PRQA Programming Research

WHITEPAPER

Continuous Code Inspection

Advancing software quality at source

Co-authors
Clayton Weimer & Fergus Bolger

January 2013

The benefits of finding and fixing defects early in the Software Development Life Cycle are widely acknowledged. These benefits are not limited to quality, but simultaneously have a positive impact on schedule and cost. There is a large body of evidence which demonstrates that rigorous Code Inspections (as supposed to ad hoc reviews) are one of the most effective ways to identify and remove defects, and they achieve this at the earliest possible stage in development. In this paper we identify three key areas which need to be considered in order to make Code Inspections practical and even more effective: the inspection criteria, the inspection process and the enabling tools which provide automation.

Introduction

You may be suffering from another software crisis [1]. Costs are overrunning, schedules are slipping. The brutal reality is you're not alone. The cost of repairing bugs is today the most expensive activity in software production. Studies show that more than 50% of the total accrued costs of software projects are associated with finding and repairing defects [2] and that 50% of them are late [3].

Even equipped with the latest test tools and utilizing the latest techniques, you have come to realize there is *"No Silver Bullet"* [4], no instant magic for putting your software quality problems to rest.

The Code Inspection

There may be no silver bullet, but there is a wealth of empirical evidence for the next best thing since first described in 1976 [5]: the Code Inspection [6]. In fact, almost all empirical data measuring efforts to improve software quality since then, make the case for prevention and early defect detection - and *Code Inspection* is by far the most effective activity for doing this [7] [8] [9].

The wealth of data showing the value of Code Inspections is abundant. Compendiums of studies are also plentiful and note that Code Inspection techniques can lead to the detection of up to 90 percent of the defects [10] at a fraction of the cost of other techniques.

Alternative Peer Reviews

These studies are mainly in respect of formal Code Inspection rather than the lightweight alternative of a code review. It is the disciplined enforcement of *Code Inspection Criteria*, and the adherence to formal process that differentiates Code Inspections [11]. In fact some studies have concluded that informal code reviews can be counterproductive, and at best no better than time spent on test activities [12].

Testing

There is also the widespread myth that testing by itself provides sufficient evidence for high integrity and quality. Everything we have learned in this industry about the effectiveness of preventing, finding and fixing defects tells us this is wrong.

Certainly testing is crucial for verifying *functionality*. However, testing by itself is inefficient:

Testing cannot catch all defects – if the code is too complex it may not be fully testable.

Testing is expensive – defect detection invariably requires numerous iterations of finding/fixing issues followed by re-integration.

Testing is hard – even with automation, verifying functionality is painstaking work; when it is mired in code structural defects and associated repairs chaos ensues.

Drawbacks

A more effective technique is rigorous inspection of the source code early and often. But if this is so effective - why is it not universally embraced?

Code Inspections have been deemed too much of a strain on key resources too early in the development process, especially in a world which increasingly embraces the idea that development should be more *agile* - not burdened with heavy-weight processes:

No measurable benefit - perceived as discussion-forums, with no defined outcome.

No follow through - comments are ignored and modification is resisted using the justification; "it compiles, links and passes its tests".

Lack of buy in - all programmers want to do a good job, but if there are inconsistencies in rule definitions and enforcement, they will find ways to avoid them.

Peer fear – reviews become too emotive and confrontational, and in the extreme become counterproductive degenerating into disparaging and personal comments. There's also the risk of management using the results in performance appraisals.

Boring – reviews often get bogged down in the tedious, the mundane or the irrelevant.

It is indeed true that Code Inspections are tedious, labor intensive, and can be a rigid, low tech process. The rest of this paper will describe a more practical and effective solution.

The Continuous Code Inspection

What makes a *Code Inspection* unique is the rigorous application of inspection criteria and its formal method of process. The *Continuous Code Inspection* is a modernization of this methodology.

In the context of developing rigorous, high quality software, we will focus on three key areas forming the Continuous Code Inspection: the ***Inspection Criteria***, the ***Inspection Process***, and the ***Enabling Technology***.

(A) Inspection Criteria

In order to make Code Inspection effective it is essential to establish well-defined criteria that need to be met at entry and conclusion of each inspection event. And, of course, these criteria should be specific and measurable.

The primary fragmentation of inspection criteria should be into ***functional*** requirements that describe the behaviors needed to support the user's needs and ***structural*** requirements that identify the attributes that address the internal integrity of the system.

The Code Inspection will typically verify ***functional*** requirements using the aid of specification-based software testing. However, it is the Code Inspection's focus on the ***structural*** requirements that truly verifies the overall quality and integrity of the system.

A useful model is provided by ISO/IEC – 9126 [13] which maps these structural attributes into the real world of system development (see Figure 1). The degree of enforcement of each quality characteristic depends upon the goals of the stakeholders, where trade-offs must be made in consideration of the requirements of the system. Additionally, there is some overlap between functional and structural requirements, for example where functional requirements state specific Reliability and Efficiency goals expected from the user's external view.[14]

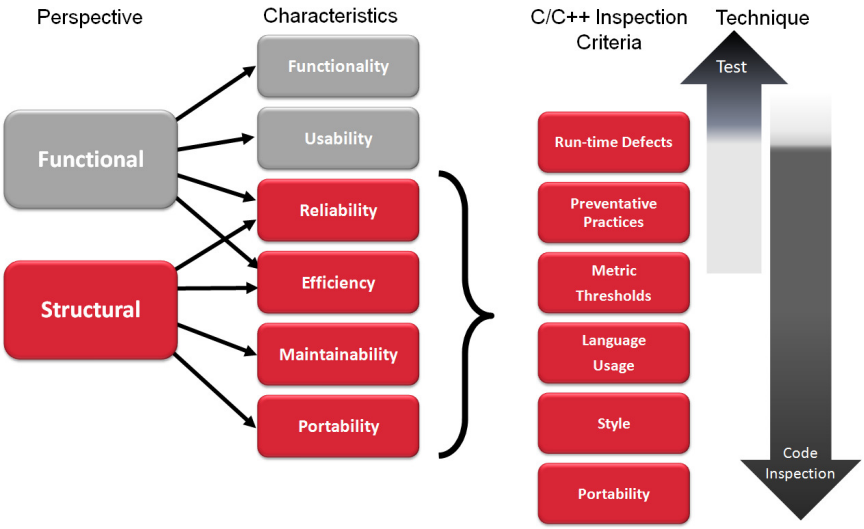

Figure 1: Inspection Criteria

In mapping to these structural attributes, inspection criteria have traditionally included a checklist or coding standard that complements the wisdom and experience provided by expert reviewers who invest a significant amount of time and effort [9].

In-house company standards are often characterized by style and coding convention guidelines. For safety critical embedded software, rigorous coding standards have been developed that extend this criteria further, codifying the wisdom and experience gained within the embedded software engineering discipline of past decades. Examples include MISRA-C [15] and MISRA-C++ [16] for automotive applications, PRQA's High Integrity C++ standard [17], the more security orientated CERT-C and CERT-C++ [18], and JSF AV++ for aviation applications [19].

A recent survey shows the popularity of these different coding standards for C/C++ (Figure 2) [3].

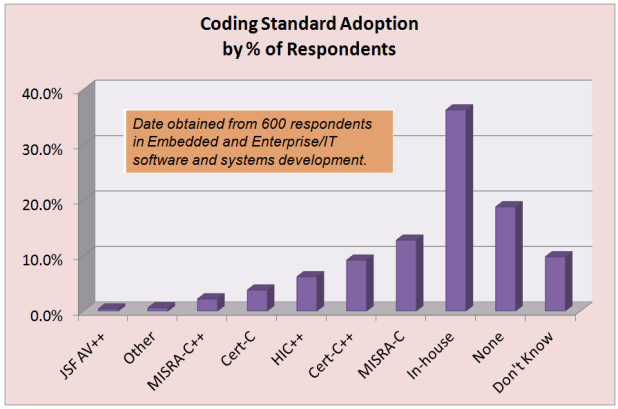

Figure 2: VDC Research

Coding rules in any of these industrial strength coding standards can be grouped and mapped to structural attributes, and thus provide the foundation for the inspection criteria presented in this paper. The following sections consider various groupings.

Run-time Defects

These are likely run-time errors that can cause unwanted behavior. Identification requires a modeling of the source code execution by walking through the data flow and control flow logic considering all potential values. For example, a path in the code may lead to a divide by zero error. The resultant system behavior can be calamitous, and the possibility of it happening needs to be eliminated. An example sub grouping of these areas:

- Initialization – (e.g. using the value of unset data)
- Arithmetic Operations – (e.g. operations on signed data resulting in overflow)
- Arrays and Pointer Operations – (e.g. array out of bounds, dereferencing NULL pointers)
- Resource & API Usage – (e.g. memory leaks, invalid pointer arguments to a function)
- Redundancy – (e.g. values assigned to identifiers never used, invariant conditions)
- Control Flow – (e.g. unreachable code, infinite loops)

Many of these may not be catastrophic errors by themselves, but may be associated with a serious logic fault.

Preventative Practices

Covering the full language spectrum, preventative practices avoid sub-optimal, unclear, confusing and error-prone language usage, instead preferring defensive programming techniques. These practices cover:

- Declarations & Scoping– (e.g. function default arguments, constness, access protection)
- Limitations on Preprocessor – (e.g. warning include guards and notes on macro usage)
- Safe Typing – (e.g. warnings on type casting, assignments, operations)
- Efficiency – (e.g. notes on better paradigms or coding alternatives).
- Design Problems – (e.g. preventive notes, potential problem warnings, resource usage)
- Maintenance – (e.g. readability and understandability notes and warnings)
- Code Structure – (e.g. analysis of switch statements, unstructured statements)

Metric Thresholds

Software complexity has an impact on reliability and maintainability, and setting acceptable complexity thresholds based on software metrics can lessen the impact. Many metrics yield complexity measurement, from the lines of code counts, comment-to-code ratio, function call counts, decision-based Cyclomatic Complexity, and class-based design complexity measures. There is extensive literature discussing the merits of a myriad of metrics, including industry standards such as HIS (Hersteller Initiative Software) and IEEE Std 1061 - Standard for a Software Quality Metrics Methodology [20].

Language Usage

The C/C++ languages suffer in particular from both a commitment to maintain backward compatibility and an emphasis on close binding to machine models. This has created ambiguity in language usage which, while remaining legal, can be the cause of subtle and hard-to-find defects. Ideally, conforming compilers would identify and document these implementation-specific behaviors. In reality this is not always the case [21], and protection is only reliably provided by sophisticated and thorough static analysis based on language analysis.

Style

Unfortunately, some organizations have an over-simplistic interpretation of what is meant by coding standards. In our experience, many in-house standards often are effectively just style, layout or naming rules

and guidelines. These conventions are important for maintainability however there is very little empirical data that any one particular convention has a superior impact on code integrity.

Portability
Portable code can handle a range of environment changes, including machine architecture (data sizes & alignments), compiler versions with improved language compliance, or different target compilers and libraries with different behaviors. Similar to language usage, only sophisticated static analysis can adequately guard against the range of portability issues that may be encountered.

(B) Inspection Process

The elements of an Inspection Criteria must be set within the process and broader context of activities surrounding inspection events. The success of modern day Code Inspections requires a process that encourages all stakeholders within the development team to continually improve the quality of the system; not just improving the code, but continually improving the process of developing the code [22].

PRQA has observed much success within organizations that embrace these concepts, most notably within the automotive industry, where inspection forms a core philosophy of the engineering process. Experience has taught us six key aspects of the inspection process that are essential to its success.

1. Continuous Application
Today, incremental integration of ongoing development is key to successful software builds, and today we are seeing the same principals of "early and often" defect discovery be rigorously applied to testing and integration. We are also seeing rapid advancement in automated build systems that facilitate Continuous Integration.

Code Inspections must also keep up. Periodic snapshots of software builds can supply seamless integration to a quality system, and enable ongoing change-based peer review. Recognition must be given to legacy source code that has attributes of stability and robustness, and a mature tested background. A baseline analysis approach means that the Code Inspection criteria can be applied to only modified code, thus streamlining the continuous aspect of the process.

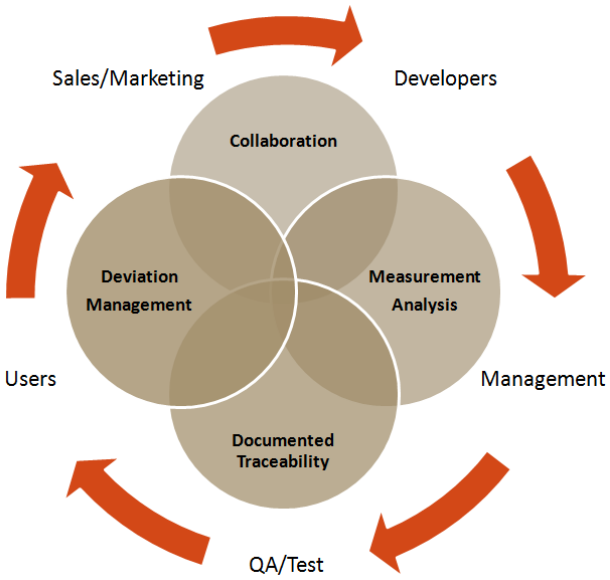

Figure 3: Inspection Process

2. Stakeholder Involvement
All stakeholders from developers to QA and test engineers, to management, sales & marketing, to the users themselves, have separate and overlapping needs and goals. Developers struggle daily with quality issues on source code, yet managers need a meaningful gauge on the structural quality of the code. In fact, all stakeholders can benefit from an x-ray insight into the quality of the system as it is being developed, as opposed to discovering quality issues during later testing phases or even worse through detection by customers.

3. Collaboration
Dr. Edwards Deming, the father of the quality movement in product development discussed in his seminal work [23] that management must encourage their team to build quality into products in the first place; and this can only be done collaboratively within the organization by trained and experience workers adept in both social interaction and the use of statistical methods. This approach is applicable to the collaborative decisions related to all quality issues including those at the source code level.

4. Analysis of Measurements
From the analysis of over 10,000 software projects since 1976, we have learned something fundamental about software development: Quality is not only free [24], but with a continuous improvement process there is

tremendous payback. It's not just about continually finding and fixing defects early and often in the product, it's about improving the process that created the defects in the first place. However you can't improve what you can't measure.

Building on metrics referencing quality attributes, the "holy grail" is to develop advanced metrics that more accurately anticipate software quality as it is being developed. Analysis based on verification measurements of each classification within each Code Inspection criteria is one promising breakthrough here. Since rigorous Code Inspection is a systematic process with an extensive list of well-defined criteria, its implementation can be quantified and governed by continuous measurement of *all* the characteristics that define high quality.

5. Deviation Principle

One aspect of collaboration that is particularly important to practical software development is the *deviation* – a decision which recognizes that there is an "exception to the rule".

Ideally code is developed 100% compliant to every rule in a coding standard. The reality is different. Let's say an organization has a coding standard rule that defends against divide by zeros. Yet a function (Figure 4) violates the rule. The developer argues "we have to live with the possibility for efficiency sake"…ok, so the review team accept the reasoning. The Code Inspection process should manage and document this collaborative decision so that it is preserved into future review sessions and properly accounted for within overall code quality reports.

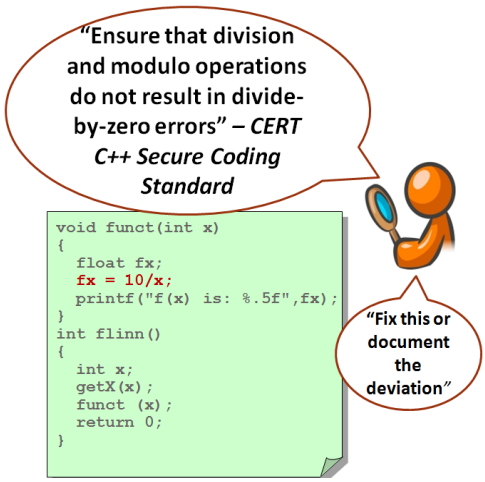

Figure 4: Deviation Management

6. Documented Traceability

For proper traceability, periodic build-level snapshot histories of software quality must be captured and presented within artifacts that can be easily retrieved and audited. This includes all inspection results from automated analysis, all relevant collaborative decisions and discussion commentary, deviations applied, and trend analysis of all chosen metrics over the collection of stored snapshots.

To summarize these six key aspects, the structural quality of a system is raised when all stakeholders can *"see to it early and often"*. Continuous Code Inspection criteria and its process provide the foundation for this. All that's left is a technology that makes it achievable.

(C) Enabling Technology (C²IT)

With rigorous enforcement of measurable criteria and statistical control over process, Continuous Code Inspection Technology (C²IT) is the most powerful means we know to achieve high quality in embedded systems software. It's the right technology that makes it practical. As shown in Figure 5 there are three enabling technologies necessary to make the Code Inspection viable in today's high-integrity systems development.

When integrated in the right way, these technologies create a seamless process that removes many of the burdens of Code Inspections while enabling a truly effective continuous improvement system. The following explores each of these key enablers.

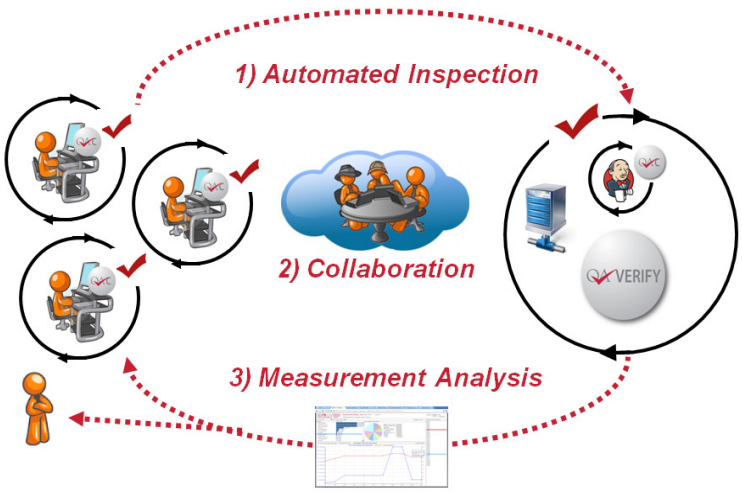

Figure 5: C²IT System Overview

1) Automated Inspection

Automated Inspection can now be performed by a new generation of static analysis technology that extensively enforces Code Inspection criteria with precision and scale such as PRQA's static analysis tools QA·C & QA·C++. Alongside inspection on the desktop before check-in, inspection is also performed at the project integration level, freeing up engineering resources to perform more productive peer review as well as more timely testing operations.

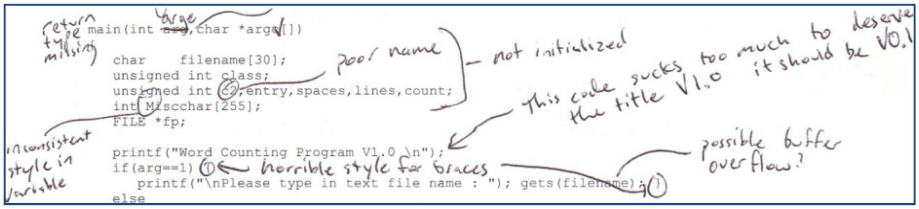

Figure 6: Manual Ad-hoc Code Review

So instead of the chaos of unbridled code review, see Figure 6, the developer works with an integrated and fully automated Code Inspection solution, whether using command line tools, within an IDE, or using a browser interface built for rich interactions with the "C/C++ expert in a box".

2) Collaboration System

With the right combination of features, an online collaboration platform hosted within the development environment implements the second part of Continuous Code Inspection Technology, PRQA's QA·Verify. This platform provides a widely accessible view (via a web browser for example, Figure 7) of the "marked-up" source code from the Code Inspection – in other words, with the output of the Automated Coding Inspection tool – we now have a centralized location for the defects discovered by the inspection to be reviewed, discussed and acted upon as necessary.

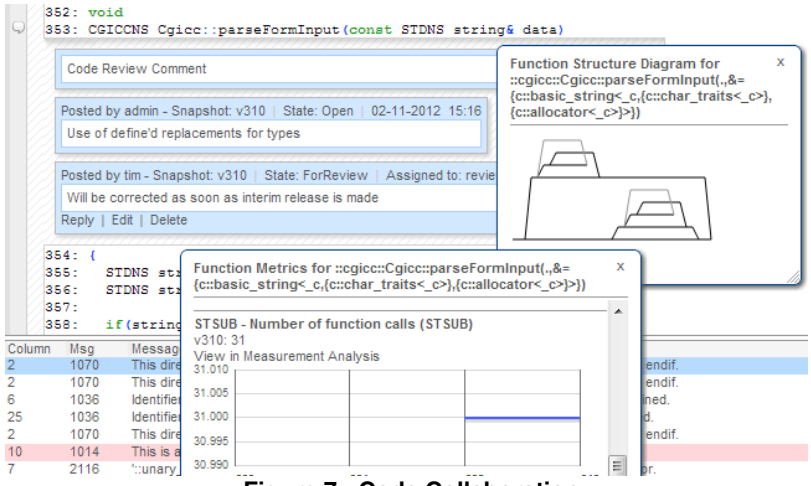

Figure 7 - Code Collaboration

The QA·Verify platform also provides integration with a version control system, the ability to assign action items against defects, notify users of their action items, and track deviations deemed necessary by reviewers - a powerful mechanism to perform Code Inspection follow-up tasks.

3) Analysis Metrics

To meet the goal of process improvement through valid and pertinent measurements, it is necessary to compute an aggregate of both language and process metrics across all historical snapshots of build software quality data.

These quality measures and trend graphs, Figure 8, can then be presented and displayed at various points of user interaction satisfying the goal of advanced Code Inspection and supplying important analytics to process improvement efforts.

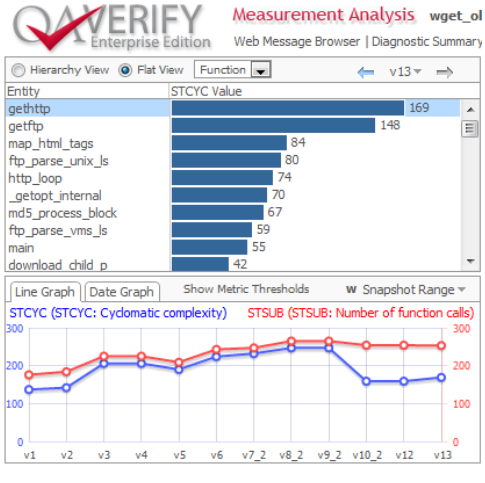

Figure 8 - Measurement Analysis

Trend analysis, both for aggregate data at the top-level of a project and detailed granular function levels, is a kind of "holy grail" for quality measurement systems. The feedback it provides into the code implementation process and automatic inspection, and for further refinement of the inspection criteria is unique and invaluable.

Conclusion

The harsh reality is that a very high percentage of software development projects overrun and contain significant material defects. Of course there are many mitigating factors - code bases are growing, complexity is increasing, product cycle times are reducing - but the fact remains that many of us are still suffering from a software crisis, and the desire for a "silver bullet" is as strong as ever. There may not be this silver bullet we wish for, but as we have shown in this paper there is a next best thing, the *Continuous Code Inspection*. We have also identified some key areas where Code Inspections can be more effective - in particular the inspection criteria, the inspection process and the enabling technology that allows structural quality of a system during development to be transparent, where all stakeholders can ***"see to it"*** early and often.

References

[1] B. Fitzgerald, "Software Crisis 2.0" IEEE Computer, April 2012.

[2] Jones, Applied Software Measurement Global Analysis of Productivity and Quality, 2008.

[3] VDC Research Group, "The Increasing Value and Complexity Of Software Call For The Reevaluation Of Development And Testing Practices" 2011.

[4] J. Frederick P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering" 1986.

[5] M. Fagan, "Design and Code inspections to reduce errors in program development" IBM Systems Journal, 1976.

[6] J. Ganssle, "The Embbeded Muse" http://www.ganssle.com/spcl_reports.htm, August 2009.

[7] Jones, Software Defect-Removal Efficiency, 1998.

[8] Shull, What We Have Learned About Fighting Defects, 2002.

[9] T. a. G. D. Gilb, Software Inspection. Addison-Wesley Publishing, 1993.

[10] O. Laitenberger, "A Survey of Software Inspection Technologies" Handbook on Software Engineering and Knowledge Engineering, 2002.

[11] J. I. ,. L. H. Ilkka Tervonen, "Software Inspection - a Blend of Discipline and Flexibility" Project Control for 2000 and Beyond, 1999.

[12] K. E. Wiegers, Peer Reviews in Software: A Practical Guide, 2001.

[13] ISO/IEC, "9126-1:2001: Software engineering -- Product quality -- Part 1: Quality model".

[14] S. McConnel, "Code Complete: A Practical Handbook of Software Construction, Second Edition [Paperback]," 2009.

[15] Motor Industry Research Association, "MISRA-C:2004 - Guidelines for the use of the C language in critical systems" Motor Industry Research Association, 2004.

[16] Motor Industry Research Association, "MISRA-C++:2008 - Guidelines for the use of the C language in critical systems" Motor Industry Research Association, 2008.

[17] Programming Research, "High Integrity C++ Coding Standard" [Online]. Available: http://www.codingstandard.com/.

[18] S. E. Institute, "CERT Secure Coding Standards" [Online]. Available: https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards.

[19] Lockheed Martin Corporation, "Joint Strike Fighter Air Vehicle C++ Coding Standards" 2005.

[20] IEEE, "IEEE Std 1061 - Standard for a Software Quality Metrics Methodology" 1998.

[21] W. Basalaj, "How good is your compiler (at finding coding defects)?" Military Embedded Systems, January 2010.

[22] B. V. Craig Larman, Scaling Lean & Agile Development, Pearson Education Inc., 2009.

[23] W. E. Dr. Deming, Out Of Crisis, MIT-CAES, 1982.

[24] Crosby, "Quality Is Free" 1980.

About PRQA

Established in 1985, PRQA is recognized throughout the industry as a pioneer in static analysis, championing automated coding standard inspection and defect detection, delivering its expertise through industry-leading software inspection and standards enforcement technology.

PRQA has offices globally and offers worldwide customer support. Visit our website to find details of your local representative.

Email: info@programmingresearch.com
Web: www.programmingresearch.com