Software Safety –

Back to Basics, Knowing Where to Tap!

~ A DSTA Perspective

ABSTRACT

Software Safety is beginning to become an important element of System Safety as more and more hardware and equipment are run and controlled by software. Customary approaches such as hazard and fault tree analyses would help one to identify and assign hazard(s) to either of a hardware or software nature. This is only the beginning. Once software hazards have been identified, software architects and developers would have to design the application architecture to ensure the software components are easy to test, verify, and maintain. This paper will share the important aspects of white-box testing, and the use of metrics as a proxy to measure the testedness, complexity and maintainability of programme codes. The paper will also briefly describe the DSTA software quality journey, the DSTA Software Quality Assurance (QA) & Testing Framework, the governance to ensure compliance and the challenges faced by the DSTA Software QA & Testing Programme in enforcing compliance of the processes and quality enhancement activities.

Lian Tian Tse

Software Safety – Back to Basics, Knowing Where to Tap! ~ A DSTA Perspective

INTRODUCTION

In the majority of accidents in which software was used to control actions of components, the cause can be traced to requirement flaws such as incomplete requirements in the specified and implemented software behaviour. For example, wrong assumptions were made on how the control system operates in the production environment. However, even if there exists a methodology or technique that could identify all software-related hazards, we could only conclude that "half the battle is won", and we are still presented with a huge risk if the software is not properly designed, built and tested.

It is common practice to use fault tree analysis[1] to identify software-related hazards. The analysis is usually carried out right down at the software interface level in order to trace the hazards into the software requirements and design implementation. Furthermore, software can do more than what is specified in the requirements. Over-zealous developers (or sales and marketing staff) may try to introduce more 'value-added' features and functions to impress and please the customers, or the problem could be having unintended functions unwittingly introduced to the system. All these extras may potentially introduce safety hazards into the system during operation and therefore have to be treated with great caution and care.

Software safety is a subset of system safety. Hence, in the context of safety, we must ensure that the system is protected against, and also designed to handle unexpected software behaviour. This paper will demonstrate how good practices of software engineering such as Software Quality Assurance (QA) and Testing are an essential and necessary function to help enhance and promote software safety within an organisation. Project teams are known to do a much better job in Software QA & Testing activities when they know that there is an independent body to carry out reviews and that a quantitative criteria have been established to assess whether due diligence has been carried out.

THE IMPORTANCE OF WHITE-BOX TESTING

Many companies fall into the trap of accepting software solely based on the user acceptance test (UAT), also commonly known as black-box or functional testing in the software industry, where a set of pre-defined scenarios (or functional tests) was applied to the software system to verify actual results obtained against the expected results. This practice of software acceptance has the following shortcomings:

- We do not know the risks involved when we deploy a system in a production environment.
- We are unsure whether all critical components / modules are appropriately tested.
- UAT and Systems Integration Test (SIT) do not reveal which part of the program is not tested – there is no visibility of test coverage.
- Given the limited resources available, where should one focus his testing efforts to reduce the risk of programme and system failure?

This form of software acceptance is a very common practice in the Information Technology (IT) industry. The reason is not because the industry is unaware of better practices. Rather, the crux of the issue lies with the customers and consumers not understanding the software development life cycle and hence are not knowledgeable enough

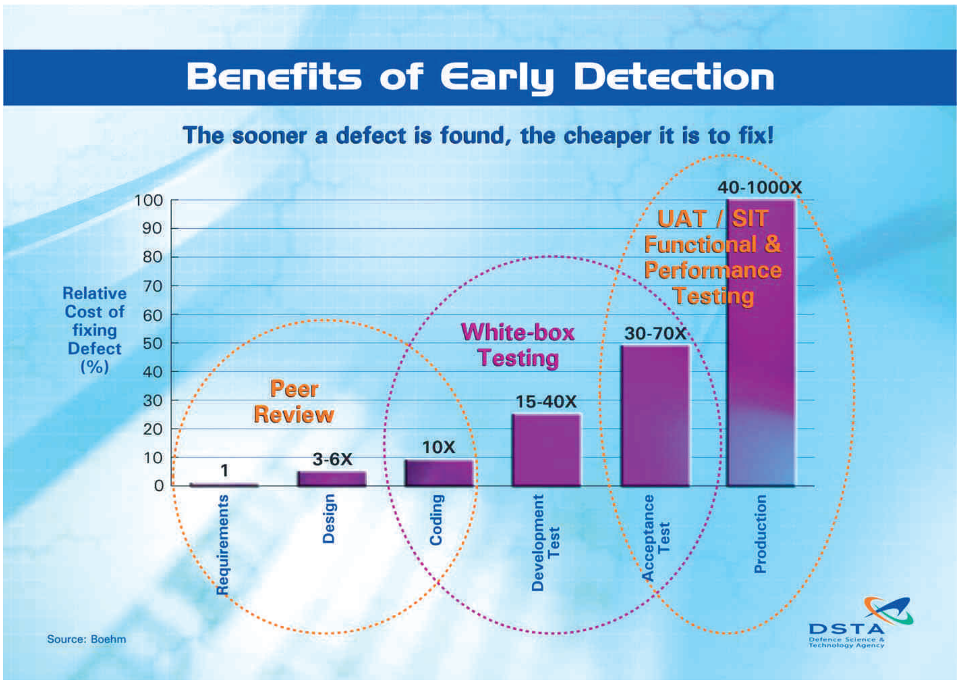

Figure 1. Benefits of early detection of defects

to know what to ask for or expect from the software house. There are obvious economic reasons why most software houses would not go beyond the traditional UAT for software acceptance. A holistic and comprehensive acceptance methodology would entail an elaborate development process that weaves in different kinds of testing requirements, procedures, as well as quality assurance and review activities. All these translate into effort and cost. Regrettably, if one does not know what to ask for, one would probably not get what is rightfully due to him.

Although the system nature of safety implies that the black-box and integration testing will be more relevant, white-box testing (also known as developer testing) can also highlight safety considerations when the safety-critical functions and constraints have been traced to individual modules and variables (Leveson, 1995). White-box testing is carried out by the developers as they write the software codes. The key benefit of white-box testing is that defects (programming bugs) or violations of coding best practices are promptly made known to the developers. This reduces the likelihood of allowing defects to be carried over to subsequent phases of the development life cycle. Studies have shown that the longer it takes to detect the defect as it is promoted through its life cycle, the more difficult and costly it is to fix the problem (see Figure 1).

White-box testing comprises static and dynamic analyses. While it is possible for one to manually conduct code reviews, it is often more cost-effective to use automated tools to flag out bad practices and potential defects. Numerous tools exist in the market, for example, Parasoft's C++ Test and Jtest, Agitar and McCabe IQ suites[2]. All these tools have a common objective i.e. to help developers and managers take guesswork, intuition and other unreliable methodologies out of the picture. What this means is that the code itself is the basis for final management decisions to accept or reject the piece of software. However, every tool has its strengths and limitations. In most cases, they cannot be used to check and verify proprietary codes such as commercial-off-the-shelf products. White-box testing tools are also not able to check Enterprise Resource Products (ERP) like Oracle and SAP because the source codes are not made available. The acceptance of such ERP solutions are normally restricted to functional, performance and security (vulnerability) testing. Hence, the decision to select which tool to invest in will depend largely on the goals and objectives of using such tools in the first place. More importantly, do we have a set of evaluation criteria that has been carefully thought through to help us decide on the tool that is the most cost-effective and easy to use?

Some schools of thought believe that testing is success-oriented. This is true to a certain extent as most testing tools focus on doing what the software is supposed to do i.e. providing required outputs, rather than on what it is not supposed to do or on the effects of failures and errors on safety-related behaviour. Some have even gone further to suggest that erroneous inputs are often not tested. This truth is more evident in the past but with the advancement of technology in recent years, we are beginning to see tools in the market today that can generate run-time test data to break software codes (dynamic tests). However, operator errors and some software error conditions may not be fully addressed by such state-of-the-art testing tools. These are extremely important for safety and hence have to be complemented with other techniques and methodologies. It is important to note that the safety of a piece of software cannot be evaluated by simply looking at the software alone.

USE OF SOFTWARE METRICS TO ENHANCE SOFTWARE SAFETY

DeMarco's insight in 1982 has now become a well-known adage in software engineering: "Managers cannot control or manage what they cannot measure." It has always been a challenge to implement software metrics in an IT organisation to control and manage safety and quality. Often, good intentions are met with unintended results and behaviour. Developers are known to be suspicious of whether the software metrics would be used against them in their performance appraisal (though conversely, it could also be used to justify higher rewards if the metrics showed that they were producing high quality codes). This common fear reflects the prevailing culture in most organisations (and the industry) and is underpinned by the management's level of trust and willingness to forgive when employees make mistakes.

Instituting a new process or a new method of doing things in a large organisation can be an uphill and challenging task. There will always be people who support the initiative; those who oppose the initiative; and those who are indifferent. If an IT organisation does not have a strong quality culture, then the introduction of software metrics without some form of change management framework will lead to limited success at best or total failure in most cases. It has been discovered that applying the CMMI's 10 generic practices (CMM Integration Project, 2001) is a good starting point for those who are interested in making software metrics or any new process operational in their organisation. However, there is no guarantee of success. It will require more than these 10 best practices and additional factors include strategic planning, timing and leveraging opportunities.

One of the most effective tools in making things safer is simplicity and building systems that are intellectually manageable. In today's environment, it is becoming increasingly difficult for managers to handle software development projects because software is growing to previously unimaginable size and complexity. There is probably more software than hardware components in any equipment today as compared to a few decades ago. A good example is the aircraft industry. Furthermore, it is increasingly common that the codes being managed are produced by someone else such as third party contractors or different business units within the company. Under such environments, the ability to manage a software application effectively depends on accurate and complete assessments of the code.

The use of software metrics like cyclomatic complexity (McCabe and Watson, 1996) measures the amount of decision logic in a single software module. It gives the number of logical paths or the recommended minimum set of test paths required to pass through every decision at least once in the software module. Another useful metric is coverage analysis. Coverage analysis can instantly identify untested paths and branches, allowing project managers to streamline test plans to address only untested parts of the programme. This prevents common problems of redundant testing or over-testing programme areas that are at low risk of defects. Coverage testing is therefore a very useful and powerful tool for identifying and analysing untested paths in safety critical modules.

Some tool vendors provide extensive visualisation capabilities with tools like battle maps and flow graphs. Battle maps display a structure chart that graphically represents the functional hierarchy of the program. Often, the tool has an 'exclude' feature for developers and designers to focus on specific segments of the program. Flow graphs graphically display the control structure of the code and path of execution inside each module. It facilitates code comprehension and is often used for code reviews. They are very good for highlighting unstructured behaviour of the code and developers and users alike can graphically see the cause of increased code complexity (see Figure 2).

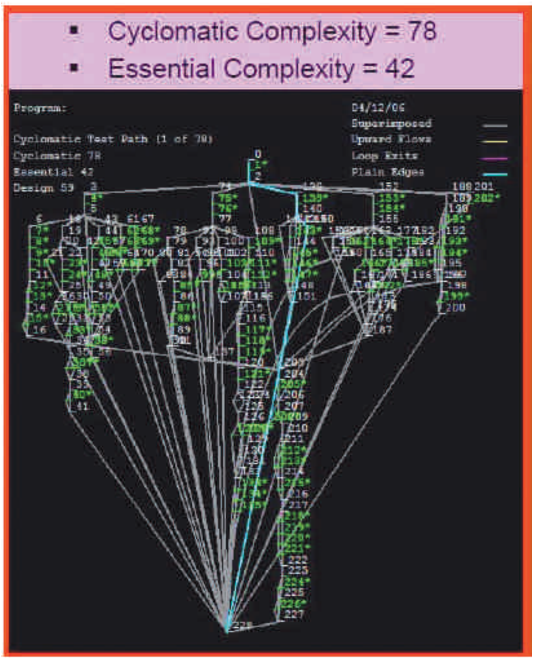

Figure 2. Example of a flow graph

The Software Engineering Institute of the Carnegie Mellon University recommends the following benchmarks for the measurement of cyclomatic complexity (see Table 1).

From the perspective of a customer without much programming knowledge or IT grounding, one could easily comprehend the complexity of the developed codes and whether they are easy to test and maintain by simply looking at the graphical display of the battle maps and flow graphs. These are very effective tools to manage the quality and maintainability of the codes. With the aid of such tools, it is more compelling for software architects, developers and third party contractors to refactor their codes when they are presented with a complex flow graph that showed that it is almost impossible to test their software. Hence, the use of these tools facilitates simplicity and the building of software systems that are intellectually more manageable and hopefully make things safer. In addition, such visibility also provides a more objective assessment of the operations and support costs incurred by both contractors and customers alike.

Cyclomatic Complexity	Risk Evaluation
1 – 10	a simple program, without much risk
11 – 20	more complex, moderate risk
21 – 50	complex, high risk program
Greater than 50	untestable program (very high risk)

Table 1. Benchmarks for measuring Cyclomatic Complexity

THE DSTA SOFTWARE QUALITY JOURNEY

Since the early 1990s, DSTA has been using ISO9001 as the foundation for the organisation's quality management system. DSTA has an elaborate quality manual that articulates the processes and procedures that project teams will have to comply with in the course of their work. Despite these processes and procedures, we still encounter quality and performance issues when systems were deployed in the production environment. It has become clear that a good ISO9001 system is insufficient – we have to do more, like subjecting the software codes to screening and quality control.

DSTA started a centralised test service to provide performance testing services to project teams in July 2004. This test is usually performed prior to the deployment of the system. While we see a significant improvement in the performance of the deployed systems that have undergone the test, there are still issues with regard to programming bugs and poor quality codes that are too complex and difficult to maintain. Quality has to be built into the system at the onset of the project life cycle. One of the most effective ways to address this problem is to introduce white-box testing methodology into the organisation. In June 2005, a small team was formed to explore and make automated white-box testing operational in the organisation. At the same time, another team was involved in the development of a Software QA & Testing Framework (Lian and Chew, 2006) which is built on top of the organisation's quality management system. The framework was endorsed by DSTA senior management on 8 February 2006 (see Figure 3).

DSTA was fortunate in that the senior management is very supportive of these quality initiatives. In April 2006, DSTA officially set up the Software QA & Testing Programme (SwQAT) with the vision of providing leadership in the delivery of quality software that is not only reliable and easy to maintain but also safe to use. The senior management also endorsed

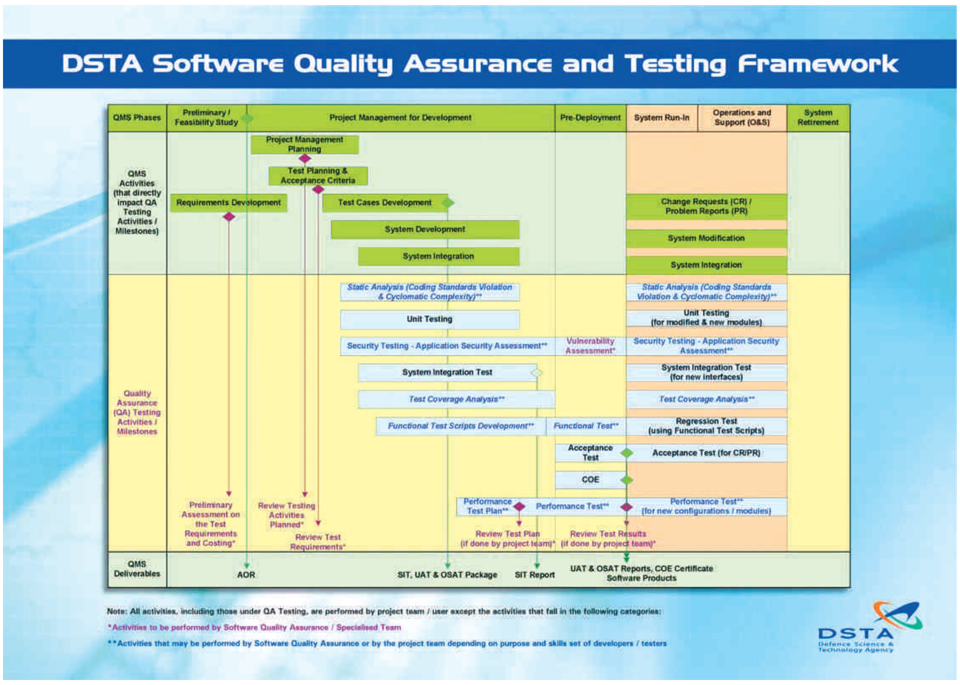
DSTA Software Quality Assurance and Testing Framework
QMS Phases
Preliminary / Feasibility Study
Project Management for Development
Pre-Deployment
System Run-In
Operations and Support (O&S)
System Retirement
QMS Activities (that directly impact QA Testing Activities / Milestones)
Project Management Planning
Test Planning & Acceptance Criteria
Requirements Development
Test Cases Development
System Development
System Integration
Change Requests (CR) / Problem Reports (PR)
System Modification
System Integration
Quality Assurance (QA) Testing Activities / Milestones
Static Analysis (Coding Standards Violation & Cyclomatic Complexity)**
Unit Testing
Security Testing - Application Security Assessment**
Vulnerability Assessment*
System Integration Test
Test Coverage Analysis**
Functional Test Scripts Development**
Functional Test**
Acceptance Test
COE
Performance Test Plan**
Performance Test**
Static Analysis (Coding Standards Violation & Cyclomatic Complexity)**
Unit Testing (for modified & new modules)
Security Testing - Application Security Assessment**
System Integration Test (for new interfaces)
Test Coverage Analysis**
Regression Test (using Functional Test Scripts)
Acceptance Test (for CR/PR)
Performance Test** (for new configurations / modules)
Preliminary Assessment on the Test Requirements and Costing*
Review Testing Activities Planned*
Review Test Requirements*
Review Test Plan (if done by project team)*
Review Test Results (if done by project team)*
QMS Deliverables
AOR
SIT, UAT & OSAT Package
SIT Report
UAT & OSAT Reports, COE Certificate
Software Products
Note: All activities, including those under QA Testing, are performed by project team / user except the activities that fall in the following categories:
*Activities to be performed by Software Quality Assurance / Specialised Team
**Activities that may be performed by Software Quality Assurance or by the project team depending on purpose and skills set of developers / testers
DSTA
Defence Science & Technology Agency

Figure 3. DSTA Software QA & Testing Framework

SwQAT's recommendation for all projects to undergo pre-deployment code screening. By the end of July 2007, SwQAT had managed to perform code screening on more than 70 projects.

SOFTWARE QA & TESTING GOVERNANCE

DSTA outsources a large number of projects to the industry. In order to ensure that the systems delivered to DSTA by the contractors are of high quality, it is important that quantitative measures are used for the acceptance criteria. There are many types of testing that a piece of software has to undergo throughout its life cycle. The more obvious ones are normally 'milestone-kind' of testing which includes interface testing, factory acceptance test, UAT and performance testing. The less obvious yet important ones like white-box and application security testing are often taken for granted and overlooked by developers. SwQAT has identified the following types of testing to be included in the tender contract which contractors must fulfill when they bid for the projects: (1) white-box testing, (2) black-box testing, (3) coverage testing for safety-critical and mission-critical components, (4) application security testing, and (5) performance testing.

For white-box testing, SwQAT uses cyclomatic complexity and violation of industry coding best practices as a proxy for systems acceptance. In addition, for safety-critical and mission-critical modules, coverage testing is to be performed to determine the percentage of codes that has been covered by the black-box testing. The white-box testing tool that we use is also able to identify unused codes and untested codes which are undetectable by the traditional black-box testing. Unused and untested codes could be potential sources of safety risk and hence should be identified and verified.

In DSTA, black-box testing is carried out by the project teams together with the users to validate and verify the functional requirements. SwQAT will help to instrument the source codes for components that need to undergo coverage testing. The project team will then carry out the coverage testing as part of the black-box testing and SwQAT will analyse and interpret the results (test logs) for the project teams as part of the test services rendered to project teams.

SwQAT will also conduct vulnerability assessment on software systems to identify any security risks. Both white-box and black-box application security tools are used for this purpose. For pragmatic and sometimes business-related reasons, defects and bugs (those which are not catastrophic in nature) may be intentionally left uncorrected in the next new release. However, for security vulnerabilities that have been identified, it is mandatory that these defects must be fixed before they are deployed.

Performance testing is usually the last test to be carried out before the system is rolled out. Depending on the objective of the performance test, SwQAT can conduct load, stress and volume tests to identify bottlenecks in the system. This test is often carried out at a staging environment and also at a pilot trial at the production environment without prior notice before declaring the system operational. Once the bottlenecks have been identified, the respective experts from the server, database and network teams would work collectively to fine-tune the system.

SwQAT will produce test reports for white-box, application security and performance testings. Black-box testing report will be done by the project team. After each test, SwQAT will brief the project teams on the findings, analyses and recommendations on how to fix the identified defects. The final approval to release the system for deployment is under the purview of the Programme Director.

CHALLENGES FACED BY SWQAT

In order to be effective, SwQAT needs to have a pool of talented professionals who are not only technically competent in software architecture design and coding practices, but also first class salesmen with the tenacity and people skills to influence and convince their peers to do things differently to achieve better results. This combination of technical and soft skills is not easy to find in any organisation. Finding experienced staff with three to four years of development experience in the organisation to staff SwQAT is a big challenge as we are always competing with the in-house project teams for talents to deliver systems to our customers. Mid-career recruitment is also difficult as many prefer to work in the private sector.

SwQAT took almost three years to build up a team of capable professionals from four to 15 staff specialising in white-box, black-box, performance and application security testing. Since early 2007, we have been focusing our efforts on building up our skills in application security testing. The initial startup was challenging but we were fortunate to be able to tap on two software architects to help kick-start the team on a part-time basis. The software architects were invaluable to the team in helping to verify and select the rule set to be used for flagging out the code violations and security vulnerabilities as well as developing guidelines and standard operating procedures.

Although SwQAT contributes directly to the branding of the organisation, we were not always taken seriously by the project teams at large. As a QA & Testing outfit, SwQAT is viewed as a cost driver and may potentially be blamed for causing delays in schedule. Furthermore, our contributions may not always be appreciated. Nonetheless, SwQAT's staff know that their work and contributions to the organisation are important and valued by DSTA's senior management, and our customers, the Ministry of Defence and the Singapore Armed Forces.

SwQAT has taken great pains to inculcate the quality culture in the organisation. For teams who are willing to work with us, we will be there to help them 'smoothen' their learning curve and also provide technical support to resolve any technical issues or problems encountered. SwQAT also conducts regular training courses to educate staff on the DSTA Software QA & Testing Framework and governance, and hands-on technical training on the use of the automated tools. Through our sincerity and professionalism, we are beginning to win over the project teams. By word of mouth, news of the value-add provided by SwQAT has spread and we are beginning to see an influx of requests for our services. For SwQAT to be successful, we operate on the principle of ensuring that our customers i.e. project teams are successful.

To meet the next challenge in achieving software safety capability, SwQAT has to train her staff to understand system safety concepts and techniques. This is the simple part as we can easily set aside time for the staff to attend relevant courses, workshops and conferences as well as provide them with opportunities to work on system safety projects to gain valuable hands-on experience. On a larger scale, the more challenging task is to train our software developers to understand not only software-related hazards, but also any software requirements plays in the implemention of interlocks and other safety design features. The question is whether we can train the software developers such that they will not inadvertently disable or override system safety features and implement software-controlled safety features incorrectly. The other difficult challenge is to get system safety engineers to be more involved in software development and to include software in the system safety process. This is an aspect SwQAT has to work on as we begin to explore and move into this new domain of work.

CONCLUSION

There are so many software paths, possible inputs and hardware failure modes that testing for all of them is not feasible. A good testing strategy may chance upon a few hazardous software errors or behaviour, but this is far from a rigorous way to identify them. This paper advocates sound Software QA & Testing practices as a means to complement software safety.

In a nutshell, with the aid of advanced automated test tools, software safety can easily be complemented and enhanced through (1) the identification of unused and untested codes; (2) the highlighting of unstructured behaviour, as well as (3) the use of an excellent tool to identify the impact on change analysis given that software systems do change over time due to new or defunct requirements both during software development and operations and support phases. Also, central to any organisation engaging in software development activities is the need to have a system for configuration management, a good traceability matrix to trace requirements for the determination of cases to test scripts, and the execution of the tests.

REFERENCES

Arthur H. Watson, Thomas J. McCabe. (1996). Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST Special Publication 500-235, 1996.

Lian Tian Tse, Chew Wee Hui, Defence Science & Technology Agency. DSTA Software QA & Testing Framework, 2006.

Nancy G. Leveson, (1995). Safeware: System Safety and Computers. University of Washington. Addison-Wesley Publishing Company, 1995.

SEI CMMI V1.1: CMM Integration Project, 2001. Retrived on October 2003 from http://www.sei.cmu.edu/cmmi.

ENDNOTES

[1] There are many types of hazard analysis ranging from a simple checklist to more sophisticated techniques like fault tree analysis. Selection of which technique to use depends largely on the goals of the analysis as each technique has its corresponding strengths and weaknesses.

[2] Disclaimer: Certain trade names and company names are mentioned in the paper. In no case does such identification imply recommendation or endorsement by DSTA, nor does it imply that the products are necessarily the best available for the purpose.

This paper was first presented at the International Systems Safety Conference (13 -17 August 2007) and has been adapted for publication in DSTA Horizons.

BIOGRAPHY

Lian Tian Tse is Assistant Director (Command and Control Information Technology Competency Community) and is concurrently heading the Software QA & Testing Programme in DSTA. He has extensive experience in the solutioning and implementation of Decision Support Systems for the Ministry of Defence and the Singapore Armed Forces. For the last 15 years, he has been working in the IT arena leading projects and programmes that cover a wide domain ranging from Logistics, Operations, Manpower, Training and Education. A recipient of the Defence Technology Group Scholarship, he obtained his degree in Mechanical Engineering from the National University of Singapore. He also holds a Master of Science in Operations Research from the Naval Postgraduate School, USA.