PRQA Programming Research

WHITEPAPER

Supplied by the PRQA UK distributor

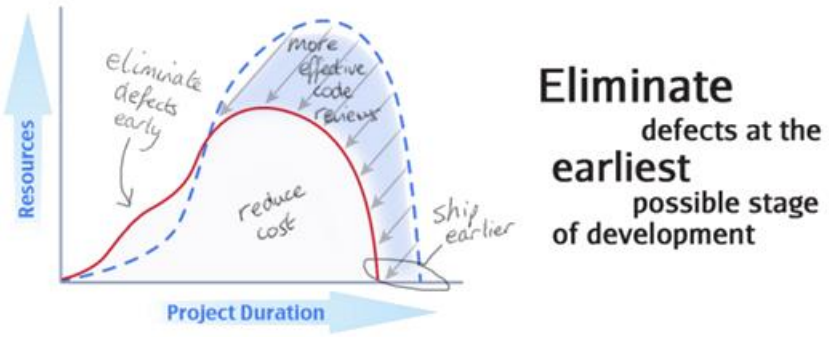
PhaedruS SystemS

www.phaedsys.com
info@phaedsys.com

Return on Investment

A business case for Static Analysis

December 2014



A Return on Investment (RoI) calculation is often complex due to the number of parameters involved and the diverse nature of these parameters. Some of these drivers are relatively easy to identify and others are harder to grasp and more difficult to quantify. In this whitepaper we identify and discuss 10 key drivers that impact the optimization of the RoI in relation to static analysis. We also generate scenarios based on the analysis code from a real open source project to help to demonstrate and quantify the potential impact on the RoI.

Introduction

Detailed RoI calculations are complex, and need to take account of many factors such as the timing of the cost/revenue streams, the timescale under consideration, the accuracy of forecasts, perceived risks, cost of capital, and so on. However, simplistically the RoI boils down to **two primary constituents**: **the costs and the revenues**.

In this whitepaper our primary focus is on the Software Development Life-Cycle (SDLC), and we look at **how common SDLC parameters** (and in particular those related to static analysis) **impact the RoI**. It is worth noting in advance that this **SDLC** dialogue relates predominantly to the **cost side** of the RoI equation. However, ultimately these drivers also clearly have the potential to enable subsequent incremental revenue streams.

First, we identify **10 key drivers** and discuss how these impact the RoI. Then, we examine the static analysis output from an open source project and use this data to provide help to substantiate the impact of these drivers.

A) KEY DRIVERS

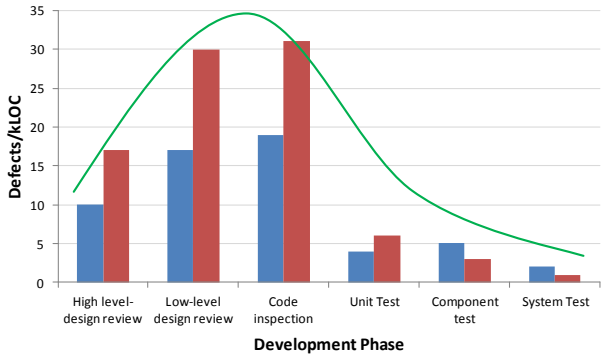
1 - Defects lifecycle

One of the most obvious and fundamental RoI drivers is the **characterization of the injection and subsequent removal of defects**.

Much research has been conducted on this subject [1] [2] [3]. For example an empirical study conducted across several projects from various service-based and product-based organizations [2] reveals a typical distribution:

SDLC Phase	% of defects introduced
Requirements	50% to 60%
Design	15% to 30%
Implementation	10% to 20%
Other (e.g. bad fixes)	Up to 20%

The defect introduction and removal pattern is usually modeled as a Rayleigh distribution [4] [5]:



It is also a universally accepted principle that the time (and, therefore, the cost) to resolve a defect increases dramatically when the fix is deferred to later in the SDLC [1] [6] [7]. The table below provides a summary of some recent research [6] and this data is representative of the escalating costs as identified by the broader body of research:

SDLC Phase	Cost to resolve defect
Requirements	1x
Design	3-6x
Coding	10x
Development	15-40x
Acceptance	30-70x
Operations	40-1000x

In deliberately simplistic terms, **the most effective approach is to a) find ways to stop introducing defects in first place, b) find ways to move up the resolution of defects** to push the apex of the Rayleigh curve leftwards and downwards.

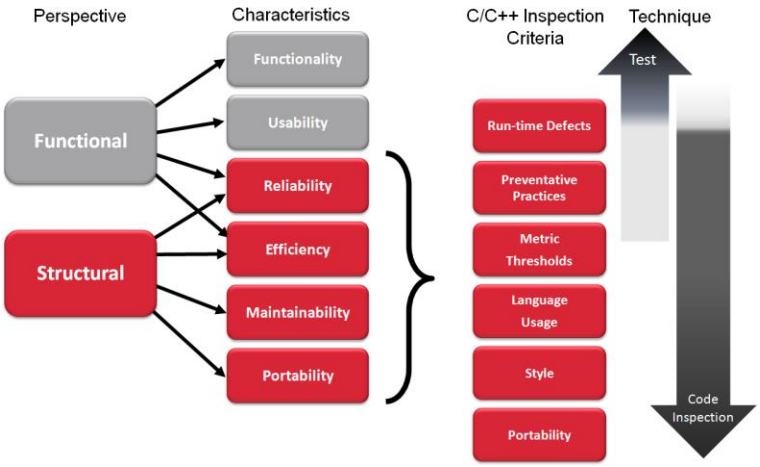
In many respects, this defect injection and removal profile is *the key* driver behind the RoI. However, this observation is very much made at a high level. It is necessary to drill down into the underlying drivers and the root causes and to really understand how these impact the RoI.

2 - Testing

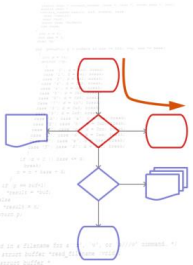
When defects do escape (especially to the field), the first reaction is to blame inaccurate, inefficient or incomplete testing. This behavior is founded on the widespread myth that testing by itself provides sufficient evidence for high integrity and quality.

Of course, testing is crucial, but what many people fail to appreciate is that **testing is not a panacea**:

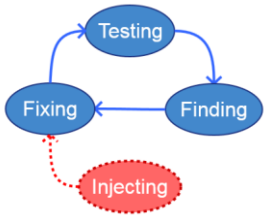
- **Testing is predominantly associated with functionality**. There is a broad consensus in software community that it is not adequate for a software system to meet the expected functional requirements (FRs – describe the behaviors required to support the user's need) without considering non-functional or structural requirements (NFRs – identify the attributes that address the internal integrity of the system). A useful model, which outlines the differences, is provided by ISO/IEC – 9126 [8]. There is some overlap between functional and structural requirements, for example, where functional requirements identify specific Reliability and Efficiency goals expected from the user's external view [9]. **The relevance of this to the RoI relates to the fact that, 1) not only FRs need considered, and, 2) when individual FRs and NFRs are considered these can be tested/verified at different stages in the SDLC. Therefore, there is a very real possibility to pull some of these forward from testing to development.**



- **100% test coverage is not realistically achievable**. There are several reasons for this. Firstly, it is not feasible or practical to generate test cases to cover every conceivable eventuality, to execute every specific statement, branch, condition and edge case [10]. Investing in increasingly granular test cases becomes a law of diminishing returns. Secondly, it's almost impossible to have complete coverage in the case of infeasible paths, dead code or defensive programming – complexity tends to grow exponentially and so do the number of test cases required [10]. It is worth noting that in contrast to this **static analysis has the potential to cover 100% of the code**. And also that **dataflow analysis**, which models the run-time behavior of a program, **provides a very effective mechanism to help to identify different classes of problems** (e.g. invariant/redundant operations, dereferencing of NULL pointers, overflow / wraparound conditions etc.)



- **Testing is expensive**: The rework required to fix late-detected software defects is one of the largest single components of their development cost [11] [12]. Defect detection invariably requires numerous iterations of finding/ fixing issues followed by re-integration (**which, in turn, can and will introduce new defects**), and as the detection time slips to later stages of the lifecycle, costs increase.

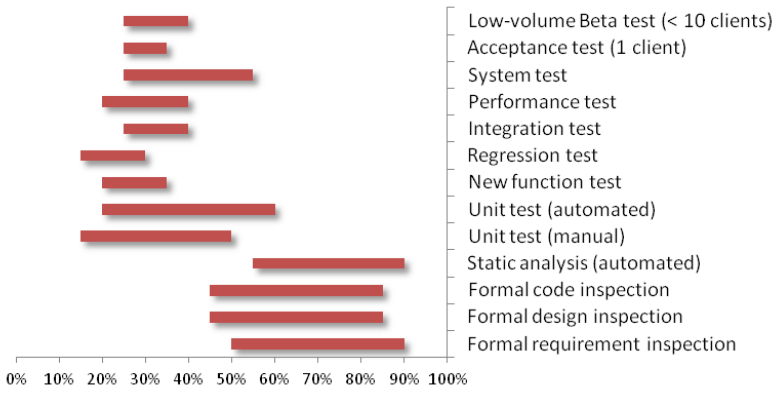


- **Testing extends delivery times**. In the most simplistic scenario, a defect identified in testing needs to be passed back to development, fixed, reintegrated and retested. This cycle can **seriously impact delivery times**. It also introduces **risk** and **uncertainty** - we have all experienced projects that are ready to ship "next week" and then suddenly slip to "next month". In practice, this scenario is usually too simplistic, typically additional test-fix-test iterations are required, for example:
  - when the initial fix from development does not resolve the issue
  - when the regression testing discovers that the initial fix has injected new defects

  Studies of safety-critical, embedded systems have shown that the rework required to fix late-detected software defects is one of the largest single components of their development cost and schedule [11] [12].

- **The structural quality of the code entering testing is important**. Thus, for example:
  - As code becomes more complex, it becomes more difficult to test. Linear increases in complexity place an exponential burden on testing. Thus, controlling complexity during the design/coding phase (for example, via cyclomatic complexity metrics, function structure diagrams, etc.) can have a major impact on the scope/cost of testing.
  - When testers are focusing primarily on functionality, they do not want to be distracted by underlying structural issues, or have these mask the functional defects.
  - Resolving a bug found in well structured code is a completely different proposition to doing the same in a chaotic, inhomogeneous code base.

- **Testing uses critical experienced resources** – This activity requires proper training and rich experience in order to plan and execute effectively; even with automation, verifying functionality is painstaking work; when it is mired in code structural defects and associated repairs chaos ensues. [9]



So, testing remains extremely important, but also has significant limitations. The table here on the left provides a good summary of the range of defect removal efficiency levels for a variety of reviews, inspections, static analysis and several kinds of test stages. As shown, most forms of testing are less than 35% efficient in finding bugs or defects [13].

Testing clearly does not sit in a vacuum, and has huge inter-dependencies with development and the use of static analysis. **The RoI for static analysis cannot be considered in isolation**, but also needs to take account of other stages in the SDLC. In particular optimizing the static analysis RoI should also involve two distinct challenges related to testing:

- **pulling the detection of defects forward from testing to coding**, (a process sometimes referred to as ***development testing***), noting also that this paradigm shifts part of the accountability for final software quality from QA to development
- **making testing more effective**, by striking the appropriate balance between functional and non-functional testing, maintaining more stable, cohesive and well-structured code base (*spaghetti code* – that is code with a tangled and convoluted structure, often result of legacy code modified during the years - will require more time to be maintained, it will break more easily and it will more likely produce unintended effects, including delays due to additional regression testing).

3 – Code reuse

In most commercial software releases the percentage of code that is freshly created, in house, from scratch, is generally very low - in reality the majority of code is existing code which is being reused.

There are many good legitimate reasons for code reuse, and clearly one of the key incentives is to improve the RoI. However, we note that there are many different flavors of "existing code" and it is extremely important to understand that each comes with their own sets of pros and cons. As companies start to rely on code that is generated outside the perimeter of their core controlled development process, they need to be conscious of these different characteristics and the broader cost implication. We highlight five key categories of code below:

a) Legacy
In principle, legacy code falls squarely under the company's core development process. However, experience shows that this code will typically have evolved over many years in a way that is inconsistent and no longer aligned with today's processes and best practices. The appropriate versions of the key development tools are no longer supported and often the developers who created the code have moved on (to different projects or different companies).

The supposition that all legacy code is "proven-in-the-field" and, therefore, automatically suitable to be re-used, (for example, on different target hardware or within a different system), also needs challenged, especially with relation to safety related systems.

Clearly static analysis tools and coding standards have a significant role to play in providing development teams with insights into the structural integrity of legacy code, and also in terms of proving mechanisms to update this code to comply with today's coding standards and best practices. The potential to impact the RoI (and to reduce the risk) is clear.

b) Open Source software (free & commercial)
Open source code and libraries are similar to legacy code in the important respect that the source code is available to the development team. Again, there are clear commercial incentives to reuse this code, but the big question this time relates to the code's uncertain provenance (and ongoing support). Different vertical markets have even coined their own terminologies as they work to develop frameworks which enable them to leverage these types of code (e.g. in the medical domain IEC 62304 identifies *Software Of Unknown Pedigree* (SOUP) - and A&D recognizes *Commercial Off The Shelf* (COTS).

The fact that the source code is accessible again creates an immediate opportunity for static analysis tools and coding standards to contribute positively to the RoI.

c) Auto-generated Code
The verification of the auto-generated code leverages the reliability of the automated code generator and is also qualified-through-use based on previously deployed codebases. Again there are opportunities to apply static analysis, the RoI being very dependent on tool's ability to identify and selectively evaluate handcrafted code.

One interesting approach adopted by some teams with relatively small codebases is to use MDD for design, visualization and modeling – but not to auto-generate code. They have come to the conclusion that it is more cost effective to hand generate and verify these relatively small sections of code [14].

d) Outsourcing

The trend for software outsourcing continues to grow. There are good reasons for this including [15]:

- addressing staff shortages
- reducing labor costs
- specialized skills availability
- round-the-clock workflow.

Outsourcing comes in many guises. Sometimes the scope is limited to the software and sometimes the software is bundled with the hardware. Take for example the automotive industry where the OEMs' supply chain is deep and extensive. In this case the OEMs typically demand that the suppliers develop code which is MISRA compliant and also ask them to provide evidence of this compliance (often even requesting that this is done via QA·C). This provides the OEMs with a guarantee of the structural integrity of the software.

One crucial milestone is the point at which the outsourcer completes their development and this is signed-off and handed back to the vendor, or indeed whoever is responsible for ongoing maintenance. If the maintenance contract is outsourced (e.g. the client pays – say - 20% per year for support, bug fixes, and minor enhancements), **it will be extremely profitable for the provider if the code is high quality and easy to understand and maintain.** Equally, if the provider inherits code, which is structurally poor (spaghetti), the cost of maintenance and defects resolution can be extremely high (as the risk is).

4 – Lifecycle models

As already noted, identifying and helping to fix defects as soon as possible after they arise and as early as possible in the SDLC process is fundamental to static analysis and has a major impact on the RoI. **Early and often are also fundamental tenets in incremental/iterative development approaches**, such as Agile and Continuous Integration/Continuous Delivery.

Agile advocates short (typically 4-8 week) sprints, with the additional objective that the code is fit to release at the end of each sprint. Consequently, the code needs to be kept in a consistently good state and defects, therefore, *have* to be removed soon after their introduction. Procrastination is not an option.

Similarly with Continuous Delivery, the objective here is also to keep the code in a consistently good state and automated checks as are typically performed on every check-in. Therefore, it is not surprising that static analysis integrates extremely well into these processes – **the fact that these implementations dictate that the analysis is performed early and often, tends to improve the static analysis RoI.**

In comparison, traditional sequential models, like Waterfall or V-model, **have a greater potential to delay the discovery of defects**, and this is especially true if it is assumed that many of the defects injected in the coding phase will remain dormant until discovered later in the testing phase. And in these sequential processes considerable time can pass between coding and testing. Under this regime, as previously discussed, maximizing the static analysis RoI means pulling the defect resolution forward from testing to development.

5 - Automation

Automation is clearly one of the key efficiency drivers and is a relevant consideration for practically every process in the SDLC and across the complete development tool-chain.

The automation of static analysis is especially pertinent from two perspectives. The first, relates to the integration into the different SDLC processes (as discussed above), and the second, relates to the effectiveness of the tool in analyzing the source code (and this is discussed in more detail in the next section).

However, automation - in its own right - delivers many benefits which are particularly pertinent to the RoI, for example:

- **scalability** - the ability to easily increase the volume of analysis
- **speed** - dramatically improving duration of the analysis
- **consistent and predictable** - an automated tool will deliver the same results every time. Individuals, with different levels of expertise, will not.
- **independent** - the output from a tool will be objective, different individuals have different perspectives
- **precision & accuracy** – often delivering a higher quality of output
- **traceability** – automated process tend to leave documented (audit) trails that can be used as evidence of enforcement or verification
- **critical expert resources** – can be freed from mundane repetitive tasks and reassigned to value-added activities.

6 – The effectiveness of the static analysis tool

Unfortunately, **the great deal of benefit originating from the adoption of automated processes can be seriously jeopardized by the adoption of ineffective static analysis tools.**

False positives

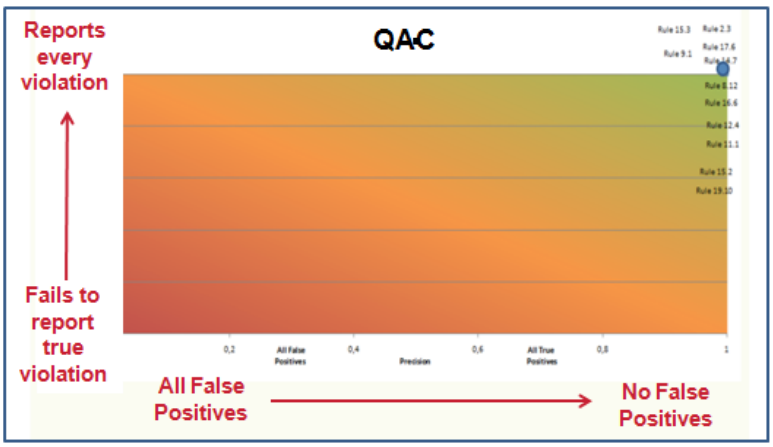
The obvious impact of a high number of false positives in the analysis results is the extra time and cost required for developers to assess, identify and eliminate these [16] [17].

This may not be a major issue if a limited number of rules are being run on a small sample of code. However, as the number of rules and code size increases, the implications (extra cost and time) of using a noisy tool become very apparent. Furthermore, note the very real danger that many false positives will ultimately destroy the credibility of a tool and result in poor adoption by the development teams.

False negatives

While false positives impact the overall productivity of the operator, **false negatives can be way more dangerous, because they lead to a false sense of confidence** [18]. This risk is explicitly highlighted in many standards including DO-178, IEC 61508 (and family) and MISRA.

Introducing the definition of **precision** as the percentage of true positives on the total detected defects, and **accuracy (or recall)** the percentage of correct classifications (true positives and true negatives), **the best static analysis tools rate the highest precisions and the highest accuracies** [16]. This topic is nicely summarized by showing the output of independent research by TERALabs [19] [20] into the effectiveness of a range static analysis tools in enforcing 11 key MISRA C rules. In the case of the performance of QA·C was exemplary - identifying all rule violations, no false positives.



One further consideration - somewhat linked to the topic of false negatives - is to get a clear understanding as to the scope of the tool and what categories of defects are detected. For example, in relation to coding standards, what is the rule coverage? Modern coding standards like MISRA C:2012 are specifically designed to make a clear distinction between those areas that can be verified using static analysis tools (*rules*) and those which require manual interaction (*directives*). *Beware of any tools that purport to have 100% coverage!*

7 – Certification requirements

Modern safety-critical projects are typically needed to adhere to modern quality standard. Whether it is in automotive, avionics, defense, medical, rail, nuclear and so on, there are industry-recognized standards (ISO 26262, DO-178B/C, IEC 62304, EN 50128, IEC 60880) that the software has to comply with. Tools deployed within these projects need be fit-for-purpose for the user to achieve such compliancy. In some cases – like in DO-178B and in the most recent DO-178C with its supplement DO330 – there is the formal requirement to use qualified tools.

Clearly, the RoI is impacted by the availability of certified / qualified tools, and if this burden is placed on the user (and/or the user opts to do the certification process on their own) this can result in significantly increased costs and uncertainty for the project.

8 – Coding standard support

One of the main purposes behind **coding standards** is to define a safer and more deterministic sub-set of the chosen language, encouraging best practices, eliminating situations where undefined behaviors can arise and helping to identify coding errors (e.g. initializations, name-hidings, unreachable code, etc).

Whether an **widely-recognized** standard is adopted (such as MISRA, JSF or HIC++), or alternatively an **in-house** standard, (the latter possibly derived from the former with the integration some **customized rules**, like dedicated naming conventions), a static analysis tool has to provide full support to coding standards enforcement; availability and customizability are key values to select among different products.

Most developers, whether novice or experienced, recognize the fact that coding standards have been created by teams of acknowledged experts who have an extremely deep understanding of their programming language and know the potential risks and pitfalls. What's more, most are available for a token price or free. The bulk of the costs (and potential cost savings) relate to the effective implementation of these best practices. Consequently, it is much more effective, from a technical perspective as well as from a cost perspective to leverage these rule-sets rather than trying to "invent" new rules from scratch. Indeed, it is worth noting the reaction of auditors, they are immediately reassured when they see recognized coding standards being adopted, and are very keen to scrutinize the justifications for deviations and "alternative" rules that have been created in-house.

9 - The Development perspective – beyond the cost

As flagged up in the introduction, the examples above tend to focus on **efficiency improvements and cost savings,** and relate primarily to development activities and the SDLC process. However, the benefits to the engineering team are not limited solely to cost savings. Per the traditional Project Management triangle, there are also associated improvements in relation to delivery times, quality (and predictability), for example:

- **quicker delivery times** and shorter intervals between consecutive deliveries
- increased confidence in **quality of code – internal and external**
- **safer** adoption of **open source** modules
- stronger acceptance gateways for **outsourced** activities
- easier support and maintenance operations
- improved delivery **predictability.**

10 – The Business perspective – the revenue side

From the business perspective, there are clear benefits in improving the efficiency of engineering – and delivering more for every $ spent. However, the incremental **revenue – increased unit sales and/or higher**

prices – will potentially have an even greater impact on the RoI (and the P&L). And clearly it is the output from development that enables for this incremental revenue.

The primarily aim of this whitepaper has been to cover the SDLC/ development side. Therefore, we will resist diving deeper into the business and revenue scenarios. However, the list below will help to identify some specific opportunities for further cost reductions and improved revenues for business operations as a whole.

Cost related items:

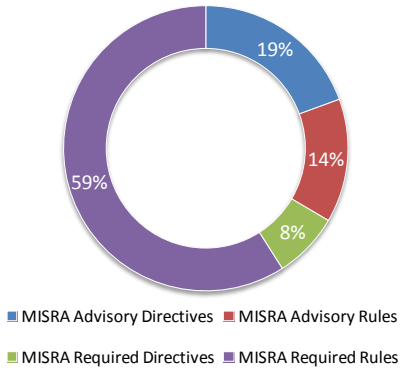
- reduce **development** cost
- reduce **support** and **maintenance** cost
- reduce **warranty** cost
- reduce cost of product recalls
- reduce legal liabilities /cost

Revenue related items:

- reduce **time-to-market**
- improve **sales volumes**
- improve **quality**
- increase **prices**
- quicker to respond to **market opportunities**
- quicker to respond to **competition**
- improved brand **equity / reputation**
- improved **predictability**, lower risk.

B) ROI ANALYSIS – AN EXAMPLE

To test the ideas discussed so far, we selected a small open-source project and looked to remove all MISRA C:2012 related diagnostics. The selected project was **time-1.7** (the GNU version of the utility that measures a program's usage of resources, such as time and CPU). This release is composed of **6 header and 5 source files** and roughly **2.000 lines of code**.

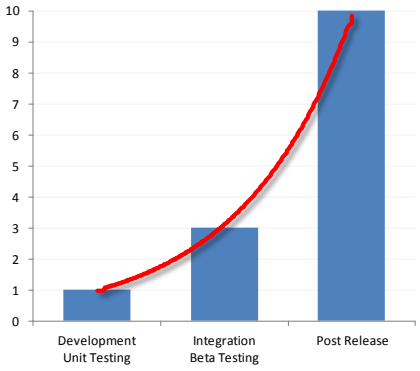


The code was analyzed using **PRQA QA·C v.8.2 with MISRA C:2012 compliance module**. Using this setup, the tool detected a total of **520** diagnostics.

In the scenarios below it is assumed that we resolve all these violations.

With respect to the lifecycle, we have simplified this into to 3 phases:

- Development/ Unit Testing
- Integration/ Beta Testing
- Post Release.



We have assumed that the average time to find and fix a defect is 1 hour during the development phase, increasing to 3 hours during the integration stage and to 10 hours post release. Note, when comparing the outcomes of the scenarios the ratio (1:3:10) is more significant than the absolute values. These find and fix times remain constant throughout all the scenarios below.

Case studies

We implemented **5 different scenarios**, primarily focusing on the sensitivity of the RoI as the % of defects fixed during each phase changes.

Scenario 1 – Reference Case

As a reference case, we based the find and fix distribution roughly on the data from NIST [21], specifically; 30% of defects are fixed during development, 60% in integration and 10% after the release of the product.

Scenario 2 - Good tool/ Development Testing

This scenario reflects the situation where an idealized top-quality static analysis tool is used (no false positives and no false negatives). The detection of the defects is pulled forward from testing to development and all the fixes are implemented by the developers as soon as they are detected, whilst still in the development phase.

Scenario 3 – Good tool / Compliancy in QA

The same top-quality static analysis tool is used in this scenario, however, this time the analysis is run later, at integration/ beta testing phase. This reflects the situation where the compliancy of the code is only assessed after development (i.e. "acceptance testing" by "QA").

Scenario 4 – Lousy tool / False Negatives

Here the static analysis tool is not a top-notch product, and in particular it misses defects (false negatives). We assume that the tool fails to detect 20% of defects and these escape to the field. For simplicity we only consider the extra cost associated with rework and fixing these defects post release. (Of course, having undetected defects in the field will also impact the business revenues, and moreover in safety-critical markets can result in very costly litigation.)

Scenario 5 – Lousy tool / False Positives

Again a low-quality static analysis tool is assumed. In this case the tool captures all true violations but also generates false positives. Therefore, we assume that the tool reports 20% more defects, additional to the 520 identified above. Here we also assume that it takes the same amount of time to fix a true violation as it does to eliminate a false positive. As for scenario 4, all detections and corrections are done during the development stage.

A summary of the key parameters used in each of these scenarios is provided below:

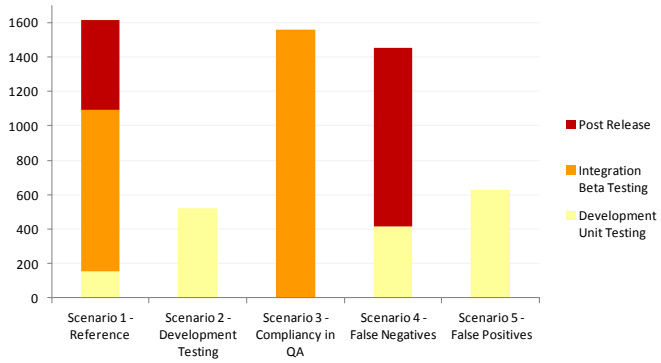
Scenario		SDLC phase			Total
		Development Unit Testing	Integration Beta Testing	Post Release	
1 - Reference	%	30%	60%	10%	*100%*
	diagnostics	156	312	52	*520*
2 - Development Testing	%	100%	0%	0%	*100%*
	diagnostics	520	0	0	*520*
3 - Compliancy in QA	%	0%	100%	0%	*100%*
	diagnostics	0	520	0	*520*
4 - False negatives	%	80%	0%	20%	*100%*
	diagnostics	416	0	104	*520*
5 - False positives	%	120%	0%	0%	*120%*
	diagnostics	624	0	0	*624*
All scenarios	Time to find and fix	1x	3x	10x	

Results

Costs - in terms of time - have been computed for the 5 scenarios. As outlined above, in our simplified model the time required to find and fix each defect is determined solely by the SDLC phase when the fixing activity takes place.

Scenario	Time cost to fix F&F defects @ SDLC phase			Total Time (/cost)
	Development Unit Testing	Integration Beta Testing	Post Release	
1 - Reference	156	936	520	*1.612*
2 - Development Testing	520	0	0	*520*
3 - Compliancy in QA	0	1560	0	*1.560*
4 - False Negatives	416	0	1.040	*1.456*
5 - False Positives	624	0	0	*624*

Scenario 1 shows the worst performance in terms of costs. The main reason for this is the fact that the bulk of the fixing activity is delayed to the later (more expensive) stages of the cycle.



As expected, **scenario 2 has the best outcome as this scenario considers an accurate tool which eliminated defects as early as possible in the SDLC.**

Scenario 3 demonstrates the negative impact on the RoI that results when the use of the tool is postponed from development to integration/ beta testing – a mindset where the codebase is written first and the tool used subsequently to verify compliance. *Despite* having a top-tier tool, the RoI is greatly diminished as the tool is being deployed sub-optimally.

The deterioration of the RoI in scenario 4 is quite apparent. Again it is worth highlighting that this simple model does not take account of any additional damages or litigation costs when severe defects escape to the field (especially in safety-critical systems).

Finally, as anticipated **false positives generate an overhead which creates a proportional increase in costs (scenario 5)**. Thus, when compared to scenario 2, the additional 20% of diagnostics lead to a 20% increase in costs.

Conclusions

In this whitepaper we have identified 10 key drivers which have a material impact on the static analysis RoI, and we have drilled down to explore these at an operational level (e.g. a level at which meaningful actions can be initiated).

The results from the analysis of an open source project have provided an additional insight into the potential quantitative impact on the RoI based on 5 different scenarios.

We note that in practice it can be tricky to calculate the RoI, in particular, due to the fact that it can be difficult to accurately estimate some of the costs and revenues. For example, it is easy to see the price of a tool, but less obvious to see the cost of developers continually reworking code or spending their time chasing false positives. One way to help to be more objective when selecting a static analysis tool, is to **evaluate the performance of the tool on a sample of your own code** in terms of:

1. **Technical ability of the tool (*effectiveness*)**
2. **The resulting RoI**

When our customers evaluate PRQA's solutions, we cover both of these areas.

Glossary

FR	Functional Requirement
F&F	Find & Fix
NFR	Non-functional requirement
P&L	Profit & Loss
QA	Quality Assurance
RoI	Return on Investment
SDLC	Software Development Life-Cycle
TU	Time unit
V&V	Verification and Validation

References

[1] e. a. L. Lazić, “Estimating Cost and Defect Removal Effectiveness in,” in *INFOTEH-JAHORINA, vol. 12*, 2013.

[2] G. N. T. Suma V, “Defect Management Strategies in Software Development,” in *Recent Advances in Technologies*, 2011.

[3] V. S. T.R. Gopalakrishnan Nair, “The Pattern of Software Defects Spanning,” *International Journal of Software Engineering (IJSE),*

[4] A. Vladu, “Software Reliability Prediction Model Using Rayleigh Function,” in *U.P.B. Sci. Bull., Series C, Vol. 73, Iss. 4, 2011*, 2011.

[5] S. H. Kan, “Metrics and Models in Software Quality Engineering,” 2003.

[6] M. C. A.A. Frost, “Advancing Defect Containment to Quantitative Defect Management,” *CrossTalk,* no. December, 2007.

[7] W. Humphrey, “A Personal Commitment to Software Quality,” in *5th European Software Engineering Conference*, 1995.

[8] ISO/IEC, “9126-1:2001: Software engineering -- Product quality -- Part 1: Quality model”.

[9] F. B. C. Weimer, “Continuous Code Inspection,” PRQA White Paper, 2013.

[10] M. Baluda, “Automatic Structural Testing with Abstraction Refinement and Coarsening,” in *European Software Engineering*

[11] R. Bartholomew, “Using Combinatorial Testing to Reduce Software Rework,” *CrossTalk,* no. January/February, 2014.

[12] A. P. S. Sulistyo, “PMG-Pro: A Model-Driver Development Method of Service-Based Applications,” in *15th International SDL Fourm*,

[13] C. Jones, “Software Quality and Software Economics,” *Software Quality, Reliability, and Error Prediction,* vol. 13, no. 1, 2010.

[14] C. Study, *Static Analysis - for Manual Code and Auto-generated Code,* PRQA, 2014.

[15] L. Rierson, Developing Safety Critical Software – A Practical Guide for Aviation Software and DO-178C Compliance, CRC Press,

[16] L. W. S. Heckman, “On Establishing a Benchmark for Evaluating Static Analysis Alert Priorization and Classification Techniques,” in

[17] D. A. J. M. B. F. Wedyan, “The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction,” in

[18] G. M. B. Chess, “Static Analysis for Security,” *IEEE Security & Privacy,* no. November/December 2004 Issue.

[19] H. V. H. K. B. M. Temmerman, “KriCode Research Report I: Comparative Study Of MISRA-C Compliancy Checking Tools,”

[20] PRQA, “Comparative Study Of MISRA-C Compliancy Checking Tools,” [Online]. Available:

[21] R. f. NIST, “The Economic Impact of Inadequate Infrastructure for Software Testing,” 2002. [Online]. Available:

[22] H. Gall, *ReUse: Challenges andBusiness Success,* Advanced Software Engineering, FS 12 - University of Zurich, 2010.

[23] D. M. Mehta, “Effective Software Security Management,” OWASP - Open Web Application Security Project, 2007.

[24] S. Goldfarb, “Industry Metrics for Outsourcing and Vendor Management,” Q/P Management Group Inc., 2008.

[25] R. P. M. S. A. Dautovic, “Automated Quality Defect Detection in Software Development Documents,” in *SQM 2011 - ifth International*

[26] C. P. J. Zou, “Control Cases during the Software Development Life-Cycle,” in *IEEE Congress on Services - Part I*, 2008.

[27] J. C. L. Amar, “Measuring the Benefits of Software Reuse,” [Online]. Available: http://www.drdobbs.com/measuring-the-benefits-of-

[28] C. C. Weber, “Assessing Security Risk In Legacy Systems,” [Online]. Available: https://buildsecurityin.us-cert.gov/articles/best-

[29] F. Bolger, “Controlling automotive software deviations in a MISRA compliance environment,” 2014. [Online]. Available:

[30] F. Bolger, “The Best Coding Standards Eliminate Bugs,” PRQA White Paper, 2011.

[31] V. L. d. Mendonça, “Static Analysis Techniques and Tools: A Systematic Mapping Study,” in *ICSEA 2013 : The Eighth International*

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@phaedsys.com
Web: www.phaedsys.com

Supplied by