mwd
macehiterward-dutton

The Business Value of Software Static Analysis

Bola Rotibi

August 2008

Software can kill. We are all pretty much aware of this fact for the software that controls planes, missiles and medical equipment. We expect the creators of such software to have the highest forms of governance and quality processes in place: a fully-documented software development lifecycle with detailed requirements that are fully connected to every aspect of the software and hardware artefacts, strong static analysis and code reviewing procedures and a foolproof testing framework. After all, lives are at stake.

But what if it is a business that is at stake? After all, livelihoods could certainly be affected. Should we not also expect business- and security-critical software applications to have strong code quality reviewing processes?

This paper has been sponsored by IBM.

Summary

Merely "paying lip service" to software quality in business-critical applications brings significant business risks.

Over the last 25 years many organisations have lacked the desire to engage adequate code analysis tooling and employ more stringent software delivery processes - mainly because the pursuit of software quality has been perceived as both expensive and time consuming.

However it's no longer sufficient to merely "pay lip service" to software quality. Although a broken link on a website or even certain types of web application failures may not appear to pose much of a risk (beyond minor frustration for end-users), an underlying failure in a business-critical application due to the quality of the application (code and design) can easily cost a company more than just money – destroying trust between provider and user.

Increased software complexity from advances in technologies and approaches is set to put further pressure on software quality.

Advances in software technologies and approaches are driving new complexity into the software applications that organisations are looking to build from them. The complexity of delivering software that takes advantage of the features of web 2.0, rich media, Software as a Service (SaaS) models, unified communications, collaboration, virtualisation and Service-Oriented Architecture (SOA), however, comes at a price: increased risk of software defects.

Software technology is advancing at considerable speed, as are user expectations and requirements. The challenge is that while many organisations are attracted by the potential of these new advances, few of them have the software processes and tools in place to offer adequate assurances for the quality of the software code delivered. As a result, they are in danger of promising much, based on the capabilities of new technology, but not being in a position to deliver software code with quality that meets the demands and expectations of users.

Financial savings, improved software quality and maintenance cost reductions all flow from the early identification and removal of software defects.

Employing static analysis strategy and tooling to remove defects in the development phase and earlier is a winning strategy. Firstly, it can considerably lessen the impact that such defects have on the productivity of the software delivery process, the quality of the application, user satisfaction and expectations. And secondly, given the increased costs down the line of any maintenance program where poor-quality code needs to be fixed, it actually reduces costs.

Software: a victim of its own success?

Software matters. It is pervasive in everything that we do in our modern world. Software is a key component for driving business transformation and innovation and engaging effectively with globally distributed teams. Software technology and applications span the entire industry and business landscape, becoming an increasingly important and integral part of the way we work, live, rest and play. Software innovation is seen as a means of differentiation and achieving competitive edge as well as a cornerstone for operational excellence.

When software fails

The flip side to the success and importance of software is the uncomfortable reality that for far too many organisations and ordinary users, software fails. What is more, it does so at a rate that wouldn't be tolerated in most other technology areas. For many, the software delivery process isn't predictable, there is often very little measurement, and there's little ability to pass on knowledge or experiences gained through the process to drive future improvements.

Would any other industry accept the failure rates of the software industry? One only has to see the backlash, public relations disaster and financial losses that occur when the medical, aviation, manufacturing or automotive industries get things wrong. Could you imagine any of these industries, once knowing the bad effects of poor procedures and the cost differential for using correct procedures, continuing to execute procedures that have been proven to fail? But this is exactly what happens in the delivery of software code and applications today.

What is perhaps more shocking, is that the path to software quality has been known for many years. Admittedly tool support has sometimes lagged behind; however the processes, methods and pitfalls have been well documented and easily available for all to read and digest over this previous decade.

The Nirvana of software quality

The benefits of enacting a robust quality process, whether in delivering software or otherwise, have been universally acknowledged with many widely-publicised studies and reports.

Following a well-structured software quality process can deliver significant value and, indirectly, enables tremendous business opportunities. It can achieve a reduction in software delivery costs, and a more efficient deployment of IT resources (because staff used for fixing defects can work on creating and building new and improved products). Importantly, a quality-focused software delivery approach does not necessarily mean a longer time-to-market. In fact much of the available evidence points to the opposite: focusing in the right way on software quality decreases time-to-market.

The ability to deliver more marketable products quickly, repeatedly and predictably translates into significant competitive advantage and the potential for increased profit margins. In addition, raising customer satisfaction and improving software success rates has the knock-on effect of raising staff morale (association with success is much more uplifting than the demoralisation of failure). Efficiency gains in resource utilisation also open up the opportunity for staff to do more varied and interesting work, allowing them to participate more easily in the innovation process.

The reality check

Unfortunately, even though there have been proven case studies demonstrating the benefits outlined above, many organisations simply pay “lip service” to software quality. The effort required to achieve the benefits above is considered by many to be too much to give. Attention to software quality is often dropped in the face of delivery pressures; getting something quickly out to the market that is “just good enough” is the prime goal.

The “good enough” mentality holds great sway in the software industry, and is accepted and even expected in a user audience that is resigned to the constant litany of software failures they experience.

A case for renewed software quality rigour

Poor software quality and the lack of robust software quality processes and tools early in the delivery cycle cannot continue. The times, and more importantly the technologies, are changing. What we are able to do, and want to do, through the latest software technology advances has increased.

Broad tolerance for bad software and poor quality software delivery processes is becoming less acceptable as the role software plays in driving innovation, transformation and entertainment becomes increasingly important. Moreover the complexities of today’s software, boosted by advances in technologies and approaches offering improved ways of reaching out to new clients and collaborating and communicating with existing customers, suppliers and employers, suggests that more will be at stake in the event of software quality failure. The risks will be further compounded by the need for increased security requirements for protecting online, public-facing applications against malicious intent.

“Good enough” is no longer good enough

More serious outcomes of failure

New business models are being founded on applications and systems developed with many of these new technologies and approaches. If organisations start to restructure their working practices around applications and systems which rely on the new generation of communication and collaboration technologies and approaches, then failure due to poor code or application quality becomes even less acceptable.

The inclusion of rich media and visuals; the push for greater collaboration through the Internet; and unified communications for richer interactive social or work activities, means that any failure in such services would not only have the potential of creating higher levels of frustration – it would reduce productivity more sharply. On top of this, company brands become more easily exposed to damage.

More complexity

Increased complexity in coding applications that support such technologies along with the increased complexity of integrating and interoperating such applications with legacy software, hardware and data sources raises the prospect of an increase in the number of code defects. What’s more, development is changing. Different techniques and approaches such as composite application and mash-up development, along with others associated with a service-oriented software environment, have added to the complexity of the development and integration strategies of past software environments.

Tools are changing, too

Change is also happening in the supply market. Independent Software Vendors (ISVs) and tools vendor are increasingly embracing variations on a model-driven approach. This allows them to bring together all the various parties in software development and create an integrated Application Lifecycle Management (ALM) environment where there is substantive data flow between the parties. As we move to smaller, agile, software development, software components are intended to be self contained but become part of a larger assembly kit. This, in reality, is the first real opportunity to show the object models of the mid-80’s finally in place. This is a world where software solutions are simply assemblies of many small components.

All this brings with it a problem. How do you validate the correctness of interactions between potentially hundreds or even thousands of software components? As systems get more complex, you need to be able to see the impact of changing any component. Software complexity means that this cannot be done by test teams alone. In many large development projects it's now practically impossible for any individual to be able to manually assess all potential breeches and problem areas.

Static analysis: achieving quality through early intervention

Amidst all this complexity and risk, improving the quality focus within the software delivery process even by a small but sustainable amount can generate significant returns.

There are a number of key angles to pursue to achieve quality within the software delivery process. One important area where significant improvements in the quality of the software code can be easily achieved is in the process of static analysis. This is because this activity can be automated through tool support at the developer level, or even during the build process; and sophisticated and deep analysis can be applied (using out-of-the-box rules or with the flexibility of custom rules) that no manual peer inspection can achieve in a similar timeframe.

What is static analysis, precisely? We define it, based on Wikipedia's definition, as follows:

Static analysis is the analysis of software that is performed without actually executing programs built from that software. There are many different forms of analysis rules that can be applied – ranging from the behaviour of statements and declarations, through to verifying the properties of software in critical systems or applications – as well as locating potentially vulnerable code.

Making a case for Software Static Analysis

Startling statistics

The power of static analysis has been proven many times over, the results of which can be readily found on the Internet.

- A 1998 study carried out by Capers Jones stated that "wastage" and defect repairs (from deployed code) absorbed almost two thirds of the US software workforce – leaving only one third for productive work on new projects.
- The same study also found that 50% of software development project budgets are spent fixing poor quality code; fewer than 6% of organizations have clearly defined software management processes in place; and software projects of 100,000 function points in size have a failure rate of 65%. More recent articles on this topic suggest that for many organisations the statistics remain very much unchanged today.

The fact that most work effort for the development organisation is spent on defect repair is not only an expensive use of skilled software personnel; it is an inefficient use of less skilled resources. Equally troubling is the fact that it is the combination of high levels of potential defects and low levels of defect removal efficiency that contributes to cancelled projects and to the dominance of error-related work patterns in the software community. It is a worrying indictment that after all these years, more often than not, more than twice as much effort is associated with defect removal than with actual product development.

Preventing defects as they emerge early in the development lifecycle and before they get shipped would clearly save money and allow more software staff to focus on value-adding application development.

Why prevention is more cost-effective than cure

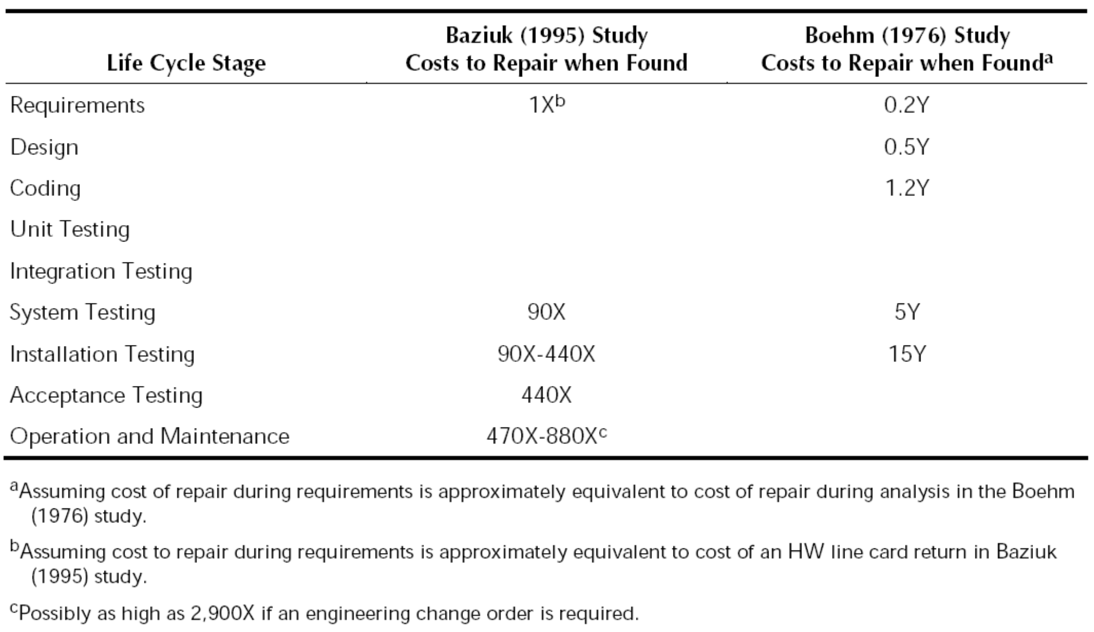
There have been countless studies done over the years that not only prove significant cost savings that can be achieved through static analysis of software code and applications particularly in the definition and coding phases but also the added benefits from improved performance, more efficient utilisation of IT staff and general reduction in downstream bottle necks.

It is not the intention of this report to repeat in detail the findings of such reports or studies, because most of them are easily and freely accessible on the Internet and go into considerably more depth than can be afforded here. However, below we summarise the main findings to demonstrate that prevention is more cost effective than cure.

The 2002 National Institute of Standards and Technology (NIST) report on “The Economic impacts on the inadequate infrastructure for software testing” is one of the most recent detailed studies to ascertain the value of improving software quality – along with the most feasible processes and tooling to put in place and the types of metrics to collect. A detailed analysis of IT organisations within two industry groups was conducted: automotive and aerospace equipment manufacturers; and financial services providers and related electronic communications equipment manufacturers.

An overview of the findings is shown in figures 1 and 2.

Figure 1: Relative Costs to Repair Defects when Found at Different Stages of the Life-Cycle



Life Cycle Stage	Baziuk (1995) Study Costs to Repair when Found	Boehm (1976) Study Costs to Repair when Found[a]
Requirements	1X[b]	0.2Y
Design		0.5Y
Coding		1.2Y
Unit Testing		
Integration Testing		
System Testing	90X	5Y
Installation Testing	90X-440X	15Y
Acceptance Testing	440X	
Operation and Maintenance	470X-880X[c]	

[a]Assuming cost of repair during requirements is approximately equivalent to cost of repair during analysis in the Boehm (1976) study.

[b]Assuming cost to repair during requirements is approximately equivalent to cost of an HW line card return in Baziuk (1995) study.

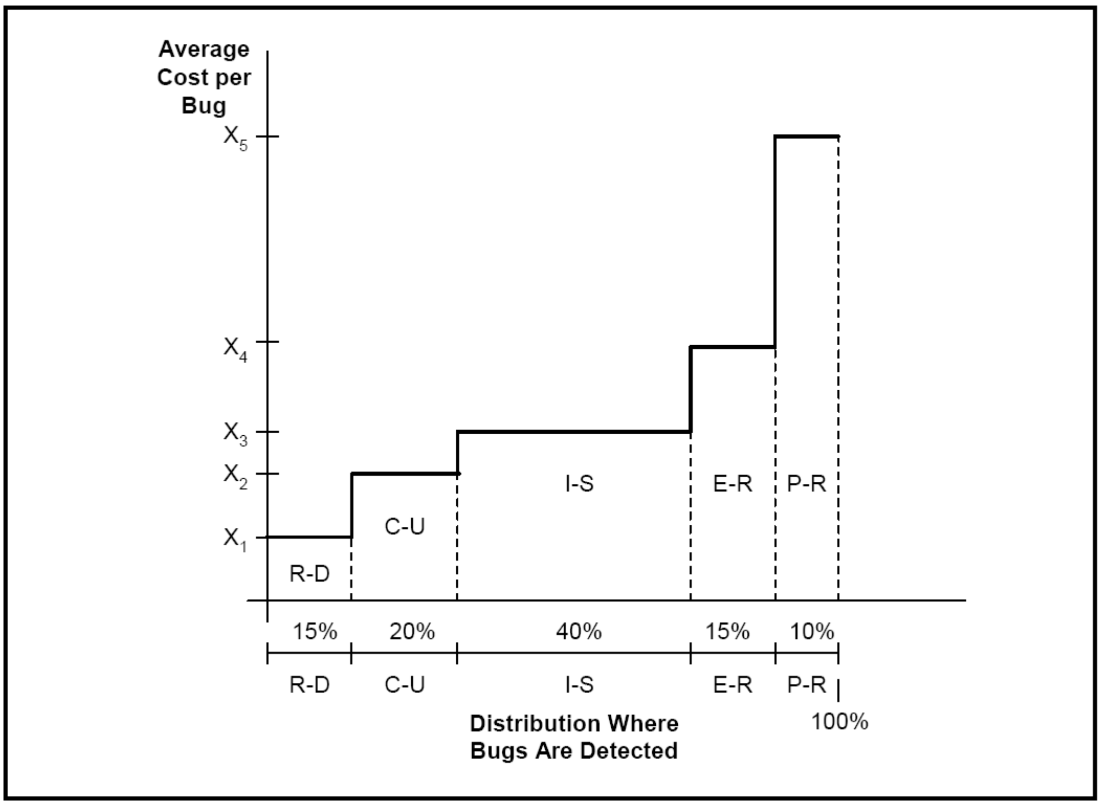
[c]Possibly as high as 2,900X if an engineering change order is required.

[Where X is a unit of cost for the Baziuk 1995 Study and Y is a unit of cost for the Boehm 1976 Study] Source: National Institute of Standards and Technology “Economic impacts on the inadequate infrastructure for software testing”, 2002

What’s worth pointing out is that these two industry sectors probably have some of the highest regard for software quality process as a result of the mission, safety, security and business-critical nature of the software applications they build and deploy. Therefore the results they produced are almost certainly bound to be better than for organisations that have not traditionally needed to develop software to the same standards – meaning that other organisations are more than likely to be experiencing considerably more defects and wasting more resources.

Figure 2: Example Software Testing Costs Shown by Where Bugs Are Detected

"Costs" can be expressed in terms of expenditures or hours of testing time.



Legend:
- R-D: Requirements Gathering and Analysis/Architectural Design
- C-U: Coding/Unit Test
- I-S: Integration and Component/RAISE System Test
- E-R: Early Customer Feedback/Beta Test Programs
- P-R: Post-product Release

Source: National Institute of Standards and Technology "Economic impacts on the inadequate infrastructure for software testing", 2002

The NIST study revealed that software developers reported that better testing tools and methods during software development could reduce installation expenditure by 30 percent. Although the study talked generally about the overall testing infrastructure for software development and maintenance, we believe it's reasonable to deduce that early intervention in removing defects should result in direct cost reduction in the software development process, and also a decrease in support costs.

Restoring faith in software code quality

In order to restore faith in the quality of the software applications that we build, we must first try to understand why software fails. Ask anyone in the industry what makes software fail and you immediately get a barrage of competing reasons – as shown in figure 3.

Figure 3: Common opinions for the reasons for bad software

Poor software programming and unrealistic timeframes:

- Unfair management pressure to shorten development lifecycles and rush applications and systems out of the door.
- Developers not taking enough personal responsibility for their full role in the lifecycle.

Lack of quality control and the wrong incentives:

- Nobody reviews the code if it works.
- Not having management buy-in to create a quality product.
- Defect removal is not very sexy and can be long and labour intensive. No cachet for developers who prefer to be and are rewarded for program functionality and user interfaces.

Scepticism:

- The view that there will always be bad code.

Poor quality staff and processes:

- Poor management with managers either lacking in skills or understanding of the development process and the code/ application being developed.
- Poorly skilled programmers with limited programming languages experience and who fail to learn lessons of the past.
- Cutting corners for things like a good working environment induces a 'don't care' attitude amongst one's developers.
- Not having the right tools for the job or the needed user/business experience.

The increased complexity of software and tools:

- Outsourcing and geographically located teams.
- Many projects are now beyond the scope of a single individual to comprehend: 'minor' changes in one component can't be easily correlated with effects in another area because no one person understands the relationships.

Source: MWD

The above opinions are all valid reasons that will lead to bad software being deployed and eventually causing a failure. However, in the face of strong financial evidence there can be little excuse for the lack of a more committed approach to the early removal of software defects.

There may be barriers and scepticism from the business because IT organisations have not always equipped themselves well in the past. They have made unnecessary and sometimes costly technology purchases while at the same time failing to deliver the expected value to business teams. However, good organisations have come to realise that responsibility and accountability for software code quality lies, to some degree, on both sides of the IT and business management divide.

Practical guidelines for software static analysis

Tools in general have come a long way in helping us to achieve more reliable software. But purchasing a static analysis tool alone will not guarantee software code quality.

As with any strategy that looks to manage or improve the delivery of software code and applications, the focus should fall on a number of common core areas: people, process, methods, tools and technology. Below we briefly highlight important considerations in these different areas.

People

People can be the single biggest obstacle to implementing a static analysis strategy and platform and a wider focus on quality.

Ideally the priority should be to employ the best people with the right attitude. But there are practical considerations: the world has a limited pool of software delivery talent available, and the best people attract significant premiums. Not every company can afford the best. This is one key reason why as well as attempting to hire the best people you can afford, you should also put in place a “quality culture” that is backed by supportive processes, strong motivational drivers and the right incentives, rewards and punitive measures (should performance fail to live up to requirements).

Software quality, and the early resolution of defects, should be goals that are rewarded over and above more obvious measures such as the number of function or feature points supported by the software code. Making this transition requires “top-down” commitment from the management chain, together with empowerment and resolve within the IT team. A clear understanding of individuals’ responsibility and accountability will be essential in driving any changes to the software delivery process. Unrealistic deadlines will make it even harder to succeed, even with a static analysis tool.

Process and methods

In the absence of consistent processes, IT teams fail to learn from past mistakes or successes, ending up repeatedly reinventing the wheel. As a result quality and success is rarely duplicated and the opportunity for predictability is lost.

It is important to employ the right processes (processes that can be customised to meet the specific needs of your organisation) that will help drive quality in software delivery from the outset, whether static analysis is being executed at the developer level or centrally at the build level. For example, carrying out risk analysis at the beginning of a development project might highlight situations to guard against, which can then be used as defect criteria. Support for using best practices for ensuring quality within software development will be vital, as will a process and framework for measuring and reviewing the success of such practices so that further improvements can be discovered and implemented.

Two common software development methods that have been widely reported as being incredibly effective in maximising software quality are:

- Using formal inspections of designs, code and other deliverables to prevent and remove software defects.
- Using software quality assurance groups and software quality process frameworks like Capability Maturity Model® Integration (CMMI) and Six Sigma.

There is also a case for the use of agile development methodologies that embody the concept of short iterations of development and shipping or deploying often to quickly ascertain customers’ needs and acceptance. Reducing the number of function points delivered and having short delivery cycles will help to lessen the load of potential defects – allowing for a more manageable and effective defect removal process.

Tools

With all our above analysis in mind, there are a number of important features that a static analysis tooling platform should encompass. These are highlighted below:

- **User friendly extensibility framework**. Tools should make it easy for customers and partners to implement scenario-specific extensions, providing clear guidelines and productivity features to help the configuration process along with support for multiple methods of integration. The benefits and value of an extensible framework can be substantial. The advantage of having a static analysis solution that is flexible means that it can be customized to meet the specific needs of a company (since every company may do things a bit differently). Having the ability to create and modify rules and reports, as well as having the ability to 'connect' to other 3rd party tools (in order to have one tool/interface in an environment where there could be many tools in-house) allows for greater developer productivity in standardising the interface and methodology of use.
- **Automation**. With developers being tasked with doing more in the development phase, automation – both of processes, functions and of use – becomes really important. You need automated analysis to work through a lot more reviews and rule-checking than a face-to-face peer review meeting can do in the allotted time.
- **Real-time testing support**. This is vital, so that developers can remedy defects as soon as they emerge and where the impact will be less significant.
- **Comprehensive and extensible static analysis rules support**. Outcomes from checking rules should be associated with multiple severity levels, and the tool should be able to automatically provide fixes for violations. Tools should allow rulesets to be altered and extended, and should also help teams share rules and best practices between them.
- **Clean and simple user interface**. Given the time pressures that development staff are under and the challenge to get software quality issues ingrained in the behaviours of all developers, it's important to look for usability features that make it easy for developers with even the most basic skills to quickly pick up and use.
- **Support for multiple views of an application or solution code structure and dependency relationships**. This is particularly useful in understanding existing code structures in the absence of sufficient supporting documentation or code comments. Development teams are often under pressure to deliver and worry less about documentation and commenting their code. If an application code is poorly structured and low in quality then being able to see this would allow decisions to be made as to whether to retain the code or invest in renewing it. At least then the cost/benefit equations can be more easily calculated. There are many tools that provide code introspection models - so while this may be an overlapping feature, its presence in the static analysis tool would allow developers better insight at the most opportune moment (i.e. during the code development stage where it can be changed easily and quickly) into how newly developed code might impact on the overall structure and dependency map.
- **Support for refactoring**. The ability to easily change poor code component structures and designs is important in the context of static analysis. There is no point adding more code to bad quality code because not only does it not improve the overall quality, it potentially increases the complexity too.
- **Delivery workflow and build-time integration**. Ideally tools should provide facilities that allow them to be integrated with other tools and processes used within the workflow of the software development lifecycle (e.g. build processes) or with previously invested static analysis tools that have known limitations. With support for build processes in place, code reviews can be configured to operate at the level of builds, and management teams can

get higher-level visibility into the progress of quality management programmes.

- **Comprehensive reporting facility**. As well as being based on a high-performance and scalable analysis engine, the reporting facility should support common output formats, rich graphical representations, widely adopted or industry specific metrics within a framework that allows customisation.
- **Rich context for reports**. Tools should allow the results of analyses to be packaged and linked to specific software requirements, design and build configurations or management reports. This supports strategies for retaining temporal knowledge and history in the event of a change of developer, development team or manager, for whatever reason.
- **Role based and modular expansion/licensing**. It's important that you can get started with a tool in a way that doesn't require you to make a "big bang" commitment. Licensing models that allow customers to start small, and grow as they prove the value, are much more consumable.
- **Team and community support**. In today's distributed, often multi-party development organisations, it's a dereliction to provide tools without making it easy for individuals to share rules and best practices.
- **Support for measurement and sophisticated trend analysis and analytics**. Tools should ideally provide these facilities in order to support future process improvements and integration with a wider IT intelligence framework if supported. This should be backed by continuous collection of data for daily, quarterly and annual metric reporting.

Technology

The sophistication of any static analysis exercise depends on the number of analysis rules/patterns for different code environments/languages that are supported by a tool.

A good static analysis tool platform should be capable of supporting a number of core programming languages. It should also incorporate defect rules specific to different technology approaches (e.g. service oriented application) and industry or regulation policies either directly or by leveraging existing policy or rules resources.

Beyond code reviews: broader applications for static analysis

Static analysis at the code level will only get you so far. In practice there are often more errors or defects in earlier stages of the software development lifecycle: i.e. in the software requirements gathering and design stages. As this report has already shown, errors and defects early on in the process will almost certainly have a negative impact on the final quality and the ongoing costs.

The future for the ability to do code reviews and static analysis at many different levels is coming. Understanding whether the design is right or whether it is conceptually flawed is the basis of forensic analysis, which is a practice that is growing in importance and prominence. Static analysis is a precursor to forensic analysis: you cannot conceivably do forensic analysis well without strong static analysis tools and procedures.

As forensic analysis becomes more well-understood and practiced, the most beneficial analysis tools will be those which can adapt to the broadening context of reviews and static analysis: tools which can integrate and interoperate with existing tool investments but enable a consistent approach to applying static analysis and review procedures wherever they may be required. This kind of adaptability and integration capability will enable an environment where knowledge and skills gained once can be applied in multiple places and scenarios.

Concrete software quality progression for the IT team

The case for automated static analysis of software code (whether at the developer-level, or centralised within the build processes) is clear: it is impossible to write flawless code. You need a static analysis strategy and framework based on the automated support of tools because:

- It helps protect you from the poor coding skills of less able developers and to share the experiences and knowledge of skilled developers and industry standard review rules.
- Organisations are global. The IT organisation may be widely distributed, and may not be part of the same organisation, because parts of it may be outsourced and located in different geographical locations. You now can no longer conceivably carry out the co-located peer reviewing and code inspections common of past software delivery processes.
- A sophisticated static analysis tool could allow you to track and validate design models, the interactions between software components, and, increasingly importantly, interactions between software components and data sources.

Static analysis is just one aspect of a QA strategy. However, if you don't tackle it then you cannot achieve the full potential of a QA framework. Improving the removal of defects and improving the occurrence rate of defects means that there could be a reduction by a factor of five in the number of defects actually delivered by average organisations.

The old excuses are no longer valid for avoiding software static analysis

Below are some of the common excuses that we find when exploring problems with software quality, and our rebuttals:

- **We haven't got time we need to get the product out**. This leads to the cycle of releasing bug fixes and patches in amongst new features or as new features and because of the timescales even more poor quality code, thereby leading to a cycle of more bugs and fixes.
- **I have no time - I just need to code and deliver**. Greater software complexity through the use of the latest technology advances and complex organisational structures means there will be soon no room for this attitude. Ensuing quality concerns means that it is no longer that simple to "just code" without any thought to quality processes and environments. End users expect more from their applications both in the way that they perform and the value they deliver.
- **The tools are too cumbersome**. Much work has been carried out by vendors in trying to better incorporate static analysis as an integral process within the developer workbench and workflow of their choice. New tool environments from some of the leading players, as well as those from the open source community, are providing sophisticated and easy-to-configure static analysis facilities as an automatic function of the development process.
- **The tools are too expensive**. In the past the tools were certainly pricey and potentially out of the reach of all but the largest organisations or development teams. However, this is not the case today. The tools are not only more affordable; many are modular and role-specific, so that you don't have to buy a "one size fits all" type of product. In addition, tool prices have decreased and there are open source plug-ins to popular IDEs like Eclipse. The savings that you will make in the long run far outweigh the initial expense. More money is spent sorting out bad code over the long run that putting it right in the first place.
- **Tools don't support code analysis of the latest technologies**. In some cases this is

true, but static analysis is being taken seriously by software development platform providers, most of whom are keen to show their credentials in promoting the development of good quality code by providing community forums and support. The specialist vendors solely focusing on static analysis are also expanding their tools. That said, we do believe more work must be done in providing rules that enable deeper levels of inspection for some of the latest technologies.

- **There is no support and I don't want to have to get expensive consultants in**. This is not as true as it used to be, although many of the vendors will provide additional consulting services to those who require them. Many vendors are building communities to share and swap analysis rules. The rule creation facilities within vendors' tools are becoming more standardised, making it easier for users to package up their tool configurations and share them. Admittedly more could be done in this area, but what is out there is a good start.
- **I'm too small an organisation and we don't build complex applications, we just do web designs**. Quality issues mean that too often, poor code is creeping into deployed web-based applications. Besides, web design and coding requires further checks (for example, to guard against malicious online hacks). As more businesses rely on software to run, differentiate or underpin their businesses, software code quality becomes vital in all applications that underpin new business models.