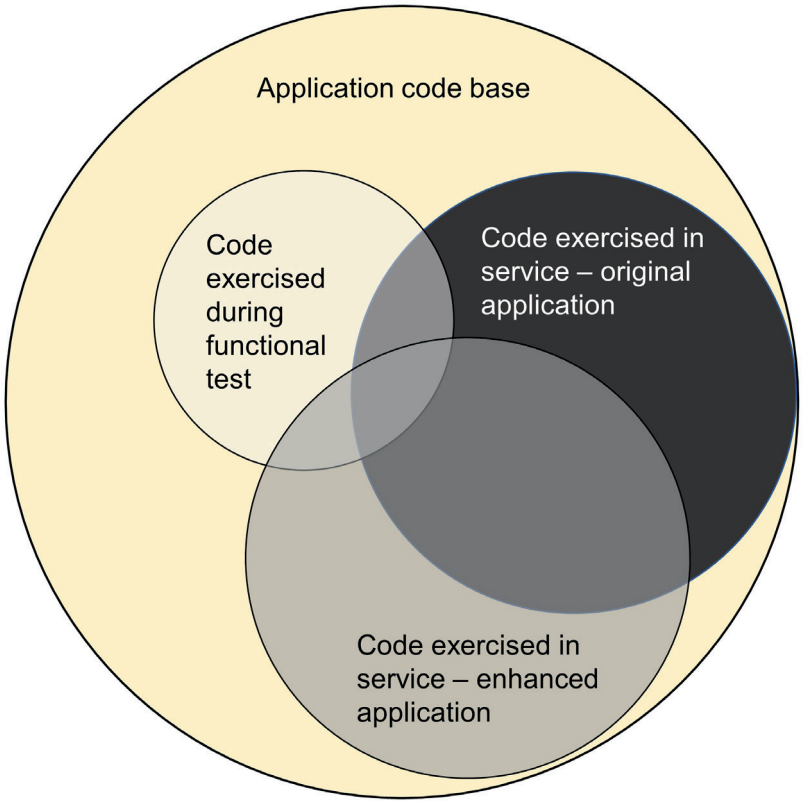


Figure 2: Code enhancements are prone to call into service combinations of data previously unencountered, resulting in the possibility of tapping into those previously unexercised paths.

[1] The Prevention of Errors through Experience-driven Test Efforts. ESSI programme: European System and Software Initiative Project No. 10438. Otto Vinter Per-Michael Poulsen, Jørn Mærsk Thomsen, Knud Nissen. March 1996.

[2] RTCA DO-178C Software Considerations in Airborne Systems and Equipment Certification, Prepared by SC-205, December 13, 2011

[3] ISO 26262:2011 Road vehicles – Functional safety

[4] IEC 62304 – medical device software – software life cycle processes

[5] SAE J3061 - Cybersecurity Guidebook for cyber-physical vehicle systems

[6] CWE Common Weakness Enumeration. A community developed list of commons software security weaknesses https://cwe.mitre.org/about/

Such situations may become particularly challenging when faced with an open source library with minimal documentation, or by legacy code has been developed by individuals who have long since left the development team, and are from a time when documentation was not of a high standard.

Given that commercial pressures often dictate the use of SOUP, what options are available?

Enhancing SOUP with formal development tools

1. Can heuristic tools improve the robustness of the code without resorting to more formalized methods?
2. What level of understanding of the existing code is available to the development team?
3. Does the existing code need to be upgraded to meet any new standards not previous enforced?
4. Should the existing functionality of the code be used as the basis for the creation of a set of formal requirements?
5. Can coding standards be retrospectively applied?
6. How can the sections of code affected by changes be tested? Can code coverage levels be proven to have adequately exercised that code?
7. Where security is a concern, is data adequately protected?
8. What if the nature of the code is such that modularity is compromised? How can new sections of code then be proven?
9. How can the revised code be proven to be functionally equivalent to its SOUP origins?

1. Heuristic static analysis of SOUP's dynamic behaviour

Some heuristic "bug finder" static analysis tools are designed to predict the dynamic behaviour of source code, to look for evidence of where source code is likely to fail, rather than enforcing standards to be more sure that it won't. These tools are often easy to apply to get a marked improvement in code quality, and are arguably complementary to the static and dynamic techniques required by functional safety standards.

Whether such tools are acceptable in isolation depends entirely on the nature of the software application under consideration, and it is clear that they sometimes lack the thoroughness sought in the development of safety-, security or mission-critical software.

Even setting that aside, these tools provide no evidence that code is functionally correct, nor that it executes correctly in its native environment – and yet software which is proven to be robust but which remains functionally flawed is clearly a potential problem.

2. Understanding existing code

If the application of purely heuristic techniques cannot provide a required level of assurance, the next step is to explore the formal methodologies promoted by functional safety standards to see how these can be adapted to address the unique demands of SOUP.

The traditional application of formal test tools is typified by the ISO 26262 'V' model diagram outlined in Figure 3.

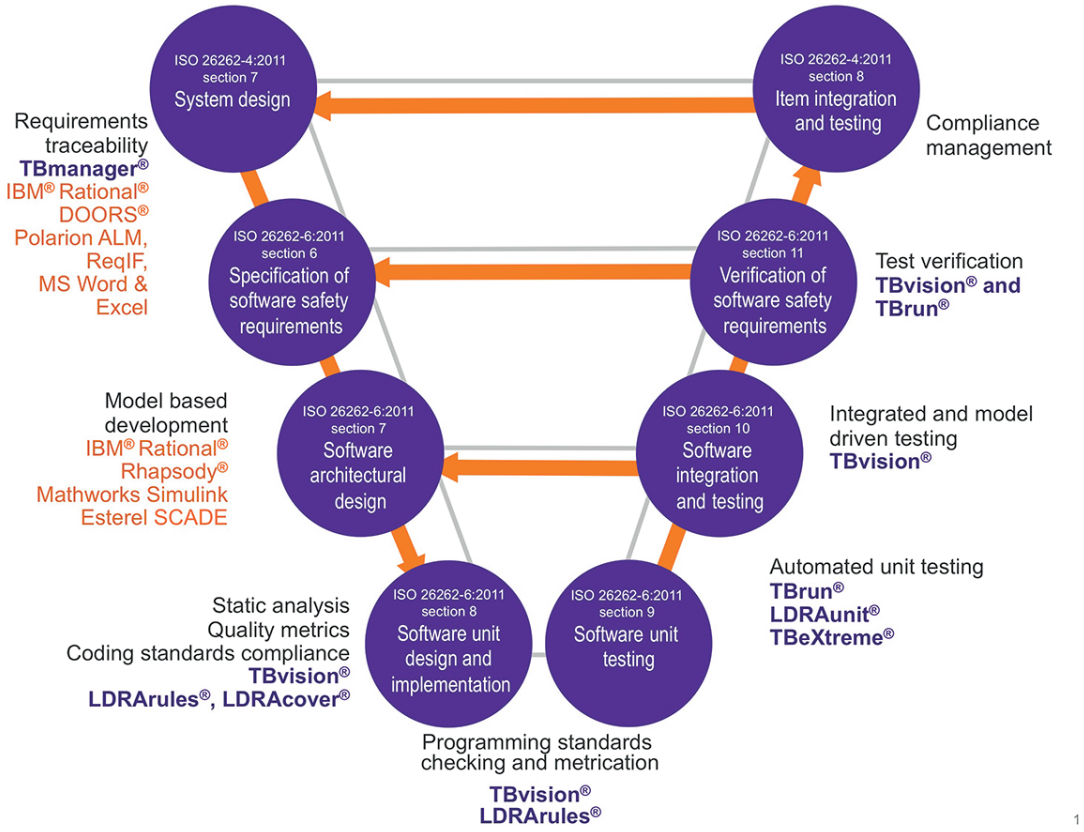


Figure 3: Traditional sequence for the application of an automated tool chain to the ISO 26262 process guidelines.

A team facing the task of building on SOUP will be required to follow a more pragmatic approach not only because the code already exists, but also because the code itself may well be the only available detailed (or at least current) "design document". In other words, the existing code effectively defines the functionality of the system rather than any documentation. In enhancing the existing code, it is vital that the existing functionality is not unintentionally modified either by rewriting to meet coding standards or as the result of the enhancements under development.

The challenge is therefore to identify and use the building blocks provided for a formalized development environment which, when used in a different sequence, can help in creating an efficient enhancement of SOUP.

3. Improving the level of understanding

The system visualization facilities provided by tools such as the LDRA tool suite are extremely powerful. Static call graphs provide a hierarchical illustration of the application and system entities, and static flow graphs show the control flow across program blocks, as shown in Figure 4. The use of such colour-coded diagrams can be of considerable benefit when understanding SOUP in that they provide a visual, easily understood representation of the underlying structure of the existing code.

Call graphs and flow graphs are just examples of artefacts resulting from the comprehensive analysis of all parameters and data objects used in the code. They are complemented by utilities such as automatic header comment generation, and data flow reporting to provide details of the relationships between the component parts of the software and problems associated with those relationships. Such information is particularly vital to enable the affected procedures and data structures to be isolated and understood when work begins on enhancing functionality.

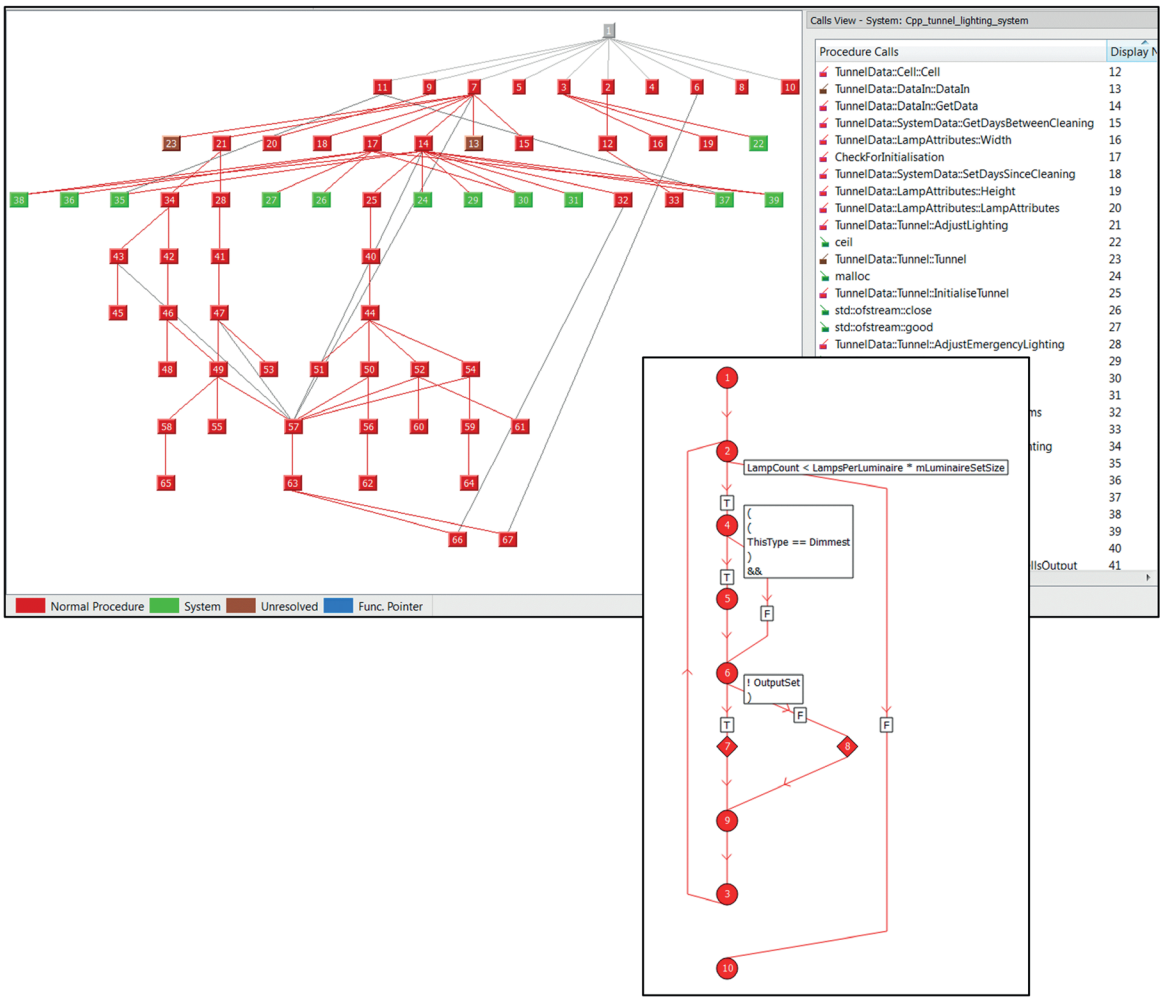


Figure 4: The static call graph and flow graph represent the structure and the logic of the code in graphical form.

4. Reverse engineering of requirements

A formal test process typified by the ISO 26262 example shown in figure 3 demands that from the outset, there should be a formal set of requirements. In this “blue sky” scenario, requirements are captured and form the basis of design documentation.

The decision whether to reverse engineer a set of requirements based on the new-found understanding of the code will be largely a commercial one, depending on the nature of the task in hand. If the updated application needs to comply with a demanding functional safety standard, then it may be unavoidable.

Clearly it is not possible to derive an underlying set of requirements of existing code in the same way that static analysis tools are capable of illustrating the design of existing code. However, requirements traceability tools typified by LDRA’s TBmanager can be invaluable in these circumstances.

Underlying any formal development process is the concept of a requirement traceability matrix (RTM) as illustrated in Figure 5. The RTM sits at the heart of the project, defining and describing the interaction between the design, code, test and verification stages of development.

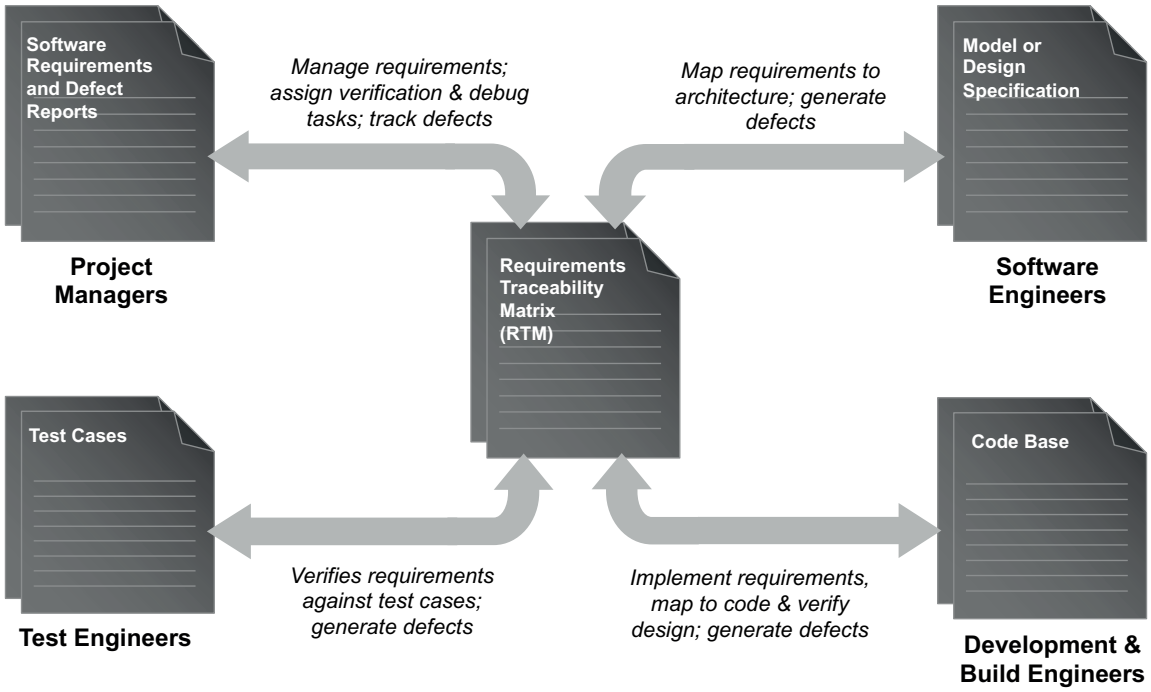


Figure 5: Formal development processes see an RTM sitting at the heart of the project, defining and describing the interaction between the design, code, test and verification stages of development.

Where requirements are to be reverse-engineered, that remains just as significant. In both cases, it is important that all requirements are traceable to software design, and that all software design artefacts are traceable to implemented code. The difference centres on the order in which that is achieved.

Figure 6 shows the relationships between artefacts in the TBmanager component of the LDRA tool suite, with the highlight showing which requirements are fulfilled by what source code. Such a tool allows any mismatch to show. In a traditional process, requirements will be completed before the source code. In this SOUP example, the opposite is true – but mismatches will be highlighted in either case.

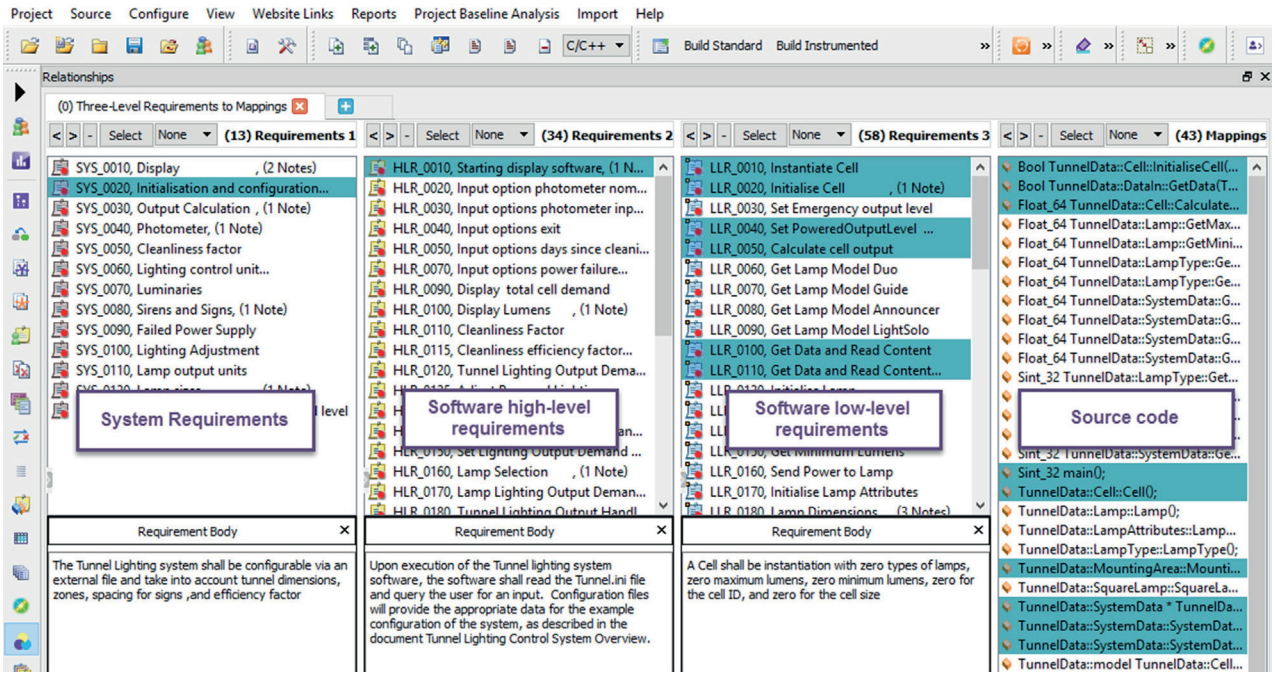


Figure 6: Automating requirements traceability with the TBmanager component of the LDRA tool suite.

5. Enforcing new standards

When new developments are based on existing SOUP, it is likely that internal, industry, or international standards will have been enhanced in the intervening period. It may be that the enforcement of a full set of current coding rules to SOUP is too onerous and so a subset compromise is preferred.

Where legacy code is subject to continuous development, it is likely that production pressures generally outweigh the longer-term ideal to adhere to more demanding standards. It may be pragmatic to initially establish a relatively lenient set of coding rules, to isolate only the most unwanted violations. A progressive transition to a higher ideal may then be made by periodically adding more rules with each new release, so that the impact on incremental functionality improvements is kept to a minimum.

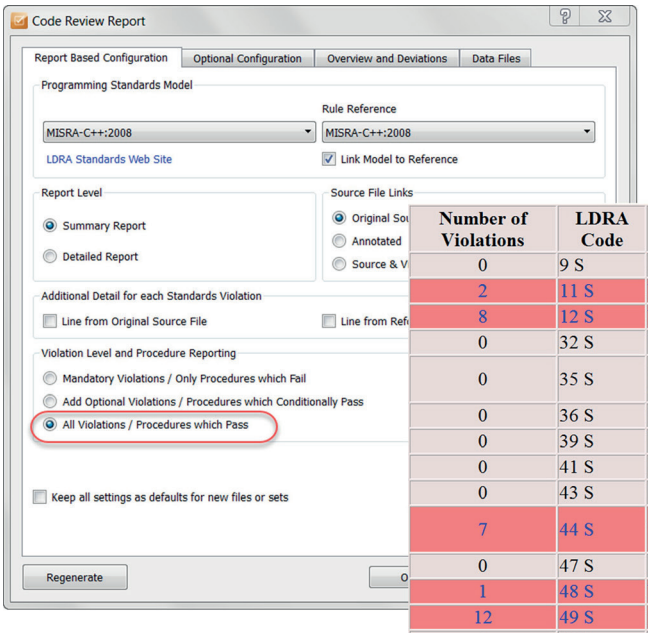
Deriving an internal coding standard

As a practical example, let us consider that the developers of such a code base designed to control a widget production machine have been tasked with establishing the quality of their software, and improving it on an ongoing basis.

Establishing a Common Foundation

The first step in establishing an appropriate rule set is to choose a global standard as a reference point – perhaps MISRA C++:2008 in this case.

Using a test tool, the code base can be analysed to discover the parts of the standard where the code base is already adequate. By opting to include information even relating to the rules which pass, a subset of the standard to which the code adheres can immediately be derived (Figure 7).



Number of Violations	LDRA Code	Required Standards	MISRA-C++:2008 Code
0	9 S	Assignment operation in expression.	MISRA-C++:2008 5-0-1,6-2-1
2	11 S	No brackets to loop body.	MISRA-C++:2008 6-3-1
8	12 S	No brackets to then/else.	MISRA-C++:2008 6-4-1
0	32 S	Use of continue statement.	MISRA-C++:2008 6-6-3
0	35 S	Static procedure is not explicitly called in code analysed.	MISRA-C++:2008 0-1-1
0	36 S	Function has no return statement.	MISRA-C++:2008 8-4-3
0	39 S	Unsuitable type for loop variable.	MISRA-C++:2008 6-5-1
0	41 S	Ellipsis used in procedure parameter list.	MISRA-C++:2008 8-4-1
0	43 S	Use of setjmp/longjmp.	MISRA-C++:2008 17-0-5
7	44 S	Use of banned function, type or variable.	MISRA-C++:2008 17-0-1,18-0-2,18-2-1,18-4-1,19-3-1
0	47 S	Array bound exceeded.	MISRA-C++:2008 5-0-16
1	48 S	No default case in switch statement.	MISRA-C++:2008 6-4-6
12	49 S	Logical conjunctions need brackets.	MISRA-C++:2008 5-0-2,5-2-1
0	50 S	Use of shift operator on signed type.	MISRA-C++:2008 5-0-21
0	51 S	Shifting value too far.	MISRA-C++:2008 5-8-1
0	52 S	Unsigned expression negated.	MISRA-C++:2008 5-3-2

Figure 7: Using source code violation reporting to identify rules which are adhered to. In this report subset, only the rules highlighted have been violated.

Typically the configuration facilities in the test tool can then be used to map the rules which have not been transgressed into a new subset of the reference standard, and the violated rules disabled as illustrated in Figure 8.

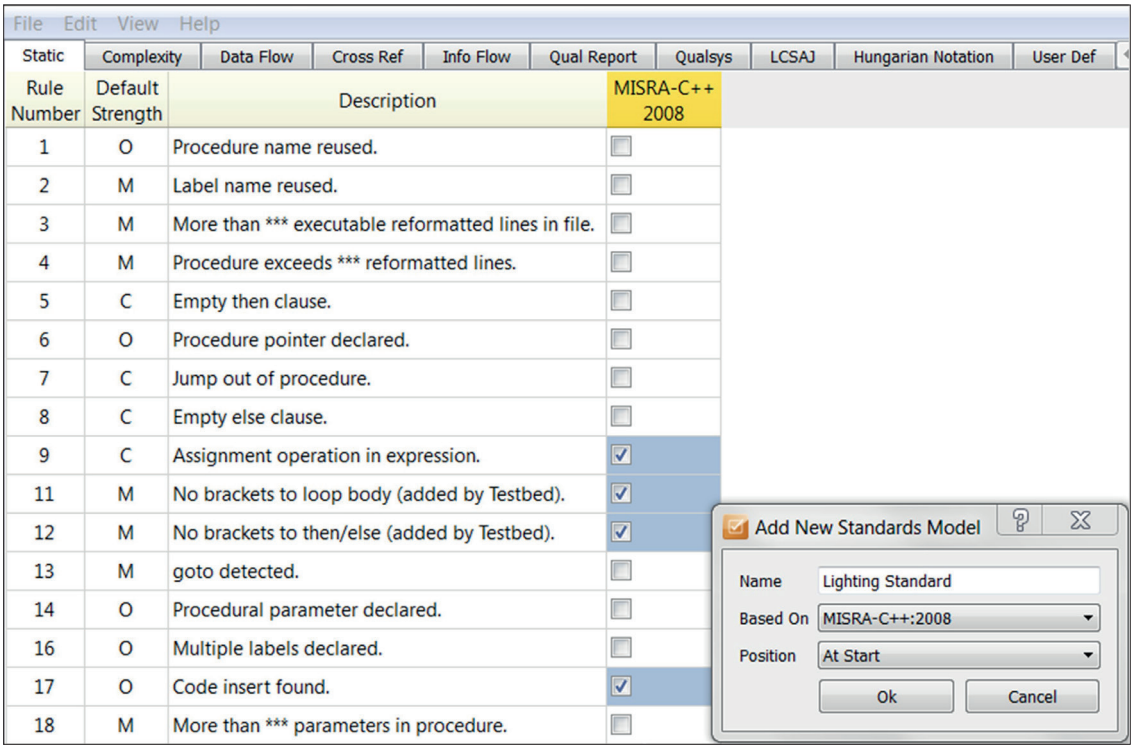


Figure 8: Using unviolated rules from a universal standard as the basis of an internal standard.

Building Upon a Common Foundation

Even if nothing else is achieved beyond checking each future code release against this rule set, it is assured that the standard of the code in terms of adherence to the chosen reference standard will not deteriorate further.

However, if the aim is to improve the standard of the code then an appropriate strategy is to review the violated rules. It is likely that there are far fewer different rules violated than there are individual violations, and test tools can sometime generate a useful breakdown of this summary information.

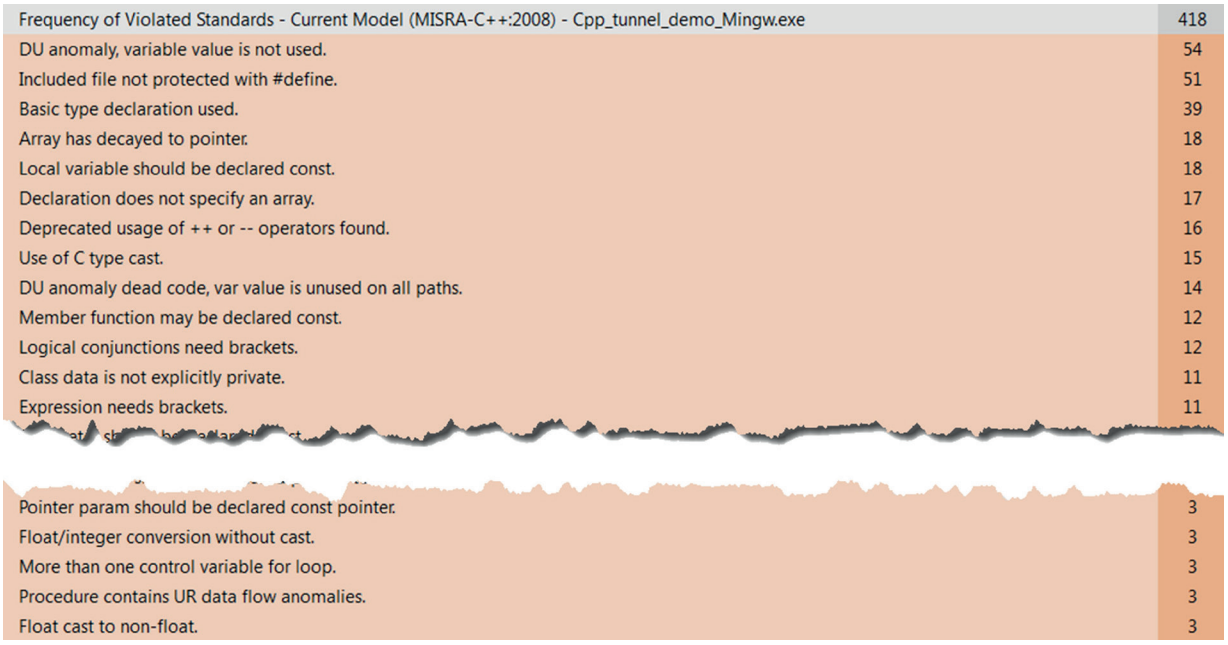


Figure 9: Summary showing the breakdown of rule violations in sample code set.

In some cases, it may be that a decision is reached that some rules are not appropriate, and their exclusion justified.

The prioritisation of the introduction of the remaining rules can vary depending on the primary motivation. In the example, it is obviously likely to be quicker to address the violations which occur once rather than those which occur 40 or 50 times which will improve the apparent adherence to the standard. However, it makes sense to initially focus on any particular violations which, if they were to be corrected, would address known functional issues in the code base.

Whatever the criteria for prioritisation, a progressive transition to a higher ideal may then be made by periodically adding more rules with each new release, so that the impact on incremental functionality improvements is kept to a minimum.

6. Validate untrusted data

The CERT division of the Software Engineering Institute is a leader in cybersecurity. Of their "top 10 secure coding practices"[7], two focus on the sanitization of untrusted data – "Validate input" and "Sanitize data sent to other systems". Clearly these are pertinent wherever SOUP is to be integrated into a connected system.

Follow any application inside a debugger and it is easy to see that information is being copied and modified all the time. Data that is input into this melting pot of moving data from an untrusted source is said to be "tainted", and hence "taint analysis" is a form of information flow analysis, focused on untrusted data.

As so often happens across software engineering sectors, many years ago an entirely separate development path established an approach to information flow analysis that was designed for use with critical embedded applications but is exactly relevant to this scenario. The DO-178C standard used in the aerospace industry leverages data and control coupling to "provide a measurement and assurance of the correctness of software modules' interactions and dependencies."

The application of control and data coupling analysis to a SOUP derived system will provide a telling insight of where and how data is manipulated, making it easy to identify where defensive strategies might be appropriate.

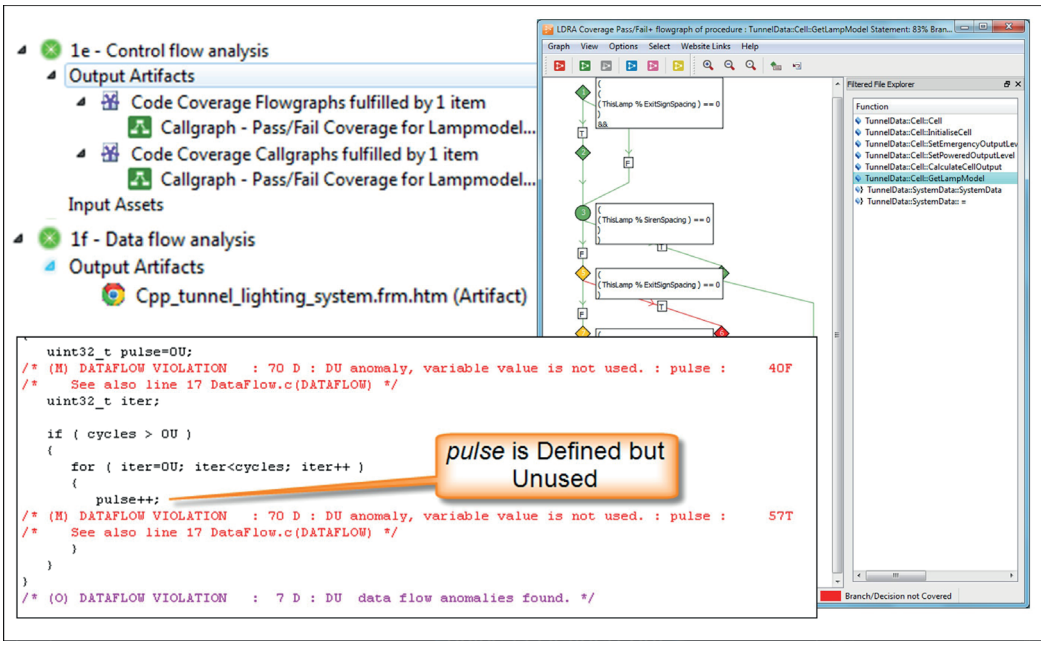


Figure 10: Control and data coupling analysis.

[7] SEI CERT Top 10 Secure Coding Practices
https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices

7. Ensuring adequate code coverage

As established earlier, testing based on a statistical sample of a program's expected input data can lead to testing being focused on a particular value or range of values. Code proven in service has effectively been subjected only to similar, if extensive, functional testing which has serious implications for SOUP enhancement.

Structural Coverage addresses this issue by testing equally across the code, assigning equal significance to the testing of each statement and branch (Figure 11).

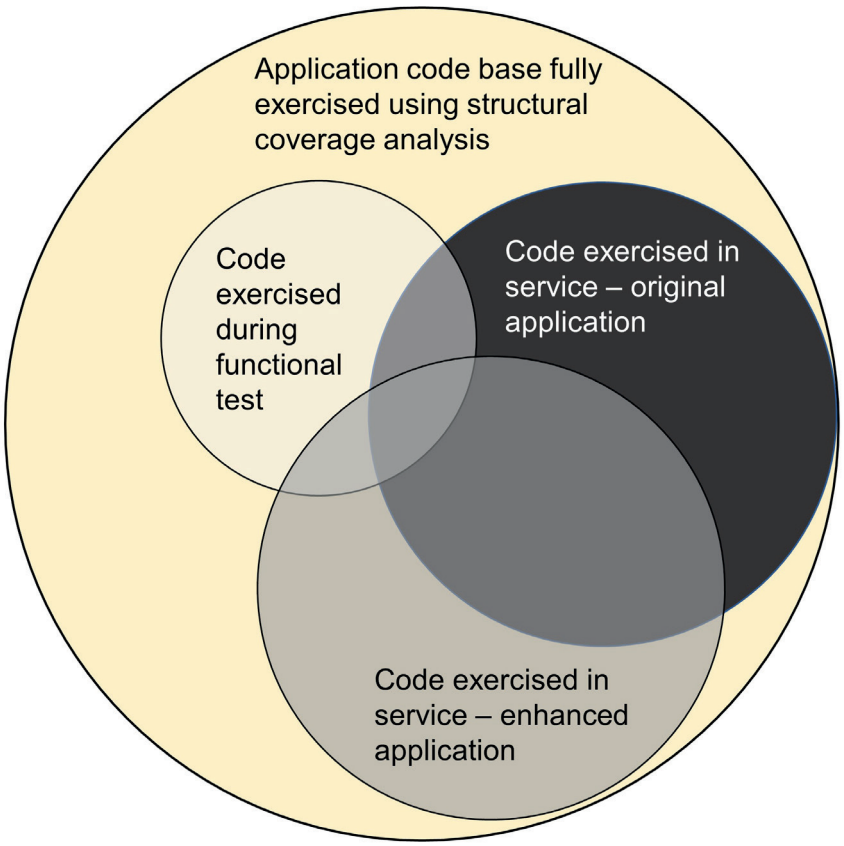


Figure 11: Adequate structural coverage throughout the code provides reassurance that changes in data or functionality will not expose untested code paths.

System testing

Although system-wide functional testing will not test all paths, clearly it will test many. Given that functional software exists at the outset, it provides a logical place to start. Structural coverage analysis provides the means to identify which parts of the software have been exercised, and so highlight those areas still requiring attention.

LDRA structural analysis tools take a copy of the code under test and implants additional function calls ("instrumentation") to identify the paths exercised during execution. Using this "white box" testing technique, tools provide statistics to show how much of the code has been used. Coloured call graphs and flow graphs complement reports to give an insight into the code tested, and to clearly show the nature of data required to ensure additional coverage (Figure 12).

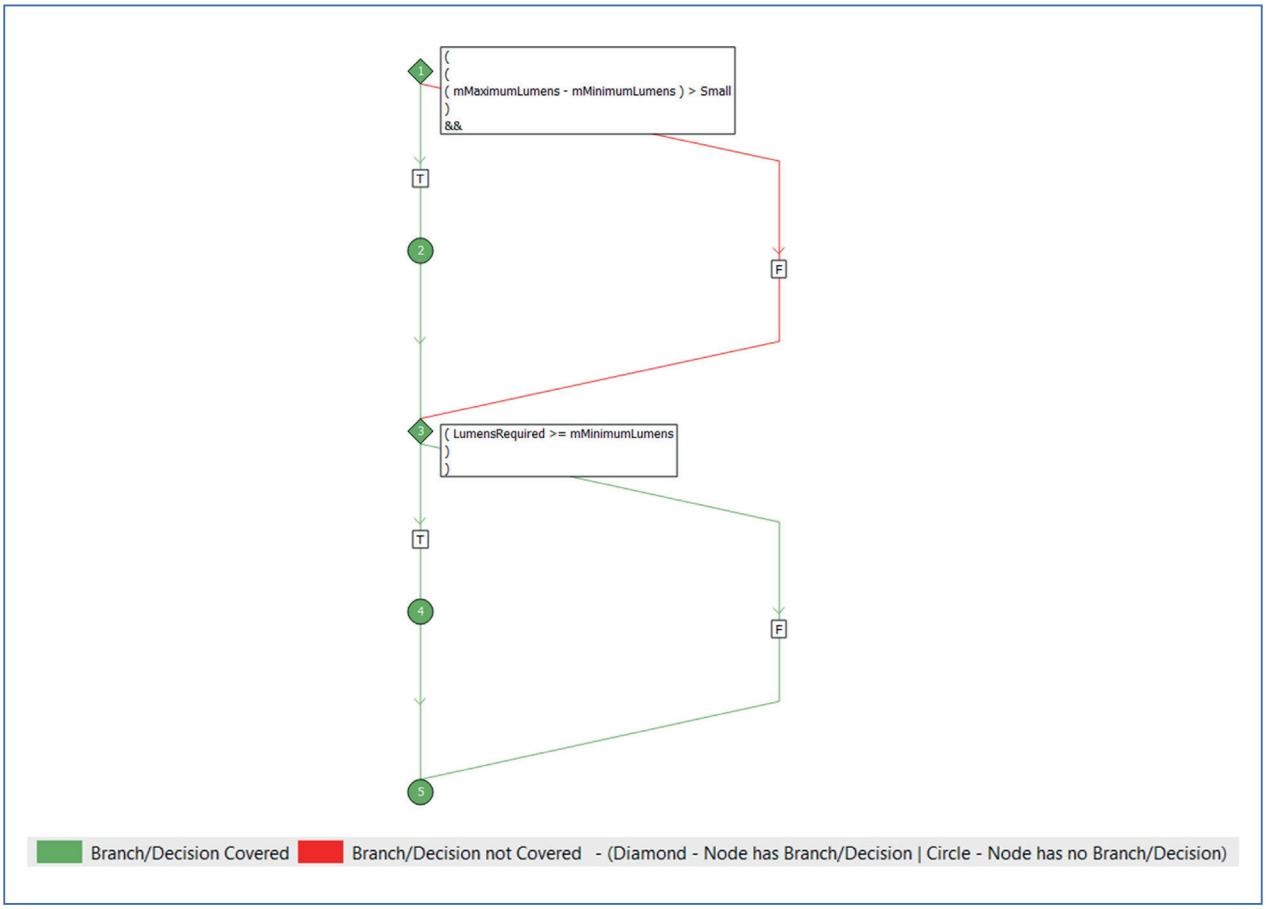


Figure 12: Coloured coded graphical information clearly identifies unexercised code.

Unit testing

System wide testing inevitably leaves some paths unproven. However, because code instrumentation can be applied on a unit test or system wide basis, it is possible to devise unit tests to exercise those parts of the code which have yet to be proven through system test. This is equally true of code which is inaccessible under normal circumstances, such as exception handlers and defensive coding (Figure 12).

Unit testing can be used to ensure that each section (or unit) of the code functions correctly in isolation. Whether testing a single function/procedure or a subsystem within an application, the time involved in manually constructing a harness to allow the code to compile can be considerable and can demand a high level of expertise on the part of the tester.

Automating unit test minimizes that overhead by automatically constructing the harness code within an easy-to-use GUI environment and providing details of the input and output data variables to which the user may assign values. It can overcome traditional headaches such as providing access to C++ private member variables for test purposes. The result can then be exercised on either the host or target machine.

Sequences of these test cases can be stored, and they can be automatically exercised regularly (perhaps overnight) to ensure that ongoing development does not adversely affect proven functionality.

8. Dealing with compromised modularity

In some SOUP applications, terminology such as “Unit test” and “subsystem” do not readily spring to mind. Structure and modularity often suffer in such code bases, which does not make them any less vital to the organization but can challenge the notion of testing functional or structural subsections of that code.

However, the TBrun component of the LDRA tool suite can be very flexible in these matters, and the harness code which is constructed to drive test cases can include as much or as little of the source code base as is deemed necessary by the user. If it is necessary to include a large section of it to test the behaviour of particular functions within a system, then so be it.

The ability to do that may be sufficient to suit a purpose: “letting sleeping dogs lie.” Elsewhere, it may be the pragmatic choice when a minor enhancement has been implemented. However, if a longer-term goal exists to improve matters, then using instrumented code can be a vital tool in helping to understand which execution paths are taken when different input parameters are passed into a function – either in isolation or in the broader context of its calling tree.

9. Ensuring correct functionality

From a pragmatic viewpoint, perhaps the most important aspect of SOUP-based development is ensuring that all aspects of the software functions as expected, despite changes to the code and to the data handled by it. In making changes to the code either to meet standards or to add functionality, it is therefore of paramount importance that functionality is not inadvertently changed.

Of course, unit tests could be used throughout to ensure that is the case but even with the aid of test tools, that may involve more work than the budget will accommodate. However, the concern here is not checking that each function is working in a particular way; more that however it works beforehand, it is not changing, except when there are deliberate attempts to make it do so.

Automatic test case generation is key here. Building upon the static analysis of the code, it is possible to automatically generate test cases to exercise a high percentage of the paths through it. Input and output data to exercise the functions is generated automatically, and then retained for future use.

They can then be used to perform batch regression testing on an ongoing basis, perhaps overnight or weekly, to ensure that when those same tests are run on the code under development there are no unexpected changes recorded. These regression tests automatically provide the cross reference back to the functionality of the original source code.

This ensures that when the enhanced software is released, clients can benefit from the new functionality and improved robustness of the revised code without fear of nasty surprises!

In conclusion

In an ideal world, a formalized development process will always start with the definition of requirements. However, sometimes commercial realities mean that SOUP code is used as a basis for further development. By using the tools proven in the development of functionally safe and secure systems in a pragmatic way, it is possible to develop SOUP code into a sound set of sources which are proven to be fit for purpose – all in an efficient and cost effective manner.

How might such a pragmatic process look? Figure 13 shows one possible interpretation.

In practise, the nature of that lifecycle is likely to depend heavily on the nature and quality of the legacy code. However, for such a system developed, there is a healthy toolbox of tools proven in the development of functionally safe systems to help with that – not least the LDRA tool suite.

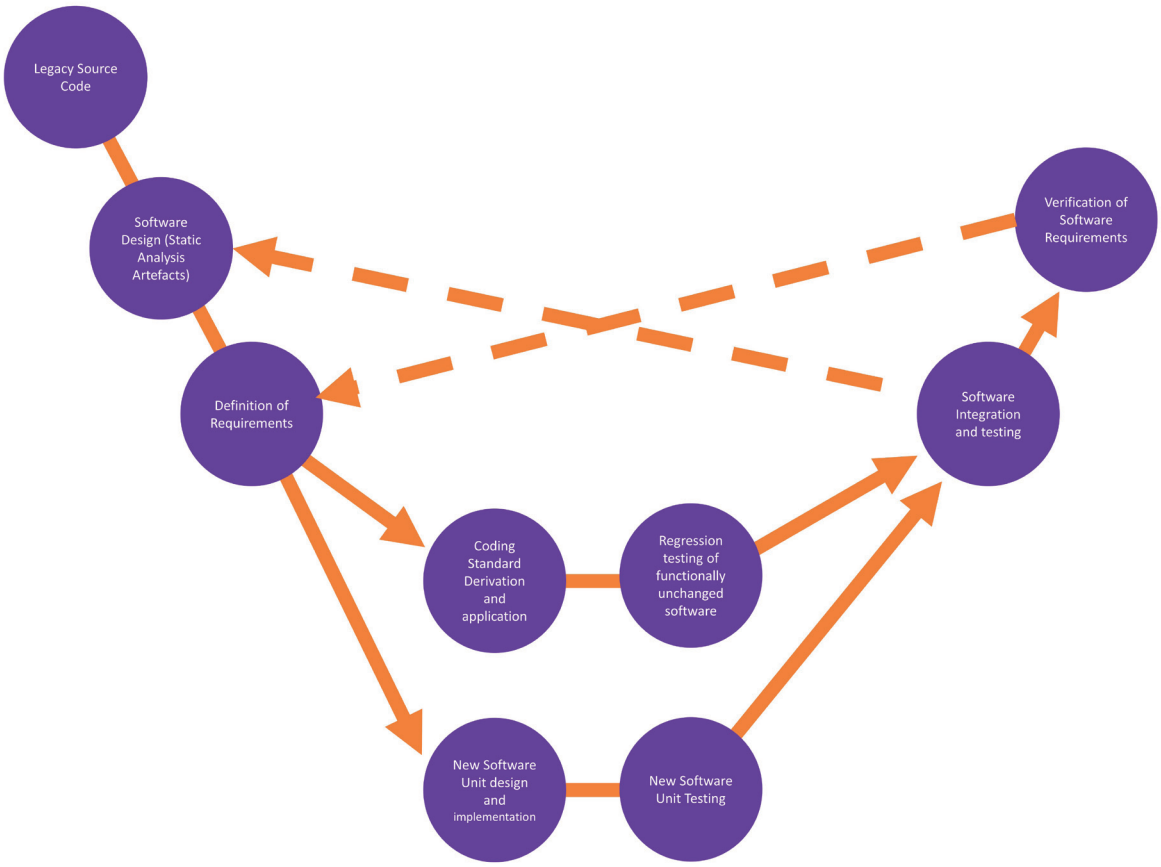


Figure 13: A possible development lifecycle for an enhanced system based on legacy code.

Works cited

CWE Common Weakness Enumeration. A community developed list of commons software security weaknesses https://cwe.mitre.org/about/

IEC 62304 – Medical device software – Software life cycle processes

ISO 26262:2011 Road vehicles – Functional safety

Prevention of Errors through Experience-driven Test Efforts, The. ESSI programme: European System and Software Initiative Project No. 10438. Otto Vinter Per-Michael Poulsen, Jørn Mærsk Thomsen, Knud Nissen. March 1996

RTCA DO-178C Software Considerations in Airborne Systems and Equipment Certification, Prepared by SC-205, December 13, 2011

SAE J3061 – Cybersecurity Guidebook for cyber-physical vehicle systems

SEI CERT Top 10 Secure Coding Practices
https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices

LDRA Technology Inc.
2540 King Arthur Blvd, Suite 228
Lewisville, Texas 75056
United States
Tel: +1 (855) 855 5372
e-mail: info@ldra.com

www.ldra.com

LDRA UK & Worldwide
Portside, Monks Ferry,
Wirral, CH41 5LH
Tel: +44 (0)151 649 9300
e-mail: info@ldra.com

LDRA Technology Pvt. Ltd.
Unit No B-3, 3rd floor Tower B,
Golden Enclave. HAL Airport Road
Bengaluru
560017
India
Tel: +91 80 4080 8707
e-mail: india@ldra.com