Fault Injection based Analysis of Defect Amplification Index in Technology Variant Commercial Software Application Development

Mr. P Mohammed Shareef Author1[†], Dr. M V Srinath Author2 [†] and Dr. K Gopalakrishnan Author 3[††]

Trimentus Technologies, Chennai, Mahendra Engineering College, Mahendrapuri, Anna University – Coimbatore, Coimbatore

Summary
Fault injection involves the deliberate insertion of faults or errors into software in order to determine its response and to study its behaviour. Fault Injection Experiments have proven to be an effective method for measuring and studying response of defects, validating fault-tolerant systems, and observing how systems behave in the presence of faults. This approach can offer both accuracy of fault injection results transparency of the system dynamics in the presence of faults. The objectives of this study are to measure and study defect leakage, analyse amplification of errors and study "Domino" effect of defects leaked. The approach for fault injection patterns presented in this research is validated by two approaches taken to arrive at the Amplification Index (AI) that represents the effect caused by defects in subsequent phases of software development in business applications. The approaches endeavour to demonstrate the phasewise impact of leaked defects, through statistical analysis of defects leakage and amplification patterns of systems, built using technology (C#, VB 6.0, Java) variants, and also through a causal analysis done on the defects injected.

Key words:
Fault Injection, Dominos Effect, Amplification Index, Defect Leakage and Distribution

Introduction

Formulating reliable and fault tolerant software is difficult and requires discipline both in specifying system functionality and in implementing systems correctly. Approaches for developing highly reliable software include the use of formal methods[1][2][3], and rigorous testing methods[4].

Testing cannot guarantee that commercial and business software is correct[5], and verification requires enormous human effort and is subject to errors[6]. Automated support is necessary to help ensure software correctness and fault tolerance.

Fault injection modelling involves the deliberate insertion of faults or errors into a computer system in order to determine its response. It has proven to be an effective method for measuring and studying response of defects, validating fault-tolerant systems, and observing how systems behave in the presence of faults. In this study, faults are injected in key phases of software development of business application following a typical water fall software life cycle viz., SRS, Design and Source code.

Literature Review

The literature review consolidates the understanding on fault injection, associated topics and subsequent studies to emphasis the need to fault injections in business software application. It also crystallizes the need for awareness, tools and analyzes defect leakage/amplification.

Even after 20 years of existence the awareness of fault injection and associated modelling with tools are very rarely used and understood in the commercial software industry and used. The usefulness in the defect modelling and building fault tolerant software systems are not properly preached and/or practiced. Added, the availability of appropriate literature and software tools is very few and not used in commercial and business application design and testing.

After a detailed review of literature by the researcher it was concluded that there is an industrious interest software fault injection in the software industry to develop commercially reliable software.

Approach

In recent years there has been much interest in the field of software reliability and fault tolerance of systems and commercial software. This in turn has resulted in a wealth of literature being published around the topic, such as the Fault Injection in the form of the 'Marrying Software Fault Injection Technology Results with Software Reliability' by Jeffrey Voas, Cigital Norman Schneidewind.

Many critical business computer applications require "fault tolerance," the ability to recover from errors or exceptional conditions. Error free software is very difficult to create and creating fault tolerant software is an even greater challenge. Fault tolerant software must

Manuscript received December 5, 2009
Manuscript revised December 20, 2009

successfully recover from a multitude of error conditions to prevent harmful system failures.
Software testing cannot demonstrate that a software system is completely correct. An enormous number of possible executions that must be examined in any real-world sized software system. Fault tolerance expands the number of states (and thus execution histories) that must be tested, because inconsistent or erroneous states must also be tested.
Mailing lists, websites, research and forums have been created in which all aspects of this fresh new niche software engineering area are discussed. People are interested, partly because it is a new area but also because the whole field of commercial software reliability is in itself so interesting; as it holds so many wide ranging disciplines, perspectives and logic at its core. Software reliability engineering is uniting professionals in disciplines that previously had little to do with one another, it is creating more opportunities for employment in the online environment, and it is changing the face and structure of all information that we seek out on the web. In the era of economic recession, customer demands reliable, certified and fault tolerant commercial and business software applications.
In this research, the focus is on software testing techniques that use fault injection. Several potentially powerful existing systems have drawbacks for practical application. We first examine existing fault injection techniques and evaluate their potential for practical application in commercial and business software applications. Available and accessible literature infrastructure including premium subscribed IEEE and ACM resources were studied and summarized for literature review from 1986 (20 years).

Fault Injection Modelling

Fault Injection Modelling (FIM) involves the deliberate insertion of faults or errors into a computer system in order to determine its response. It has proven to be an effective method for measuring and studying response of defects, validating fault-tolerant systems, and observing how systems behave in the presence of faults. In this study, faults are injected in all phases of Software Development Life Cycle viz., Requirements, Design and Source code.

Objectives:

The objectives of conducting these experiments are to measure process efficiencies, statistically study, analyse and report defect amplification of defects (Domino's effect) across software development phases with a similar system constructed with technological variation.
The goal of this research is to understand the behaviour of faults and defects pattern in commercial and business software application and defect leakage in each phase of application development.
Throughout the literature certain questions reoccur, which one would anticipate when a new field emerges in commercial software fault tolerance. People are interested, and want to understand and define commercial software reliability and fault tolerance, so the following questions which are recurrent throughout the literature are not surprising:

- Why study Fault Injection Modelling?
- Why study business software fault tolerance requirements?
- Why are they called 'Fault Injection & Error Seeding'?
- Why review Software Implemented Fault Injection (SWIFI)?
- What work was performed, current status and work proposed?

These questions will be expanded upon throughout the research, and seek to bring clarity to those who want to find the answers to the above, or to see if there truly are any answers!

Background Concepts

A fault is a hardware or software defect, inconsistency, transient electrical field, or other abnormal circumstance. An error is an invalid internal state, which may or may not be detected by the system.
A failure is an invalid output. Thus a fault or error becomes a failure when it propagates to the output. There is a natural progression from fault to error to failure. Recovery code is the part of a program that is designed to respond to error states. Recovery code executes after the program recognizes that some erroneous or abnormal state has been entered. This code should gracefully restore the system to a valid state before a failure occurs.
Figure 1 shows the progression from faults to errors and finally to failures. The recovery code should serve as a safety net to prevent the progression from error to failure. A fault tolerant system should never fail, even if it has faults.
Testing recovery code requires the modeling of bad states that accurately simulate exceptional situations. As much as 50% of a fault tolerant program can consist of recovery code. Although testing might include invalid data that executes some of the recovery code, often much of this code is never executed during normal testing.
Any recovery code testing technique must be based upon an assumed fault model[7]. We assume that all faults will behave according to some specific rules. Any fault model can only consider a subset of all possible faults.

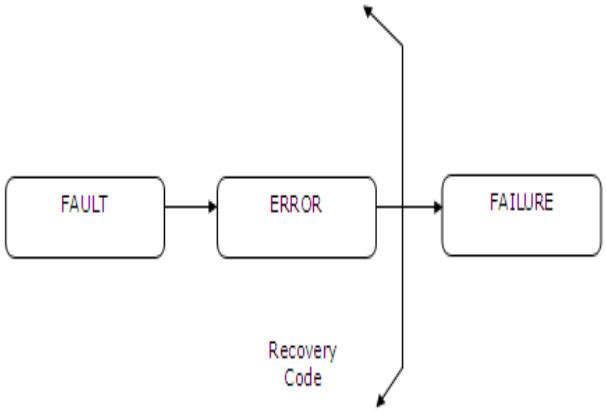

Figure 1 : Fault Tolerance Terms

For example, a common debugging practice is to insert a series of \print" statements in key positions. This debugging practice assumes a particular fault model.
Faults will cause the program to execute in the incorrect order and will be demonstrated in Figure 2: Taxonomy of Fault Injection Techniques in the printed output. Clearly, not all faults will adhere to this model.

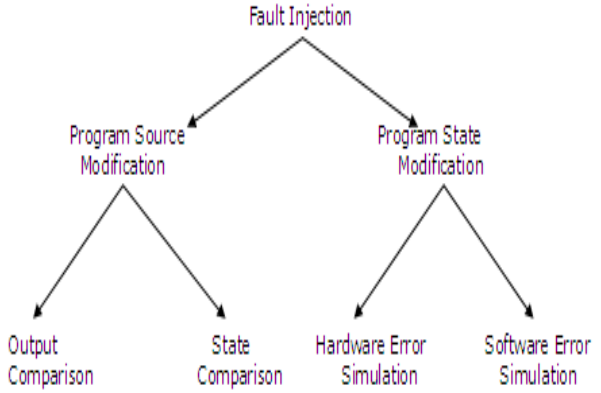

Figure 2: Taxonomy of Fault Injection Techniques

No one fault model will fit all faults. However, a fault model can be very effective in detecting faults that fit the model.
Fault Injection technique of fault injection dates back to the 1970s when it was first used to induce faults at a hardware level. This type of fault injection is called Hardware Implemented Fault Injection (HWIFI) and attempts to simulate hardware failures within a system. The first experiments in hardware fault injection involved nothing more than shorting connections on circuit boards and observing the effect on the system (bridging faults). It was used primarily as a test of the dependability of the hardware system. Later specialised hardware was developed to extend this technique, such as devices to bombard specific areas of a circuit board with heavy radiation. It was soon found that faults could be induced by software techniques and that aspects of this technique could be useful for assessing software systems. Collectively these techniques are known as Software Implemented Fault Injection (SWIFI)[8].
Martin defines software fault injections as faults which are injected at the software level by corrupting code or data. So faults are applicable at the implementation phase when the code of the system is available, and it can be applied on an application to simulate either internal or external faults.
Internal faults represent design and implementation faults, such as variables/parameters that are wrong or not initialized, incorrect assignments or condition checks. External faults represent all external factors that are not related to faults in the code itself but that alter the system's state.
The injection of failures can discover errors that normal procedures cannot. First, it tests the mechanisms of exception and treatment of failures that in normal circumstances are not sufficiently proven and, helps to evaluate the risk, verifying how much defective can be the system behavior in presence of errors. All of the injection failures methods are based on concrete hardware or software characteristics associated to systems which are applied, then, to realize generalizations is a very complicated task.

Prior Work on Fault Injection

Fault injection can be used to modify either a program's source code text or the machine state of an executing program. Figure 2 shows taxonomy of the key methods of fault injection. Fault injection techniques based on static analysis -program source modification - are modeled by the left subtree.
The most common static fault injection is mutation testing. The right subtree in Figure 2 models dynamic fault injection techniques where changes are made to an actively running program's state. Much of the recent fault injection research is concerned with dynamic injection.

Domino’s effect:

Domino’s effect is the cascading effect of defects from the initial stages of the project to all the subsequent stages of the software life cycle. Errors undetected in one work product are ‘leaked’ to the child work product and *amplifies* defects in the child work product. This chain reaction causes an exponential defect leakage. E.g.: undetected errors in requirements leak and cause a significant number of defects in design which, in turn, causes more defects in the source code. The result of this study is to arrive at an “Amplification Index” which will

characterize the extent of impact or damage of phase-wise defects in subsequent Software Development Life Cycle (SDLC) phases.
The defect components in a work product and leakage into subsequent phases are illustrated below:

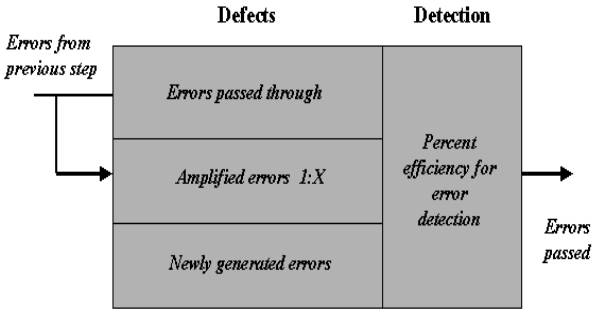

Figure 3: Fault Injection Patten

Trimentus approach for Fault Injection Experiments

Defects were deliberately injected into each phase (work product) in the software development life cycle of a typical application development project and the effect of the defects injected was studied subsequently. The injected defects are typical defects that are characteristic of the software systems of a commercial application on Library Management System (LMS) and were chosen from the organizational defect database.
An approach was adopted towards studying the impact of defect amplification in a software system was causal analysis of the defects occurring in subsequent phases caused due to injected defects.

Fault injection can occur in several ways:

- Additional code can be linked to the target program and executed synchronously with the program flow.
- A separate process can perform the injection asynchronously with the flow of the target process.
- Separate hardware can directly access the memory to modify the state, thus not affecting the timing characteristics of the target process.

Overlay faults occur when a program writes into an incorrect location due to a faulty destination operand. Chillarege and Bowen claim that overlay faults account for 34% of the errors in systems programs. The experiment involved the use of failure acceleration, decreasing fault and error latency and increasing the probability that a fault will cause an error. The experiment applied failure acceleration by corrupting a large region of memory in a single injection. To inject an overlay fault, all bits in an entire page of physical memory are set to one. Because the page is in physical memory, the probability that the latency will be short is further increased. About 16% of the faults immediately crashed the system; about 14% caused a partial loss of service, which was usually recovered from soon after.
Half of the faults did not cause failures. These potential hazards are failures waiting to occur. The injection process used was manual and only 70 faults were injected during the entire experiment.

Software faults introduced include:

- Initialization faults: incorrectly or uninitialized variables. They are modeled by dynamically replacing the initializing assembly instructions with incorrect values or no-ops.
- Assignment faults: incorrect assignment statements. Variable names on the right hand side are changed by dynamically mutating the assembly code.
- Condition check faults: missing condition checks, for example, failure to verify return values. Condition checks are either entirely overwritten with no-ops, or replaced an incorrect condition check.
- Function faults: Invalid functions. The assembly code for a function is dynamically replaced with the assembly code from a manually rewritten alternate version.

Initialization faults can be caught statically with a good compiler. The assignment and condition check faults are clearly relevant to the testing of recovery code, since an incorrect assignment or condition can be a condition that should force the execution of recovery code. Function faults are also relevant, especially if they could be automatically generated. Unfortunately, manual rewriting of sections of code is prohibitive in a large system.

Why study Fault Injection Modelling?

Fault Injection Modelling has gradually crept into prominence over the last decade as one of the new buzz words in software design.
However, as Martin observes:

‘The main characteristic of fault injection software is that it is capable of injecting failures into any functional addressing unit by means of software, such as memory, registers, and peripherals. The goal of the fault injection software is to reproduce, in the logical scope, the errors that are reproduced after failures in the hardware. A good characterization of failure model should be allowed that

this one was as versatile as possible, allowing a major number of combinations among the location, trigger conditions, kind of fault and duration, so that the coverage was maximum. Recent days, the Fault Injection technique has been considered as a very useful tool to monitor and evaluate the behavior of computing systems in the presence of faults. It's because the tool tries to produce or simulate faults during an execution of the system under test, and then the behavior of the system is detected.'[9]

- The Carnegie Mellon Software Engineering Institute[1] reports that at least 42-50 percent of software defects originate in the requirements phase.
- The Defense Acquisition University Program Manager Magazine [2] reports that a Department of Defense study that over 50 percent of all software errors originate in the requirements phase.

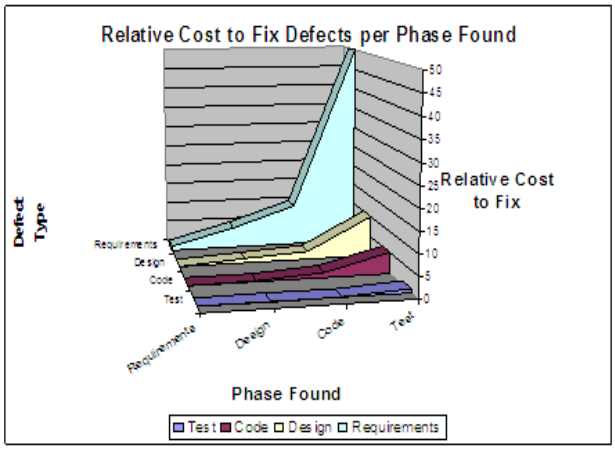

Figure 4: Relative Cost to Fix Defects Vs Development Phases

1. MSDN (November, 2005) "Leveraging the Role of Testing and Quality Across the Lifecycle to Cut Costs and Drive IT/Business Responsiveness "

2. Direct Return on Investment of Software Independent Verification and Validation: Methodology and Initial Case Studies, James B. Dabney and Gary Barber, Assurance Technology Symposium, 5 June 2003.

Description of software system

A Library Management System (LMS) help in automating functions of the library. It helps in reducing the time spent in record keeping and management effectively. The management information system application was used to conduct the fault injection experiments. The same application was developed in the following technologies in 3G languages;

Table 1

Project Id	Programming Language	RAD Tool	Database
LMS 1	C#	Visual Studio 2005	SQL Server 2005
LMS 2	Visual Basic 6.0	Visual Studio 6.0	Ms Access 2007
LMS 3	Java (jdk1.5)	NetBeans IDE 5.0	SQL Server 2005

LMS was simultaneously developed by same project team and were made mutually exclusive. The application development for the projects followed the same process as described in the quality management system for software development of Trimentus. LMS was chosen to FIM because common MIS Domain knowledge for the application was high; it can be independently managed and developed; it covers the entire development life cycle; and the technology used is typical of current commercial applications and technologies in vogue.

SDLC, technology, exclusiveness allows different types of faults to be injected at various phases without bias and enables direct comparison.

In this paper, the system contains injected defects common across all projects. The same count of defects (5 numbers) were introduced in each phase of SDLC.

Results of the experiments

Requirements Review

SRS (Software Requirement Specification) document was prepared and used as the basis for development of for all the projects. However, after the review of SRS, defects were injected into the same document. The SRS containing the defects were baselined by project team respectively to be used as basis for the design.

The common known defects injected into the SRS are given below:

[1] Carnegie Mellon Software Engineering Institute, The Business Case for Requirements Engineering, RE' 2003, 12 September 2003

[2] Defense Acquisition University Program Manager Magazine, Nov-Dec 1999, Curing the Software Requirements and Cost Estimating Blues

Table 2

Action taken	Defect Injected	Defect severity	Defect Type
Deleted	Reports based on classification by Type of books	High	Missed Requirement
Modified	Changed User Login to Student ID Changed the default status of the books given from "Pending" to "Borrowed" Add more records option not given as part of screen layout	Medium	Incomplete, Missed Requirement
Added	Obtaining the proposed date for return of books	High	Ambiguous
Deleted	Set the type of fine	High	Missed Requirement
Added	Set the number of times a books can be renewed by the members	Medium	Incorrect Requirement

Design Phase Analysis

Design document prepared with (fault injected) SRS as basis. There were several defects observed with "source" as SRS. The Injected defects are major cause for design defects.

Design Review

Table 3

	Source			
	SRS		Design	
	Injected	Inherent	Leaked	Inherent
LMS 1	5	4	4	8
LMS 2	5	8	7	6
LMS 3	5	5	7	9

Design Defect Amplification: Technology Variant

The following chart represents the comparison of amplification index between the LMS developed on different technologies. The amplification of design defects caused due to the injected requirement defects in LMS is evidenced in all technologies and more prominent in VB technology.

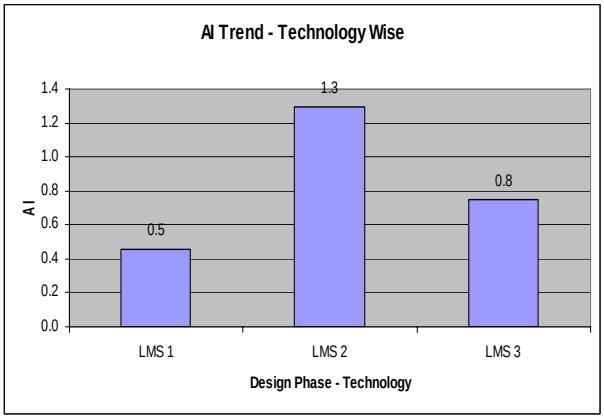

Figure 5: Amplification Index Trend – Design

Amplification Index (AI) for Requirements

The following methodology was used to calculate Requirement Amplification Index (i.e. impact of Requirements defects on Design)

Table 4

Application	Formula	AI (Requirements on Design)
LMS1	No. of design defects caused due to injected Requirement Defects / No. of injected Requirement defects	2/5 = 0.5 (rounded) ➔ *One requirement defect leaked causes 0.5 defect in design in C # technology*
LMS2	No. of design defects caused due to injected Requirement Defects / No. of injected Requirement defects	6/5 = 1.3 (rounded) ➔ *One requirement defect leaked causes 1.3 defect in design in VB technology*
LMS3	No. of design defects caused due to injected Requirement Defects / No. of injected Requirement defects	4/5 = 0.8 (rounded) ➔ *One requirement defect leaked causes 0.8 defect in design in Java technology*

Defects in Design

Various types of known design defects were introduced after design review:

Table 5

Action taken	Defect Injected	Defect severity	Defect Type
Removed	Validation and authentication of authorized students	High	Interface, Incomplete
Modified	Data Type Changed	Medium	Database, Incorrect
Review finding	Editing of book type by borrower	High	Incorrect
Modified	There is a possibility to add null records when no validations are made or no exceptions are handled	Medium	Incorrect
Changed	A datagrid displays the content only when the recordset is open.	Low	Database Incorrect

Statistical Analysis and Validation

Based on the AI derived from the above requirement data analysis, a statistical study was carried out to understand and analyse the statistical significance and relationship of AI across phases.
A hypothesis was formulated based on the conditions of analysis as follows;

Ho : Requirements Amplification Index is same across technologies
H1 : Requirements Amplification Index is different between technologies

Minitab tool was used to analyse the data set of Requirement Amplification Index. A simple T-test was run to validate the statistical significance of the requirement AI data across technologies.

Analysis Results:

One-Sample T: C8
Test of mu = 0 vs mu not = 0

Variable	N	Mean	StDev	SE Mean
C8	3	0.867	0.404	0.233

Variable	95.0% CI	T	P
C8	(-0.137, 1.871)	3.71	**0.065**

The statistical rule of elimination is;

1. If the P- Value > .05 , Then Ho is true and there is no difference in the groups. = Accept Ho
2. If the Value < .05 , Then Ho is false and there is a statistically significant difference. = Reject Ho and Accept H1

This results in; 0.065 > 0.05 ;So by the rule , Accept the Ho.

To conclude that, “Requirements Amplification Index is same across technologies and there is no statistical significant difference on AI across technologies in the Library Management System (LMS) developed in different technologies”.

Coding Phase Analysis

Coding was performed with (fault injected) design as basis. There were several defects observed with “source” as Design and Requirements. The Injected defects were the major cause for Code defects detected in Code review.

Code Review of LMS

Table 6

	Source					
	SRS		Design		Code	
	Injected	Inherent	Leaked + Injected	Inherent	Leaked	Inherent
LMS 1	5	4	4 + 5	8	7	7
LMS 2	5	8	7 + 5	6	9	6
LMS 3	5	5	7 + 5	9	10	6

Code Defect Amplification: Technology Variant

The following chart represents the comparison of amplification index between the LMS developed on different technologies. The amplification of coding defects caused due to the injected design defects in LMS is evidenced in all technologies and more prominent in VB technology.

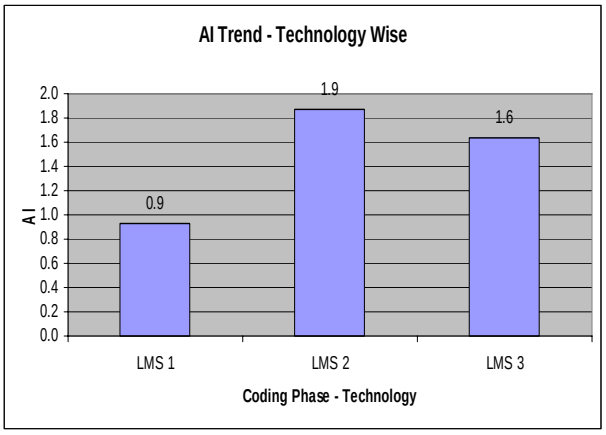

Figure 6: Amplification Index Trend – Coding

Amplification Index for Design

The following methodology was used to calculate Design Amplification Index (i.e. impact of Design defects on Code)

Table 7

Application	Formula	AI (Design on Code)
LMS1	No. of Code defects caused due to injected Design Defects / No. of injected Design defects	4.9/5 = 0.9 (rounded) ➔ *One design defect leaked causes 0.9 defect in code in C # technology*
LMS2	No. of Code defects caused due to injected Design Defects / No. of injected Design defects	9/5 = 1.9 (rounded) ➔ *One design defect leaked causes 1.9 defect in code in VB technology*
LMS3	No. of Code defects caused due to injected Design Defects / No. of injected Design defects	8/5 = 1.6 (rounded) ➔ *One design defect leaked causes 1.6 defect in code in Java technology*

Statistical Analysis and Validation

Similarly, based on the AI derived from the above design data analysis, a statistical study was carried out to understand and analyse the statistical significance and relationship of AI across design phases.

A hypothesis was formulated based on the conditions of analysis as follows;

Ho : Design Amplification Index is same across technologies
H1 : Design Amplification Index is different between technologies

Minitab tool was used to analyse the data set of design Amplification Index. A simple T-test was run to validate the statistical significance of the design AI data across technologies.

Analysis Results:

One-Sample T: C9

Test of mu = 0 vs mu not = 0

Variable	N	Mean	StDev	SE Mean
C9	3	1.467	0.513	0.296

Variable	95.0% CI	T	P
C9	(0.192, 2.741)	4.95	**0.038**

Table 8

The statistical rule of elimination is;

1. If the P- Value > .05 , Then Ho is true and there is no difference in the groups. = Accept Ho
2. If the Value < .05 , Then Ho is false and there is a statistically significant difference. = Reject Ho and Accept H1

This results in; 0.038 < 0.05 ;So by the rule , Reject Ho and Accept H1.

To conclude that, “Design Amplification Index is different across technologies and there is a statistical significant difference on design AI across technologies in the Library Management System (LMS) developed in different technologies”.

Testing Phase Analysis

Testing was performed with (fault injected) code as basis. There were several defects observed with “source” as Coding, Design and Requirements. The injected defects were the major cause for Code defects detected in Testing.

Testing of LMS

Table 9

	Source					
	Design		Code		Testing	
	Leaked + Injected	Inherent	Leaked + Injected	Inherent	Leaked	Inherent
LMS 1	4 + 5	8	7 + 5	7	9	0
LMS 2	7 + 5	6	9 + 5	6	14	4
LMS 3	7 + 5	9	10 + 5	6	15	2

Test Defect Amplification: Technology Variant

The following chart represents the comparison of amplification index between the LMS developed on different technologies. The amplification of test defects caused due to the injected code defects in LMS is evidenced in all technologies and more prominent in VB technology.

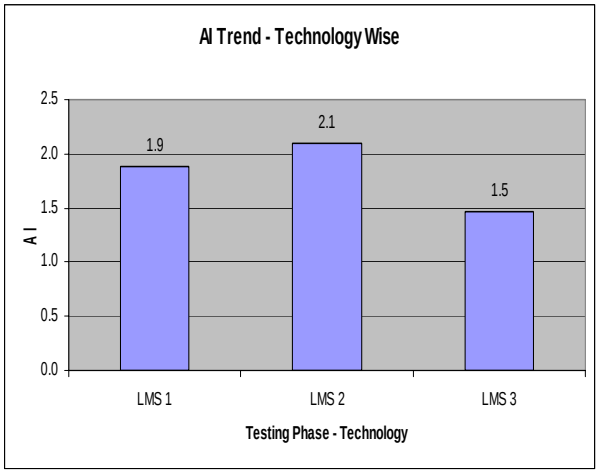

Figure 7: Amplification Index Trend – Testing

Amplification Index for Code

The following methodology was used to calculate Test Amplification Index (i.e. impact of Code defects on Test results)

Table 10

Application	Formula	AI (Code on Test results)
LMS1	No. of Test results defects caused due to injected Code Defects / No. of injected Code defects	9/5 = 1.9 (rounded) → *One code defect leaked causes 1.9 defect in test results in C # technology*
LMS2	No. of Test results defects caused due to injected Code Defects / No. of injected Code defects	11/5 = 2.1 (rounded) → *One code defect leaked causes 2.1 defect in test results in VB technology*
LMS3	No. of Test results defects caused due to injected Code Defects / No. of injected Code defects	7/5 = 1.5 (rounded) → *One code defect leaked causes 1.5 defect in test results in Java technology*

Statistical Analysis and Validation

Similarly, based on the AI derived from the above code data analysis, a statistical study was carried out to understand and analyse the statistical significance and relationship of AI across test phase.

A hypothesis was formulated based on the conditions of analysis as follows;

Ho : Code Amplification Index is same across technologies
H1 : Code Amplification Index is different between technologies

Minitab tool was used to analyse the data set of Code Amplification Index. A simple T-test was run to validate the statistical significance of the code AI data across technologies.

Analysis Results:
One-Sample T: C10
Test of mu = 0 vs mu not = 0

Variable	N	Mean	StDev	SE Mean
C10	3	1.833	0.306	0.176

Variable	95.0% CI	T	P
C10	(1.074, 2.592)	10.39	**0.009**

Table 11

The statistical rule of elimination is;

3. If the P- Value > .05 , Then Ho is true and there is no difference in the groups. = Accept Ho

4. If the Value < .05 , Then Ho is false and there is a statistically significant difference. = Reject Ho and Accept H1

This results in; 0.009 < 0.05 ;So by the rule , Reject Ho and Accept H1..

To conclude that, "Code Amplification Index is different across technologies and there is a statistical significant difference on design AI across technologies in the Library Management System (LMS) developed in different technologies".

Conclusions

AI Trend Analysis.

The Amplification Index indicates the extent of damage caused by a defect in various phases of the project. The index increases with every step in the life cycle of the project. This is evident in the case of Microsoft technologies (VB and C#.net) but AI in the case of open source technologies such Java, the AI increases in requirements and design but in code, it is found have marginal decrease compared to other technologies. It is also seen that defects amplification in the VB Technology show substantial increase in the amplification index across phases compared to other selected technologies.

The relative growth of AI across phases in Java technology is less compared to Microsoft technology. This indicates a better fault tolerance for Java technology.

It was concluded and validated statistically that;

- Requirement defects amplification index across on identified technologies remains are same,
- Design and Code defects amplification index across on technologies vary based on technologies for the common application developed in the same domain

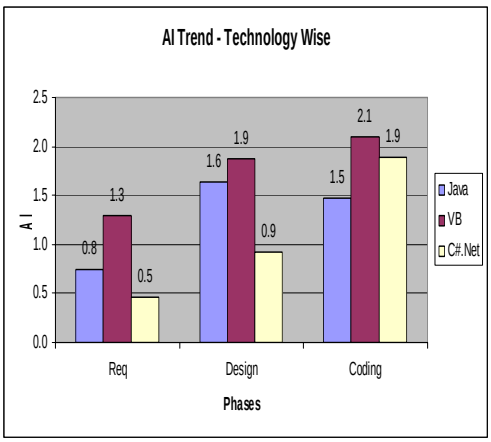

Figure 8: Amplification Index Trend – Technology Wise

- Defect Leakage and Distribution Analysis

The defect leakage analysis emphasizes the importance of thorough and systematic reviews in the early stages of a software project with an emphasis on defect prevention. The analysis indicates a high increase of cost and effort to remove the defects at later stages. The number of defects increases exponentially as a direct result of defects leaked from previous stages.

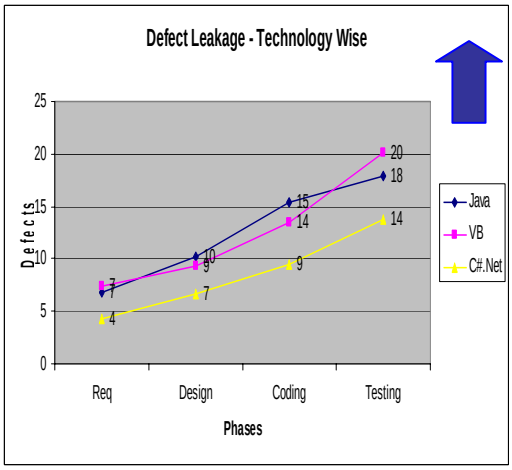

Figure 9: Defect Leakage

Future Experiments

Currently, the study is being extended to analyse the effect of the defects and amplification index in the development phases of the different domain based projects developed with same technology.

Guidelines for review time and effort estimation are being computed by analysing and defining the review and test stop criteria. Error seeding during testing can be carried out to define the test stop criteria.

Limitations of Experiments

The following are the limitations of the experiments:

- Causal analysis is relatively subjective to understand the cause of amplified defect. This required detailed review and discussion with project team and technical/technology experts.
- Defect removal efficiency percentage used for experiments in different technologies are based on a test in a sample requirement, design and code with known defects provided to project members and review efficiency percentage derived from the defects detected.
- It is verified that the skill set of the analysts and programmers working in the projects are same and/or similar across technologies.

References

[1] Hall. Seven myths of formal methods. IEEE Software, 7(5):11-19, September 1990.

[2] C.B. Jones. Systematic Software Development Using VDM. Prentice-Hall International, London, 1986.

[3] S.J. Garland, J.V. Guttag, and J.J. Horning. Debugging larch shared language specifications. IEEE Trans. Software Engineering, 16(9):1044-1057, September 1990.

[4] W. Howden. A functional approach to program testing and analysis. IEEE Trans. Software Engineering, SE-12(10):997-1005, October 1986.

[5] L. J. White. Basic mathematical de_nitions and results in testing. In B. Chandrasekaran and S. Radicchi, editors, Computer Program Testing, pages 13-24. North-Holland, 1981.

[6] R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. Communications of the ACM, 22(5):803-820, May 1979.

[7] Barry W. Johnson. Design and Analysis of Fault-Tolerant Digital Systems. Addison-Wesley, Massachusetts, 1989.

[8] Daniel Dreilinger, Lijun Lin. Using Fault Injection to Test Software Recovery Code, November 1995

[9] Leme, Nelson G. M.; Martins, Eliane; Rubira, Cecilia M. F. "A Software Fault Injection Pattern System". Proceedings of the IX Brazilian Symposium on Fault-Tolerant Computing. Florianópolis, SC, Brazil, March 5th-7th, 2001, pages 99-113.

Mr. Paloli Mohammed Shareef, CISA, CISM, CGEIT, PMP, is the Executive Vice President and Principal Consultant of Trimentus Technologies, is a research scholar at Anna University – Coimbatore with over 12 years of experience in Software Engineering, Quality Management and Information Security. He has special interest in software reliability and information security management. He has authored 15 technical papers and made presentations in forums and institutions such as CSI, SPIN, Anna Universities, and Professional Engineering Colleges.

Dr. M V Srinath, PhD, is the Professor, Mahendra Engineering College, Namakkal, with over 12 years of experience in Multimedia Instructional Design and Delivery and Principles and Practices of Software Engineering and Web Engineering. He has special interest in Instructional Materials and Media. He has authored over 50 technical/journal papers and the resource person for the courses organized and conducted by National Institute of Technical Teachers Training and Research (NITTTR) for the teachers of Polytechnics and Engineering colleges.

Dr. K Gopalakrishnan, is the Executive Director of Deccan Institute of Advanced Studies and member of Academic Council and Board of Studies & Research of Dr. MGR University, Chennai and Anna University – Coimbatore. He is having nearly 24 years of experience in Teaching & Research, Industrial Consultancy & Training. He is having 32 Technical publications to his credit. His research area of interest includes Design and Analysis of Integrated Management System, Quality, Optimization and SQP Trilogy.