The Proceedings of the Second International Research Workshop

On

Advances and Innovations in System Testing

The University of Memphis
Systems Testing Excellence Program
FedEx Institute of Technology
The University of Memphis
Systems Testing Excellence Program
FedEx Institute of Technology

Second International Research Workshop
on
Advances and Innovations in System Testing

The workshop is organized by the:

Systems Testing Excellence Program - STEP (step.memphis.edu) at the University of Memphis. STEP is an interdisciplinary program under the auspices of the Departments of Computer Science, Management Information Systems and Computer Engineering, and FedEx Institute of Technology.

http://STEP.us.com/conference/

May 4-6, 2008
Memphis, Tennessee
Monday, May 5, 2008

Presentation Title & Speaker

10:15–11:00  Paper Session 1

P1: Sources of Conflict between Developers and Testers in Software Development: A Preliminary Investigation
Xihui Zhang, Jasbir Dhaliwal, Mark Gillenson, Gertrude Moeller

P12: A Simulation of Six Models of Engaging Testers in the Systems Development Life-Cycle
Mark Gillenson, Michael Racer, Sandra Richardson, Xihui Zhang

P21: Allignment between developers and testers – an Implementable Methodology
Colin Onita, Jasbir Dhaliwal

2:30–3:30. Paper Session 2

P36: Integration of Testing and Other Software Development Tools
Thomas Meservy, C. Zhang, Bill Kettinger

P51: Representing Test Cases for Semantic Web-based Software Testing Systems
Nor Yahaya

P55: Testing Using Program Instrumentation and Visualization
Lakshmi Narasimhan

P65: A Survey on Fault Injection-based Approach for Dependability Evaluation
S.M. Masum, S. Saha, Mohammed Yeasin

3:30–3:45. Coffee Break

Tuesday, May 6, 2008

Presentation Title & Speaker

10:00–10:45 Paper Session 3

P74: Data Mining of Business Processes to Detect Software Errors
Alan Hevner, Balaji Padmanabhan, University of South Florida

P80: Predicting Defects in Large Open Source Software
Raluca T. Stoian, Vasile Rus, Sajjan Shiva

P90: Machine Learning-based Approach for Testing Large Scale Software: A Survey
S. Sorower, A.K.M. Rahman, Mohammed Yeasin

1:00–1:45. Paper Session 4

P101: Testing as a Service
Yang Yang, Colin Onita, Jasbir Dhaliwal

P117: Applying TDD to Web-Application Development – Does Variety Help or Harm?
Wolf-Gideon Bleek, Sebastian Pappert
Hamburg University

P127: Defect Management System for Software Development Process
Dinesha Koravangala V.
I.I.I.T. Bangalore

2:00–2:45. Paper Session 5

P137: Wikis for Knowledge Management in Software Testing: Success Factors and Benefits
Ted Lee, Bill Kettinger

P145: Perspectives of Offshore Testing Vendors
Robin Poston, Bill Kettinger, Judy Simon

P155: Offshore Vendor Management: Results from a Recent Survey of Managers
Judy Simon, Robin Poston
Sources of Conflict between Developers and Testers in Software Development: A Preliminary Investigation

Xihui Zhang  
*University of Memphis*  
xihui.zhang@memphis.edu

Jasbir S. Dhaliwal  
*University of Memphis*  
jdhaliw1@memphis.edu

Mark L. Gillenson  
*University of Memphis*  
mark.gillenson@memphis.edu

Gertrude Moeller  
*FedEx Express*  
gmoeller1@fedex.com

Abstract

Interpersonal conflict between software developers and testers is inevitable and pervasive. This conflict is likely to be negatively associated with software quality and job satisfaction. This study addresses one major research question: What are the sources of interpersonal conflict between developers and testers in software development? Using a qualitative approach, we collect and analyze fifty developer-tester conflict scenarios from professional developers and testers. Preliminary results indicate that conflict sources between software developers and testers fall into three major categories: Process, people, and communication. Conflict sources are presented in a category-subcategory-example format. Implications for research and practice are discussed.

1. Introduction

Software development is a complex process that necessitates interactions between diverse individuals in different roles. These roles typically include end users, business analysts, systems analysts, designers, programmers, testers, and project managers. Interaction can occur between any two roles; and one of the most important interactions in software development process is between developers (a category that typically includes systems analysts, designers, and programmers) and testers [9], [39]. One natural outcome of human interaction is interpersonal conflict. Interpersonal conflict results when interdependent parties have different goals, mindsets, values, preferences, backgrounds, and experiences [5], [9].

Interpersonal conflict between developers and testers is inevitable and pervasive in software development process, given the inherent task and individual differences between them [9], [43]. Prior research has focused on the conflict between end users and IS staff [7], [23], [36], [44], [55], and conflict among IS staff [18], [50]; however, little research has focused specifically on the conflict between developers and testers.

Interpersonal conflict between developers and testers is likely to be negatively associated with software quality as well as job satisfaction of the conflicting parties, drawing upon prior related research [5], [9], [10], [13], [24], [50]. Software delivered with poor quality leads to unhappy end users, which in turn, may cause infrequent system use [22], an indicator of system failure [14], [15]. Job dissatisfaction is often positively associated with absenteeism, intention to leave, and actual turnover [21], [24], [25], [29]. Given the link between job dissatisfaction and turnover, it is especially important to study developer-tester conflict due to the current and projected shortage of IT talent [28]. It is also critical that IS researchers and practitioners thoroughly understand the sources of developer-tester conflict so that appropriate actions can be taken to mitigate its overall negative impact, which often leads to organizational ineffectiveness and inefficiency [35].

This research attempts to understand the nature of developer-tester conflict. In particular, it addresses one major research question: What are the sources of
interpersonal conflict between developers and testers in software development?

This paper is structured as follows: First, we provide a review of related work. Next, we describe the research methodology. We then present results, concluding with a discussion of findings as well as implications for research and practice.

2. Related work

This section reviews previous work related to the sources of interpersonal conflict. It starts with an overview of general literature on interpersonal conflict antecedents, then proceeds to software development related literature, and finally ends with literature related specifically to developer-tester conflict.

2.1. Overview of interpersonal conflict antecedents

"Conflict is a process in which one party perceives that its interests are being opposed or negatively affected by another party" [47, p. 517]. Conflict can occur at five different levels, including personal, interpersonal, intergroup, interorganizational, and international [17]. In this study, we focus on conflict at the interpersonal level, in which an individual is in conflict with other individuals [47].

Antecedents of interpersonal conflict can be sorted into two broad categories: Individual characteristics and interpersonal factors [47]. Individual characteristics that contribute to interpersonal conflict include personality [6], values [3], goals [11], [33], [54], commitment to position [47], stress and anger [16], and desire for autonomy [47].

Interpersonal factors known to contribute to interpersonal conflict include perceptual interface, behavior, communications, structure, and previous interactions [47]. Perceptual interface involves belief about another's intentions or motivations, regardless of accuracy [26]. Conflict arises when the other is perceived to have goals or intentions incompatible with one's own goals or payoffs [33], [51], to violate norms of equity or fairness [2], [47], or to harbor harmful intent [47]. In contrast, behavioral factors speak to actual harmful effects: Blocking of one's goals, or control attempts by the other [1]; the threat or actual loss of power caused by the other [8], [19]. As a conflict antecedent, communication is mediated by factors such as frequency, content, context, and facility of the communicators. Conflict results when low communication leads to ineffective coordination [32]; but high communication can produce misunderstanding, especially in a cross-cultural context [34], [46]. Structural factors refer to "contextual" characteristics of the social environment (e.g., laws, norms, customs, contracts, requirements, etc.) that constrain or enable interaction and help give it form and content. Previous interactions between parties can work with other interpersonal factors in destructive ways. For example, repeated resolution failures can lead to negative stereotyping, prejudice, and self-fulfilling prophecies that engender new conflicts [42], [45].

2.2. Software development research related literature

Rather than explore conflict antecedents, research into the role of conflict in team-based software development can be somewhat utilitarian: Often it focuses on the impact of conflict, conflict management, or associated factors on project outcomes, begging the question of how conflict arose to begin with. Some studies do consider conflict precursors, including (1) individual and interpersonal factors, (2) organizational, structural, and contextual factors, and (3) communication.

2.2.1. Individual and interpersonal factors

Sawyer [41, p. 159] states "individual characteristics" are "typically included in most models of conflict among software developers." This might appear inconsistent with general conflict research findings, but a closer examination reveals that very often what Sawyer and others [4], [37], [38] include in their models are not in fact individually held values, goals, commitments, emotions, personality scores or similar characteristics, but what would clearly be classified as "interpersonal factors" using the Wall and Callister [47] system: Levels of participation and influence, disagreements between team members, beliefs about other team members or the team in general, and so on. A possible exception is user-developer value divergence [52]; though the study focuses on software evaluation, it's likely that end product assessment differences were preceded by disagreements over other factors earlier in the project.

2.2.2. Organizational, structural, and contextual factors

Organizational, structural, and contextual factors are generally neglected in IS research though some studies note them in passing, e.g., distinguishing between users and developers as members of distinct
organizational cultures [5], [20], [37], remarking on the contextual nature of conflict and communication issues and the role of scarce resources, organizational rules, and procedures [55], or pointing out that systematic conflict due to divergent, synthesis-resistant goals is endemic to organizations and can result in distrust and conflict among team members [41]. Process factors such as requirements volatility receive attention as well [12], [49].

2.2.3. Communication

Communication has been identified as a critical success factor [53] with conflict resulting from miscommunication and misunderstanding between stakeholders, particularly users and developers. Such problems are minimized when developers work closely and consistently with users so that minor points can easily be clarified throughout the development process, while relationship building and increasing levels of trust defuse emotional negativity and conflict escalation [27]. However, as noted in the general literature, communication is a double edged sword—when mishandled it can promote, rather than prevent or resolve conflict.

2.3. Developer–tester interpersonal conflict

Sawyer [41] notes that while much IS and software development-specific conflict research has focused on interaction between user and IS staff, the relationship is transformed when software development is carried out by specialized firms so that it becomes just as important to understand conflict among IS staff themselves. This study focuses on the interpersonal conflict between developers and testers in software development.

The software testing process is inherently adversarial, setting the stage for inevitable developer-tester conflict. The role of the tester is to surface flaws and errors in developer code [9], [30]. It is a type of systemic conflict that develops as a result of lateral working relationships [32]. Tensions may arise between developers and testers because “they often have different and even divergent goals that are difficult to synthesize” [41, p. 160].

Software developers and testers are very different from each other in terms of mindsets, goals, experiences, and perceptions of their relative importance [9], [31], [40]. First, developers and testers often have different mindsets, as developers always think in terms of “building” something, whereas testers devise means of “breaking” what developers build. Second, developers and testers typically have distinct goals due to the difference in their job functionality. Developers usually seek to maximize “efficiency,” that is, to get the work done with the least amount of effort; for example, building the same functionality with the least number of lines of code or in the shortest amount of time. In contrast, testers usually seek to maximize “effectiveness,” that is, to deliver the end product with the best quality. Third, developers and testers may differ in experience; for instance, it is not uncommon that more seasoned developers work side-by-side with less experienced testers. And finally, developers and testers may have different perceptions of their relative importance in the organization; for example, testers often feel that they have to constantly work to gain the same level of respect as that of developers [9].

Individual characteristics testers identified as differentiating them from developers align with Wong’s [52] findings on developer-user value divergence. Testers described themselves as compulsive and detailed while characterizing developers as creative, temperamental, and apt to personalize their code, reacting to error detection as if to personal criticism. These tendencies find expression in work orientation: Testers focus on compliance with user requirements while developers look for ways to exploit the technical possibilities, sometimes violating specifications in the process.

Pettichord [31] provides a unique way to show that testers and developers are different by comparing a list of twelve traits that make good testers and developers.

To summarize the traits, good testers are expected to have broad knowledge of many domains, learn new things quickly, focus on user needs and behavior, think empirically, and concentrate on reporting problems. In contrast, good developers are expected to have specialized knowledge of product internals, gain understanding of new things slowly but thoroughly, focus on system design, think theoretically, and concentrate on understanding problems.

By analyzing in-depth field interviews with ten software testing professionals, Cohen et al. [9] categorized the sources of conflict between developers and testers into three conflict layers comprising the software testing process, people, and organization. Some of the conflict sources are: (1) Developer and tester competition for the scarce resource of project time, (2) Testers focusing more on user requirements whereas developers focusing more on technical requirements, (3) Differences in tester vs. developer “mental process and personality attributes” related to the process of software development, and (4) The role of differential power and politics, e.g., for some testers, “the struggle for recognition becomes part of the job itself” [9, p. 79]. These findings are both interesting and informative. However, the study has three
3. Research methodology

We used a qualitative research approach to address our major research question: What are the sources of interpersonal conflict between developers and testers in software development? We analyzed and categorized developer-tester conflict scenarios collected from professional developers and testers themselves.

3.1. Samples and data collection

Conflict scenarios were collected during a five-day system testing training program for a large, globally branded company. Most of the trainees are professional developers or testers, while the remainder are business analysts or IS managers. After finishing the “conflict and conflict management” training session, each participant was asked to describe in writing one of his or her own conflict experiences with others involved in software development. They were asked to specifically describe the issues and the reactions from both parties involved.

3.2. Data analysis and categorization

First the original hand-written conflict scenarios were transcribed into an MS Word file. Soft copies of the transcript were then distributed to two persons: (1) A doctoral student very familiar with the information systems development interpersonal conflict literature, and (2) A developer with a master's degree but no past experience interacting with testers and little knowledge of the interpersonal conflict literature. These two independently analyzed the transcripts, categorizing each according to a schema based on Cohen et al. [9]: Software testing process, people, and organization. The categorized scenarios were then reviewed and validated by a senior MIS faculty member actively involved in the system testing training program. The results presented below were synthesized from these three input sources. Divergence of opinion regarding categorization of a conflict scenario was resolved by majority rule.

4. Results

Sixty-four conflict scenarios were collected. Of these, fourteen were discarded after initial analysis due to irrelevancy; i.e., they are not related to developer-tester conflict. The remaining fifty scenarios were sorted according to reported sources of developer-tester conflict. Although initially guided by the "software testing process, people, organization" schema described by Cohen et al. [9], scenario analysis resulted in development of an alternative three-category conceptual structure that provides a better fit for the data: Process, people, and communication.

Figure 1 depicts the relationship between the three as a layered subset structure in which each constrains or influences the category or categories contained within it: (1) "Process" provides an organizational context for software development and is therefore depicted as the outermost layer; (2) "People" is contained within process because when enforced, process constrains individual and group behavior; and (3) "Communication" is the third layer because it is a key component of human behavior and thus a function of, and contained within, the people layer. This structure is intuitive and experientially supported. Working from the bottom up, project team communication is performed by people, whose choices of when, how, and what to communicate are constrained and guided by established process. Reversing direction and starting with the top layer, process directly influences team behavior (people) and thus indirectly influences the nature, frequency, content, direction, and effectiveness of communication.
The three layers or categories were then further divided into nine subcategories (see Table 1 for detail). Following is a presentation of findings in a category-subcategory-example format.

Process (documentation). Conflict arises when testers fail to provide adequate documentation so that developers can understand what is wrong, and how to reproduce the error.

Process (procedure). Some of the reported developer-tester conflicts are related to procedure. They include: (1) Tester opened a change request (CR) without thorough consideration, (2) Developer provided an incorrect version of code to testers for testing, (3) Developers and testers don’t know the process or don’t follow it correctly when a CR is opened, (4) Developers fix defects without properly documenting the code changes, (5) Developers and testers fail to involve third parties whose participation is needed for resolution, (6) Testers ask developers to change requirements after code completion, and (7) Developers access the test environment without invitation or clearance from the test organization.

Process (resource). Cohen et al. [9] noted that developer-tester conflict arises when developers and testers compete for scarce resources; two commonly reported factors in the general conflict literature are budget and time. In accordance with this observation, one scenario reports conflict that resulted from failure to include testing in the project budget. Another notes that last minute requirements changes and resulting delays in development squeezed the testing schedule.

Process (standards). Finally, lack of code standards or divergence in standards expectations may cause conflict between developers and testers. For instance, testers may use testing methods that make assumptions about the code while developers code according to the detailed design, not taking code standards or testing expectations into account.

People (attitude). Developers and testers have strong negative attitudes toward each other. One scenario describes a developer who holds such a low opinion of testers that he will not offer any suggestions, leaving them to work without the benefit of his support and expertise. In another scenario, a tester is said to derive negative enjoyment from finding errors in developers’ code. A third case describes a heated argument during a code review, in which a tester stated to the developer that “your code” has such and such defects. A fourth scenario states that whenever anything goes wrong, developers and testers point fingers at each other; testers blame a logic error in the code, while developers say it is a data issue. And finally, developers may constantly question the validity of the testing organization’s metrics.

People (view). Developers and testers can have conflicting views about appropriate levels of validation. At one point during a development status meeting, a frustrated development project lead almost screamed that testers were testing “too many combinations.” Developers and testers may also have different opinions about what defects should be fixed first, and how the fixes should be validated.

People (focus). Developers and testers bring differential focus to software quality assessment. Developers often focus on technical requirements; while as end user representatives, testers often focus on business requirements.

People (knowledge). Developers may accuse testers of lack of knowledge about the application, or lack of understanding of the business requirements.

Communication (communication). Many communication related conflicts between developers and testers were reported, including: (1) Test lead was left out of the communication loop, (2) Not knowing each other’s process or responsibilities, (3) Communicating only at the last minute, (4) Not communicating with each other about procedural changes, (5) Delay in communication of new requirements, (6) Failure to provide timely feedback, (7) Speaking different languages which leads to misunderstanding or not being able to understand each other, and (8) Developers making code changes without notifying testers.

Table 1. Developer-tester conflict sources in software development
<table>
<thead>
<tr>
<th>Category</th>
<th>Subcategory</th>
<th>Examples (number refers to original scenario)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>Documentation</td>
<td>Testers fail to provide documents to show developers what's wrong, and how to reproduce the errors. (39)</td>
</tr>
<tr>
<td></td>
<td>Practice</td>
<td>Testers open change requests (CRs) without thorough consideration. (30)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Developers provided incorrect version of code to testers. (46)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Developers and testers don't know the process or don't follow it correctly when a CR is opened. (9)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Developers fixed some defects without properly documenting what they did. (15)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Failed to involve third party responsible for the problem. (27)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tester asks developer to change requirements after code is complete. (58)</td>
</tr>
<tr>
<td></td>
<td>Resource</td>
<td>No budget for testing. (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Testing time was squeezed because of last minute requirements changes, causing development to drag on. (28)</td>
</tr>
<tr>
<td></td>
<td>Standards</td>
<td>Lack of standards and code expectations. (17)</td>
</tr>
<tr>
<td>People</td>
<td>Attitude</td>
<td>Testers have fun at developer expense by finding errors in their code. (18)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Developers question testing organization's metric for testing. (33)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code personalized: During a heated argument with a tester, a developer is told that &quot;your code&quot; has such and such defects. (49)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Developer who has a low opinion of testers. (57)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>What's wrong? Testers say it is a logic error in the code; developers say it's a data issue. (6)</td>
</tr>
<tr>
<td></td>
<td>View</td>
<td>Conflict views on the level of testing: During a development status meeting, the development project lead almost screams that testers are testing too many combinations. (19)</td>
</tr>
<tr>
<td></td>
<td>Focus</td>
<td>Different point of views on what defects should be fixed first, and how the fixes should be validated. (44)</td>
</tr>
<tr>
<td></td>
<td>Knowledge</td>
<td>Testers lack knowledge of a particular application. (3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Testers lack understanding of the business requirements. (8)</td>
</tr>
<tr>
<td>Communication</td>
<td>Communication</td>
<td>Test lead left out of the communication loop. (10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not knowing each other's process or responsibilities. (14)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Last minute communication. (16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Changes of procedure not communicated. (20)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Late communication of new requirements. (35)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Prompt feedback not provided. (47)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Different languages spoken. (50)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Developers changed code without communicating to testers. (62)</td>
</tr>
</tbody>
</table>
5. Discussion

The purpose of this study was to identify sources of interpersonal conflict between software developers and testers. Results indicate that developer-tester conflict antecedents fall into three major categories: Process, people, and communication. In particular, developers and testers experience conflict primarily because: (1) They don’t have standardized procedures to follow, (2) they have distinctive attitudes, views, foci, and knowledge bases, and (3) they don’t communicate effectively or efficiently.

5.1. Comparison of two models

Our research focus and data analysis were significantly influenced by Cohen et al. [9] so it is instructive to compare methodology and results between the two studies. In both a qualitative approach utilized open-ended questions to gather information about conflict experiences from professionals involved in software development projects. While Cohen et al. conducted in-depth field interviews with ten software testing professionals, we collected fifty written conflict scenarios from both testers and developers. Based on their analysis, Cohen et al. [9] constructed the three-layer conflict model depicted in Figure 2 below. Similar to our process layer (Figure 1), "organization" appears to function as a contextual base for the remaining layers; but in contrast to our model "software testing process" is contained within "people" and communication is not featured as a major conflict source.

![Figure 2. Layers of conflict [9]](image)

We believe that both our data (in terms of sample size and characteristics) and more particularly, our model represent a significant advance from the departure point provided by Cohen et al. [9]. The earlier study does not provide a rationale for the apparent ranking of the three layers (Figure 2) or the nature of the relationships between them, nor are the conceptualizations informing the arrangement readily inferred. While "organization" (layer 1) appears to function legitimately both as a major conflict source and as an overall context within which team conflict arises, positioning of the other two layers is problematic because experientially, it is difficult to support the assertion that "software testing process" (layer 3) is primarily a function or subset of "people" (layer 2). Rather, people act (or fail to act) in compliance with process, a key organizational feature. Nor is it inevitable that formal process will be significantly altered if the people assigned to a project are replaced; the usual expectation is that process will remain relatively stable regardless of team composition. This suggests that the ranking should be altered so that "software testing process" functions as the second layer, contained by "organization" (layer 1) while constraining the behavior of "people" (layer 3).

Playing devil's advocate for a moment, it's true that project team members can and often do function as owners of (rather than just clients constrained by) formal or informal processes existing at a variety of levels: Organizational, departmental, project, subteam, etc. It is also true that when established process does not adequately serve the needs of a particular project, or when incompatible processes interfere with team effectiveness, team members may choose to solve the problem with formal or informal process innovations. Just the same, vis-à-vis Cohen et al. [9] our layer ranking recommendations stand. By its nature, process constrains individual behavior so as to produce predictable results; that is the entire purpose of creating, documenting, and enforcing it.

A final contrast between this study and Cohen et al. [9] is our finding (consistent with the general conflict literature) that communication functions as a major source of conflict between developers and testers. It should be noted, however, that all but one of the communication-related scenarios we describe in the Results section above appear to be symptoms of an underlying cause: Lack of unified, appropriately designed and effectively applied project structure and process. Thus, through second layer mediation ("people," Figure 1), process gaps or flaws...
can negatively impact project communication, arguably rendering suspect the decision to classify such issues as arising from layer 3 (communication) rather than layer 1 (process). Our classification decision was guided by respondent perception as indicated by use of the keyword "communication" within the descriptive text.

5.2. Implications

The findings of this study have important implications for research. Conflict appears to originate from the organizational layers of process, people, and communication, which can be perfectly mapped to interpersonal factors identified by previous conflict studies including structure, perceptual interface, and communication, respectively. Our findings partially support those of Cohen et al. [9] in terms of people and process as conflict sources. However, this study suggests that communication is an additional major source of interpersonal conflict. This supports prior studies identifying communication is an antecedent of interpersonal conflict [6], [32], [34]. Therefore, it is critical that development managers and technical leads create an environment which facilitates effective communication between developers and testers and at the same time, decreases the chance of misunderstanding between them.

Our findings also have important implications for practice. Accurate identification of potential conflict sources enables development managers and technical leads to proactively target management strategies, preventing dysfunctional conflict from developing, or intervening before it can escalate and damage project outcomes, software quality, and ongoing working relationships. In short, our research supports the creation of both long-term strategies and short-term tactics to prevent, mediate, or resolve software developer-tester conflict.

5.3. Limitations

The study has several limitations. First, it is based on a convenience sample, limiting the generalizability of our findings. Second, the sample size is relatively small (fifty usable conflict scenarios), making it difficult to quantify the validity of the findings. Third, it didn’t differentiate conflict sources specified by developers from those described by testers, which would have enabled response comparisons. And finally, it didn’t collect standard control data (e.g., age, gender, tenure, education, etc.) that would support identification of sub-constituencies within the developer and tester groups, or of demographic differentials that could interact with other conflict factors in the software testing context. All of these limitations will be addressed in subsequent studies.

5.4. Further studies

Further studies are planned that will take a more systematic approach to collecting conflict scenarios, with a combined open-ended / structured online questionnaire administered via surveymonkey.com. We will distribute the survey link to potential respondents representing two key stakeholders in software development, i.e., developers and testers. Our targets are a minimum of 200 developers and 200 testers. Each respondent will be asked to describe a personal developer-tester conflict experience. Questions will include: (1) What was the issue that caused the problem? (2) Who was involved in the conflict? (3) How did you react; how did the other party react? (4) Was the conflict escalated to management level? (4) Was the conflict resolved; if so how and who was involved in the resolution? (5) How often does this type of conflict occur? We will also collect demographic data for the purpose of providing descriptive statistics for the sample and testing for potential control variables. When data collection is complete, we will analyze the survey responses to identify and categorize sources of conflict, comparing the perceptions of developers and testers in a structured, easy-to-understand, and easy-to-communicate way.

6. Conclusion

Interpersonal conflict between developers and testers is an ubiquitous phenomenon in software development. It can develop from factors associated with three major contextual layers – process, people, and communication. Although complete elimination of developer-tester conflict is impossible, overall software development effectiveness and efficiency can be greatly improved if conflict is managed at the source. To fulfill this goal, the first step is to clearly identify and thoroughly understand the sources of conflict. Our preliminary results have demonstrated the utility of this approach.

7. References


[28] Luftman, J. (2008) Yes, the tech skills shortage is real, InformationWeek, January 12, 2008.


A Simulation of Six Models of Engaging Testers in the Systems Development Life-Cycle

Mark L. Gillenson  
University of Memphis  
mark.gillenson@memphis.edu

Michael J. Racer  
University of Memphis  
mracer@memphis.edu

Sandra Richardson  
University of Memphis  
sandra.richardson@memphis.edu

Xihui Zhang  
University of Memphis  
xihui.zhang@memphis.edu

Abstract

In the early history of systems development, testing was confined to test the finished code. However, errors can be introduced at all stages of the systems development process and it is a well-accepted fact that the earlier in the process an error is found, the less costly it will be to fix. Thus, as the practice of systems development has evolved, there has been increasing interest in expanding the role of testing into the earlier systems development stages. While testing finished code is accomplished with established black box testing, white box testing, and code review techniques, testing in the earlier systems development stages involves less well-defined design inspection and walkthrough exercises. This paper describes the development of the simulation of a set of six models for embedding testing throughout the systems development process.

1. Introduction

An application system is a product that, like other “manufactured” products, must be developed in stages. A new application system has to be conceived, it has to be justified, the development must be found to be feasible within stated parameters, its requirements have to be analyzed, it has to be designed, the design has to be implemented, the implementation has to be tested, and the system has to be maintained over time. We have learned that collaborative and cross-functional development teams are very effective in new product development. There are many examples of collaborative product design, development, and manufacturing in traditional manufactured goods. Wheelwright and Clark [23] discuss an instance where manufacturing engineers were brought into the new product process for the design and development of a new camera.

Just as traditional manufactured products can benefit greatly from having representatives from each group responsible for the success of the product involved in the early stages of product development, so, too, can application systems. In particular, we suggest that personnel from the systems testing organization should be part of the new systems development process at all stages, for two major reasons. First, they can lend their expertise to “testing” or evaluating the “product” of each systems development stage, whether it be a set of requirements, a systems design, or a program. Second, they can influence the program design and implementation to make the job of testing the finished code more manageable and more effective, which will be to everyone’s benefit. These two propositions will guide the current research. It is also proposed that an additional impact will be that the testers will see themselves as stakeholders in producing high quality, defect-free products rather than just being the people who find bugs.

The question then remains, how can testers be embedded in the systems development process to obtain the most beneficial testing results? This paper will propose six models for embedding testers in the systems development process. It will then discuss the development of simulation models of these six tester embedding models.

Our goal in this study is to address a wide range of questions within the simulated environment, rather than experience the cost of actual implementation and testing, which would be quite impractical and costly. There are several issues that we wish to understand.
Given the characteristics of a software project, which of the proposed tester schemes is most beneficial? What is the likelihood that a significant error will be discovered and corrected, under a given set of conditions? What are the expected costs involved in implementing a particular testing scheme, as well as the actual costs for testers, the savings accrued by discovering errors, and the costs incurred by failing to discover errors?

2. Literature review

Systems development and testing has been utilized for decades. In the early days, writing a program was restricted to analyzing and coding [14]. There were no professional testers involved, and testing was usually performed by developers or users to ensure the code ran correctly and met the expectations. Eventually, systems development life cycle models were developed to provide a framework for systems development. Life cycle models divide the process of systems development and testing into well-defined phases or stages, typically including analysis, design, coding, testing, and implementation [10]. Testing eventually emerged as a distinct phase, and in some organizations, began to be carried out by professional testers. Testing was limited to the testing of code, after the code development was finished.

More recently, the trend in systems development and testing has been to engage testers earlier in the development process [13], [17], [22]. Also, there has been a further realization that testing is indispensable to ensuring software quality [2]. Furthermore, integrating testing early in the life cycle has the potential to catch and fix defects early in systems development. This practice is very rewarding because fixing a defect detected in later phases can cost up to 100 times more than if the defect was detected earlier in the development process [18].

The systems development and testing life cycle models include sequential models (e.g., waterfall model, traditional V model, PV model), progressive models (e.g., phased model, iterative models (e.g., spiral model), and agile models (e.g., XP). The waterfall model includes a set of sequential phases that flow downward like a waterfall. The traditional V model has the same set of phases as that of the waterfall model, however, with the traditional V model, each testing activity is mapped to some development activity [8]. With both the waterfall model and the traditional V model, testers are involved only in the traditional testing phase, and testing does not begin until coding is completed. In these models it is very difficult to go back to previous phases when changes must be made or new features need to be added. Thus, using the waterfall model or traditional V model often results in projects that run over time and over budget [17].

The PV model is similar to the traditional V model; however, the PV model separates each testing activity into two parts: test planning and test execution. Test plans (i.e., test specifications) are developed along with each development activity, and the test execution is exercised in reverse order after coding is done [15]. The PV model tries to engage testers earlier in the systems development process, and compared to the traditional V model, it has the advantage that test plans can be completed during the early development phases, and therefore the overall testing time can be shortened.

The progressive model, the iterative model, and the agile model are all iterative and incremental approaches to systems development, which require full integration of testing and development [16]. The intent is to produce high quality systems in a cost effective and timely manner while meeting the changing needs of the end users.

In a field study of thirty-two business application systems, McKeen [10] found that “systems which spent more time in the analysis phase required less time to code, resulted in greater user satisfaction, and were developed in agreement with established budgets and deadlines” (p. 47). Bringing testers into the analysis phase has the potential to aid in this endeavor. Waligora and Coon [22] presented quantitative evidence that, by starting testing earlier in the development life cycle, project performance, in terms of cost, cycle time, and quality, was improved without sacrificing the quality of the end product.

Cusumano and Selby [4] noted that “since the late 1980s, Microsoft has used variations of the synch-and-stabilize approach to build Excel, Office, Publisher, Windows 95, Windows NT, Word, Works, and other products” (p. 60). Two of the major features of the synch-and-stabilize approach are (1) product development and testing are done in parallel, and (2) there are frequent synchronizations (daily builds) and intermediate stabilizations (milestones). These two features require “[teams] of programmers and testers [who] frequently synchronize and periodically stabilize the changes they make to products in progress” (p. 53). Pyhajarvi and Rautiainen [13] viewed testing as the means to maximize customer satisfaction and provide feedback for process refinement, in addition to finding defects. They argued that testing is “an integral activity in software development” and they recommended that “testing should be included early in software development” (p. 33).
Vijay [21] provides us two very convincing reasons why we should bring testing earlier into software development. (1) “Testing activity consumes 40% of the whole project time” (p. 2). If we want to cut down the cycle time of the systems development process, the testing phase is a “more lucrative timesaving phase as compared to [the rest of the systems development process]” (p. 2). (2) Some of the testing activity has to wait until coding is complete, however, “around 60-75% of the activities (like the test-plan, test design) don’t require any code” (p. 2). Thus, if we can bring testers earlier into the systems development process, let them work on the test plan and test design, this would dramatically decrease the total cycle time of the systems development process.

In another relevant vein, there is literature that discusses manufacturing engineers being involved in new product development. For example, Nemetz and Fry [12], in describing flexible manufacturing organizations, describe manufacturing personnel as being part of a multifaceted team. Milgrom and Roberts [11], talking about “design for manufacturability,” say, “…products are developed by teams composed of designers, process engineers, and manufacturing managers” (p. 515). Several books have been written on new product development, including Cooper [3], Ulrich and Eppinger [19], Urban and Hauser [20], and Wheelwright and Clark [23]. In particular, as mentioned earlier, Wheelwright and Clark [23] describe the development of the Kodak FunSaver I single-use camera. Sample quotes from this description include, “Concurrent with these decisions about product line architecture, the development team began to focus on issues of manufacturing” (p. 112). “Design and manufacturing tasks had to be carefully integrated” (p. 112). “To deal with these challenges, a manufacturing engineer was co-located with several parts designers and a tool designer, all of whom were dedicated to the project” (p. 113). “…and part manufacturing people could easily gain early access to part and tool design decisions, enabling them to estimate costs and identify and evaluate automation options” (p. 113).

The broader issue is that of cross-functional or matrix organizations. Ford and Randolph [5] provide a good survey of the literature up to that point. Griffin [6] said, “…the use of cross-functional teams, is one of the factors that has been derived through a grounded theory approach by multiple researchers as purporting to decrease new product development cycle time” (p. 27). Samplings of other papers on this topic include Brown and Eisenhardt [1] and Gupta et al. [7].

There is one more basis for this work of which we are aware. Jordan Lewis’ book Trusted Partners addresses the idea, at several levels, of interpersonal relationships being a key factor in a successful business enterprise. The concept in the current proposal would not only be for the testers embedded in earlier stages of the systems development process to see themselves as stakeholders, but also for the analysts and designers in the earlier stages seeing the testers as “trusted partners” who can help them in their work and add value to the outputs of those stages.

3. How to engage testers early and throughout the systems development process

There are often variations in the stages of the systems development process as practiced by different organizations. For example, the first stage may include business justification, feasibility in several dimensions, and determining information requirements. The analysis stage can include varying levels of detail. The design stage may be split between code design and database design, or it may be one unified process. In fact, the stages of the systems development process can vary in name, in number, and in content to suit the needs of individual organizations. Generally, different systems development process variations are ultimately similar in their effect. For our purposes, we will define five systems development process phases or stages, as follows:

- Business Requirements: The set of specifications of what the business unit expects the application to accomplish.
- Systems Requirements: The systems analysis stage in which the business requirements are translated into graphical formats that show processes and data flows.
- High-Level Design: The specification of the code modules and their functions, and the flow of data among the code modules.
- Detailed Design: The design of the functions within each code module.
- Implementation: The programming of each code module.

Having established a systems development process framework with which to work, we now turn to the nature of the participation of the testers at each systems development process stage. This concept can be broken down into two possibilities, as we noted earlier, one or both of which can be practiced. One, which is applicable at all of the stages, is the
idea of testing the output of the stage. Do the requirements make sense, do they meet a set of accepted standards, and what are those standards? Do the diagrams that result from the systems analysis stage flow correctly and make sense? Have the systems analysis diagrams been constructed to meet accepted standards and what are those standards? Are the program design specifications and the database design acceptable?

The second possibility regarding the nature of the participation of the testers is directed toward the creation of systems that will lend themselves to being more easily and effectively tested. This would begin in the systems analysis stage and significantly impact the systems design stage. This could have profound implications in systems testing, in streamlining the process in general, and in specifics such as determining the test datasets to use.

There are a number of factors to consider in formulating a model to use in integrating testers into the application development team and process. An initial factor is whether a company believes there is value in embedding testers at all levels of the systems development process. As we have stated, we believe that the arguments for doing so are compelling and so for our purposes we will assume that this is the case.

Assuming there is significant value in embedding testers earlier in the systems development process, one factor in deciding which model to use is the skill set of the individuals in the testing department or organization. We will assume that any tester assigned to represent the testing organization in an application development project is skilled in testing in at least one of the application development stages. This can even be extended back into the education and training backgrounds of the individuals. It is as difficult to imagine a tester without a business background leading a business requirements review, as it is imagining a tester without a programming background leading a code review. Pursuing this further, the Tester Embedding Model chosen will also depend on whether the company’s testers tend toward breadth of skill or depth of skill. Are the individual testers expected to have skills that range from business to technical skills or are their skill sets expected to be narrowly focused?

Another factor is the amount of resources the company is willing to invest in testing. This certainly will depend on the company’s commitment to testing and on the size of the application development project. Generally, in this regard, more would seem to be better; however, even a company that takes testing seriously would not want to overwhelm the application development teams with testers.

With the previous discussion as background, we propose six “Tester Embedding Models” for embedding testers early and throughout the systems development process.

3.1. Model 1: “The single tester model”

As the name implies, in The Single Tester Model, one tester is assigned to an application development project and stays with it through all of its stages. This has the advantage of continuity as one person begins learning about the project from the very beginning of the business requirements phase and continues building her project knowledge through each successive stage. The disadvantages of this model include a project over dependency on one person and the expectation that the one person must be well-versed in a breadth of skills ranging from requirements analysis to program design and programming. Assuming you can find such a person, if she gets sick or leaves the company, the project is left in the lurch. And, if The Single Tester Model is attempted without a sufficiently broadly skilled tester, then clearly the principle of having strong testing expertise at each development stage will have been defeated.

3.2. Model 2: “The specialist model”

If The Single Tester Model is one extreme, then The Specialist Model is the other extreme. In The Specialist Model, a different, highly specialized tester works on the application development project in each of its stages. Presumably, each tester is a true expert in the work being done at their particular
development stage and thus the advantage is the level of testing expertise that can be applied at each stage. Conversely, each tester does not have to possess a broad skill base. However, there are some disadvantages as well. One disadvantage is that each Specialist must learn the nature and details of the project when they cycle onto the project. Another disadvantage is the lack of communication between the testers in the different stages. Indeed, it is this lack of communication that inspires certain aspects of the next four models.

3.3. Model 3: “The leapfrog model”

The Leapfrog Model is designed to overcome some of the problems associated with both The Single Tester Model and The Specialist Model. The Leapfrog Model begins with Tester A, who is a requirements testing specialist, working on the Business Requirements development stage. Tester A continues working on the project in the Systems Requirements stage, where she is joined by Tester B, whose expertise is more geared towards systems analysis and high-level systems design. As both Testers A and B work on the Systems Requirements stage, Tester A is able to gradually transfer her project knowledge to Tester B. At the end of the Systems Requirements stage, Tester A leaves the project. At the beginning of the High-Level Design stage, Tester B is joined by Tester C, whose expertise is focused on both high-level and detailed systems design. Similarly, at the end of the High-Level Design stage, Tester B leaves the project and at the beginning of Detailed Design stage, Tester C is joined by Tester D, whose expertise is focused on detailed program design and programming. Tester C leaves the project at the end of the Detailed Design stage. An advantage of The Leapfrog Model includes having two testers working on each development stage except for the first and last stages. At each of the intermediate stages there is the opportunity for one tester to gradually transfer her project knowledge to the next tester. However, as with The Specialist Model, The Leapfrog Model assumes the availability of a stable of relatively specialized testers and, with two testers involved at each of the intermediate stages, it is even more resource intensive.

3.4. Model 4: “The balanced bifurcated model”

In The Balanced Bifurcated Model, there are two testers, A and B. Tester A has a broadly-based systems analysis background that extends to requirements on the one end and to high-level systems design on the other. Tester B has a programming background that includes the higher levels of systems design. In this model, Tester A begins at the Business Requirements stage and stays with the project through the High-Level Design stage, after which she leaves the project. Tester B joins the project at the High-Level Design stage and continues with it to its conclusion in the Code stage. With both Testers A and B working together in the High-Level Design stage, they have the opportunity to transfer project knowledge from A to B. Their skill bases must be broader than those of the testers in either The Specialist Model or the Leapfrog Model, but not as broad as the testers in The Single Tester Model. While there is a shift back toward the problem of over-dependence on individuals as in The
Single Tester Model, there is also not as much of a resource drain as in either The Specialist Model or the Leapfrog Model.

3.5. Model 5: “The top-loaded, unbalanced bifurcated model”

The difference between The Balanced Bifurcated Model and the two Unbalanced Bifurcated Models is the point of hand-off of responsibilities. The principle of The Top-Loaded, Unbalanced Bifurcated Model is that one tester, Tester A, will work with all aspects of the project through and including the Detailed Design stage. Then, Tester B will join her in the Detailed Design stage and be responsible for testing in the Coding stage. This model heightens the personnel dependency issue, plus Tester A must be very broadly based. The clear advantage of this model is it provides specialized testers whose sole purpose and total focus is to look at the detailed design and then work in the highly technical pursuit of code testing.

3.6. Model 6: “The bottom-loaded, unbalanced bifurcated model”

In The Bottom-Loaded, Unbalanced Bifurcated Model the point of hand-off is the Systems Requirements stage. That is, Tester A handles the Business Requirements stage and the Systems Requirements stage. He is joined in the Systems Requirements stage by Tester B who begins working in this stage and then follows the project to its conclusion in the Implementation stage. In many structural respects it is similar to The Top-Loaded, Unbalanced Bifurcated Model, except that now, one tester, Tester A, is focused on business requirements, and the other tester, Tester B, begins with system requirements and then proceeds through both of the design stages and the implementation stage.
project, and the company's dedication to testing, the testers described in any of the six models may well serve as "test leads" and bring additional testing personnel into the application development process as required. In all cases, this could obviously be simply a matter of handling the volume of work at hand. Also, in all cases, this decreases the dependency on only one or two people at any given development stage. Naturally, it also increases the amount of resources expended in testing. In addition, in Model 1, The Single Tester Model, and to a lesser extent in Models 4, 5, and 6, the three bifurcated models, it could involve calling in specialists to supplement the skills of the test leads in those models.

The other consideration has to do with the overlap between the testers at the various intermediate stages of development. Models 1 and 2 have no overlap. If we eliminate the overlap in Model 3, The Leapfrog Model, it effectively reverts to Model 2, The Specialist Model. The issue is whether in Models 4, 5, and 6, the three bifurcated models, the overlap could be eliminated. This has the advantages of reducing the expenditure of resources and of not requiring the testers to be quite so broad in their skill sets. On the other hand, there will clearly be a cost in losing the transfer of project knowledge from one tester to the next that is provided by the overlap. Perhaps the elimination of the overlap is most appealing in Model 5, The Top-Loaded, Unbalanced Bifurcated Model. Eliminating the overlap in Model 5 would isolate Tester B who is responsible for code testing. Historically, this responsibility has in many organizations been the only real job for testers. Even in those instances in which testing is embedded early and throughout the systems development process, the nature of code testing and its specific and highly technical techniques make it a candidate for being largely separable from the other systems development process stages. In fact, as a practical matter in today's IT environment, code testing can be looked upon as a candidate for outsourcing, which would fit the variation that we might now call The Unbalanced Bifurcated Model Without Overlap.

4. Simulations of the six tester embedding models

One of the most widely-used tools in analyses today is simulation. The fundamental goal in simulation is to mimic the essential elements involved in a process. A simulation model is very user-friendly, in the sense that the components of the simulation reflect actual components of the process being modeled. Consequently, simulations are extremely robust. Simulations allow for the collection of data at all stages of the process, so hidden factors might be easily revealed. And the most appealing attribute of simulation is the "what if...?" capability. That is, the user has the ability to alter conditions in the model, identify how the changes influence the outcomes, and project that same behavior to the real-world. Once the fundamental model is in place, the potential for investigating these alternative scenarios is practically limitless.

Within the context of software testing, the progress of a piece of software from conception is a good example. From the model development standpoint, there may be components in the simulation that mimic contributors at every stage (business requirements, software requirements, high-level design, detailed design) and processes developed that show how these various factors interact and influence the quality of the final implementation. From the quality standpoint, we may capture the testing layer, and show how the performance of the various testers embedded throughout product development ultimately and collectively influence the final product.

The "what if?" standpoint is often the most significant element of a simulation. For example, a simulation model can be developed to show how a change in testing strategy (e.g. single-tester vs. specialist) can influence both costs and product quality. If we can understand the system better, we can make decisions on the front end of the suitability of a particular tester framework.

While econometric and statistical models have capabilities to capture and explain some very basic relationships, they lack the capability to address detailed relationships within the process. For either of these models, it is extremely difficult to reveal the nature of interactive relationships (e.g. "How do certain business requirements influence the quality of the product when a bifurcated testing system is used?") Compounding this, as econometric and statistical models increase in complexity, their comprehensibility declines - the model becomes less understandable to the original user, and consequently, less defensible.

In summary, the fundamental value of a simulation lies in these three characteristics:

- dynamic: a simulation truly mimics the real-world process itself, allowing for in-depth, yet understandable modeling
- interactive: a simulation allows the user to capture the impact of complex relationships
• "what if?": a simulation allows the user to "witness" the impact of a change in conditions, within the computer, rather than an expensive, impractical, real-world test.

The key to simulating the alternative testing environments is understanding the system components and their attributes, and how those attributes interact. Some of those basic components and their attributes are the software length, complexity, number of programmers required, and application type (business, engineering, recreation, etc.). In addition, there is the experience and expertise of the tester, and the nature of the testing environment (i.e. the variations expressed in the six tester engaging models.) Finally, there are the issues surrounding the errors to be found in the testing, such as their location, their severity, and the period at which they were introduced (business requirements, software requirements, high-level design, detailed design, coding.)

5. Conclusions and further study

Overall, the goal is higher quality application code with fewer errors produced, leading to higher quality applications being introduced into production. An additional contribution will be testers seeing themselves as stakeholders in the quality of the finished applications by virtue of their work throughout the systems development process. This will lead to the further development of systems testing as a recognized and respected specialty within information systems organizations.

At the time of writing, we are developing a questionnaire that we will be used to survey a small number of experts in the field with the goal of developing needed parameter values in the simulation models.

6. References


Engineering Laboratory,” NASA’s Software Engineering Laboratory.

A Systematic Methodology for Aligning Testing and Development

Colin Onita
363 Fogelman College Administration Building
University of Memphis
Memphis, Tennessee 38152
Tel: 901-678-4658
cgonita@memphis.edu

Jasbir Dhaliwal
301 Fogelman College Administration Building
University of Memphis
Memphis, Tennessee 38152
Tel: 901-678-4613
jdhaliwl@memphis.edu

Abstract
Alignment between testing and development has been raised as an issue for successful systems development. Missing however have been actionable how-to methodologies for assessing and enhancing such alignment. This paper attempts to fill this gap by describing a systematic methodology for this purpose that can be pragmatically used by systems development organizations. This methodology considers alignment at both strategy and execution levels. By dissecting alignment into internal (within) and external (between) categories, it outlines pragmatic mechanisms by which the coherence between the internal components of developer-tester alignment can be assessed and managed. At the strategy level the aspects of these components that have to be aligned include scope, governance and resources. At the execution-capability level, aspects that have to be aligned include processes, architecture and skills. Specific mechanisms and tools for ensuring that the execution capabilities of testing groups are aligned with their stated strategies are identified. Systematic mechanisms for tying testing strategy and capabilities to development strategies and capabilities are also discussed.

1. Introduction
The need for accurate, real time, reliable IT systems that provide support for business needs and allow seamless communication and dissemination of information and knowledge is an prerequisite for gaining competitive advantage in today’s global, fast pace business environment [23], [28]. In spite of their importance and their critical operational, tactic and strategic role, many new and old IT systems have failed in part or totally [34], [42]. According to Gartner [18], on average, 7% of software functionality that was paid for is actually used, while 85% of IT projects failing to meet objectives (32% being cancelled outright). Dhaliwal and Onita [13] posit that many of these development failures are a result of poorly executed development process. These employ either inadequate development models or flawed implementation due, in part, to the lack of proper testing and effective collaborative mechanisms between testers and developers. These issues have not yet found a proper solution, due, in part, to a lack of a methodology that would allow the analysis and correction of software development processes. A review of the testing and development literature reveals that relations between the development and testing functions are lacking for projects of medium and large magnitude, where testing is separate from the development activities [9], [27]. The literature does not provide any empirical data or methodology pertaining to how the two functions can be aligned in both strategic and execution terms. Most research in the area of systems testing deals with technical issues related to code testing [17], [38], and other technical aspects. Similar issues have been investigated in the area of business and information systems alignment, under the name of Strategic IT alignment. This type of alignment arises when the business goals and
activities of an organization match information systems that support them [25]. This has been a top priority for CEOs and CIOs and has been studied extensively [26], [36], [21], [31], [37], [41], [6], however, issues of alignment within the IT shop have been entirely ignored in practitioner and academic literature. Without proper alignment between the elements of the IT shop (mainly development and testing), the alignment between business and IT can be tenuous at best [13].

IT alignment research focuses largely on management issues, without investigating components of the alignment process. Henderson and Venkatraman’s Strategic Alignment Model [16] looks at connections and linkages between business and IT. Other authors [1] analyze the Strategic IT alignment through an enterprise architecture knowledge management framework, and identify enabling and inhibiting factors of strategic alignment [20]. Reich and Benbasat [29] tries to measure alignment by looking at internal consistency and external validity. However all this research ignores the alignment issues within the IT department and assumes harmonious internal operation within the corporate IT unit.

This paper proposes a development-testing alignment (DTA) methodology which posits that such alignment leads to beneficial effects such as lower costs and shorter time of development, greater system quality, fewer errors and a better relationship between the corporate IT unit and customers in business functions who have commissioned new systems (Figure 1). Alignment models and measurements have been studied in other related contexts [16], [30], [32] but never within corporate IT units and specifically between the development and testing functions. The paper therefore decomposes DTA alignment into a series of aspects for the purpose of assessing and analyzing each of the construct. These aspects are drawn from the overarching framework developed initially from prior literature [13]. The DTA methodology will allow IT managers to improve the effectiveness of testing and development by both synergistically integrating testing in the development process and by aligning the testing and development units in terms of strategy and execution capability.

Figure 1: DT alignment and expected outcomes.

2. Key Concepts of Developer-Tester Alignment

Alignment between the development and testing functions can be defined as the strategic and operational fit between the development and testing functions on components of strategy and capabilities [13]. Since systems development as well as systems testing are integral parts of the corporate technology acquisition strategy, they too have to be aligned to ensure business success. In many organizations, there is a gap, or misalignment, at the strategic and/or execution level, between the development and testing groups as well as between individual testers and developers. To correct these misalignments, this paper proposes a methodology, grouped under the DTA model [13] that draws upon the strategic alignment model initially proposed by Henderson and Venkatraman [15]. This DTA model focuses on the fit between the development and testing functions.

A key goal of this research is to develop a methodology for applying these concepts within the corporate IT unit tasked with building and implementing business system applications.

A high level of integration of business and IT plans facilitates communication and collaboration [35]. Similarly, in the areas of development and testing, a high level of integration and correspondence at both the execution and strategic levels may also facilitate communication and collaboration. Integration represents the level of linkage between development and testing, while correspondence represents how closely their capabilities mirror and complement each other. Varying levels of alignment can either promote or hinder integration and correspondence. This is a common characteristic of all alignment models in the literature as verified by Reich and Benbasat [30].
Figure 2 details the key structural and flow components of the DT alignment model for development and testing within the corporate IT unit. This model decomposes the alignment of the development and testing functions along three key flow dimensions: 1) strategic alignment, 2) capabilities alignment, and 3) strategy-execution alignment.

The first structural component, development strategy, looks at strategic choices of the development function. This component is comprised of three key aspects: the scope of development, governance of development and development resources. Here the scope of IT development is defined in terms of IT goals that support the business strategy. The formal organizational makeup of the IT development departments and teams, buy or build decisions, as well as the overall competencies and responsibilities of the development and testing groups are also taken into consideration.
The second structural component, development capabilities, has three key aspects: development process, development skills and development architecture. These directly impact the applications being developed, the tools used in development processes, as well as the models or frameworks employed in the development process. Decisions about development models, such as SDLC, RAD, prototyping, etc., the skills and competencies of the development personnel are also considered at this level.

On the testing side, the third structural component, testing strategy, focuses on three key aspects: the scope of testing, issues regarding responsibilities and resources and the governance and reporting structure of the testing function.

The fourth structural component describes the testing capabilities and has three aspects, testing processes, testing skill/competencies and testing architecture. The specific methods of testing (traditional, V-mode, iterative), as well as choices about testing tools, architecture, communication structure, etc. are considered and analyzed from an alignment perspective. The individual skills of testing personnel are also assessed.

In conclusion, the top two quadrants of Figure 2 represent the strategy level while the lower two quadrants represent the capability level. The left two quadrants represent the development function while the right two quadrants represent the testing function.

DT alignment implies that all four dimensions are matched in capabilities, resources, structure, etc. This does not mean that they have to be similar, but that testing complements development and acts as an enabler of development success by providing verification, validation and bug-finding services.

Each structural component (quadrant) of the Alignment Model deals with alignment from a double perspective: strategy/capabilities and development/testing. DT Alignment also has three flow dimensions (as represented by the numbered vertical and horizontal arrows): strategic alignment, capabilities alignment, and strategy-execution alignment.

2.1 Strategy Alignment

Strategy Alignment (flow 1) deals with matching the development strategy and the testing strategy. It focuses on aligning the testing and development functions to ensure that the testing function acts as a validation and verification tool for the development activities. The resource endowment of both functions has to be similar, the purpose and scope of development and testing has to be coherent, and the level of authority and autonomy (governance) of the two functions has to be complementary.

2.2 Capabilities Alignment

Capabilities Alignment (flow2) focuses on matching the capabilities of development and testing at the operational/execution level. It deals with the operational “linkage” between development infrastructure and processes and testing infrastructure and processes. Processes, skills and architectures of the development and testing functions have to be synergistic and complementary. Development tools and methodologies and philosophies have to match testing tools and methods. The skills of testers and the procedures of testing have to supplement and support the skills of developers and procedures used in development. Dysfunctional relations between development and testing are often the result of misalignment of capabilities, especially when one unit is capable to execute its task better than the other.

2.3 Strategic Execution Alignment

2.3.1 Strategy Execution Alignment of Development

Strategy-Execution Alignment within development (arrow 3a) focuses on the ability of the development function to execute its stated strategy. Strategic decisions have to be supported by the operational infrastructure that is in place, by the competencies of the development staff and by the tools and methodologies used in the software development process.

2.3.2 Strategy Execution Alignment of Testing

The testing Strategy-Execution Alignment (arrow 3b) deals with the ability of testing capabilities (competencies, tools and methodologies) to support the execution of stated testing strategies. Testing strategies have to be executable, and testing capabilities have to empower and support the strategic goals and decisions.
Prior studies have identified additional influences that impact the alignment between the components of the DTA framework. Reich and Benbasat [30] found that shared domain knowledge and cross-participation in decision making of the business and IT functions are enablers of alignment. Similarly, when talking about alignment between testing and development, shared domain knowledge of development executives and testing executives will, most likely, positively influence the level of alignment between the two functions. When testing executives have development experience and/or knowledge, and when development executives have testing experience and/or knowledge, their decisions will lead to better alignment of the functions. This is especially true if participative decision making takes place, where testing executives are part of the development functions decision making and development executives are part of the testing function’s decision making process.

3. A Methodology for Achieving DT Alignment

Improving any process can be facilitated by proper planning and by following detailed and fitting methodologies and techniques. Based on case study and field study approaches [4], this study proposes a methodology for achieving DT Alignment (see Figure 3). The methodology is derived from a survey of the literature from Strategic Alignment [15], [8] Testing [16], [3], [4] to Project Management and Information Systems development methods [7], [9], [10], [12], [32], [33], [34]. To improve the reliability and validity of this methodology, alignment case studies and field studies were conducted and real life examples are given to improve the applicability of the methodology. A list of techniques is also mapped onto each step of the methodology. While not exhaustive, this toolbox gives IT managers a good idea about the available techniques that can used when attempting to secure high levels of DT Alignment/Realignment strategy.

Based on the DTA methodology the next sections present a detailed description of each of the three phases that make up the methodology (Figure 3). Phase 1 is the initiation stage, where misalignments in the software development process are identified, the decision to pursue a DT Alignment/Realignment approach is taken, and goals for the DT Alignment/Realignment process are set. Phase 2 details the steps that need to be taken to properly investigate the sources of misalignments that disrupt the software development process. Phase 3 involves the design and implementation of an action plan to address the misalignments identified in Phase 2.

3.1 Phase one: Identification of Misalignment

3.1.1 Symptoms/Triggers

When software development efforts fail, information systems leaders must pay careful attention to the underlying reasons as to why success was not achieved. At times, there are other organizational, project management or technical reasons besides misalignment between developers and testers that have caused the failure. Often however, DT misalignment is the real underlying reason behind organizational, project management and technical causes of software failure. For example, project schedules may not have been met because of miscommunications between developers and testers or a technical reason caused software failure because certain technical features were inadequately tested due to DT misalignment. It is therefore critical that IT leaders develop the skills necessary for identifying situations where developer-tester misalignment is an issue. Being familiar with symptoms of developer-tester misalignment and mastering several tools for isolating such misalignment are therefore key requirements for success.

Typical symptoms of developer-tester misalignment that serve as triggers for software development failure include:

- high incidence of developer-tester conflicts
- disagreements about respective roles of developers and testers
- high levels of false-positives in “bug reports”
- low morale amongst testers
- unrealistic testing schedules
- higher comparative turnover amongst testers as against developers
- incompatible or unused automated testing tools
- regular software failures occurring whenever changes are made to code
- significant disparities in remuneration of developers and testers
- low levels of testing knowledge possessed by developers
- testers without backgrounds in software development
- testing budgets that are unrealistic
In our experience, many software development managers fail to recognize the significance of these largely because their “world-view” does not distinguish between the complementary roles played by developers and testers. The implicit assumption is that the two groups are indistinguishable in terms of role and responsibility.

Arising from the above, the initial focus of our methodology is on methods that help IT managers to distinguish between development and testing to enable them to form initial assessments about the underlying quality of the organizational and working relationships between them. These initial assessments help reveal if developer-tester misalignment is indeed a key reason for failures in software development.

3.2 Phase two: Assessment of DT Alignment

The second phase of the DT Alignment methodology investigates the current level of alignment between and within the two functions – development and testing. During this phase, the processes, resources, organization and human resources are looked at to identify any potential misalignments that are a cause of the symptoms identified in Phase 1.

Stage 2.1: coherence WITHIN components:

Before alignment can be investigated between the functions of development and testing, the components internal to the development and testing functions have to be analyzed to see how well they support each other and how well they support the business needs.

- 2.1.1 assessing internal alignment/coherence of development strategy

The logical place to begin the investigation of DT alignment is with the coherence of strategic aspects within the development function. We analyze issues regarding the scope that development activities have, how these activities cover the needs of business and also how the authority and responsibility relegated to the development department matches the goals that business has set for development. This is also closely related to the resources available for development to meet its goals and also related to the governance structure of the development function that has to provide adequate support for development activities.

- 2.1.2 assessing internal alignment/coherence of development capabilities

The strategic choices made by the upper levels of development management have to match the capabilities and processes of the development activity. For this reason assessing the level of internal coherence of the capabilities component of development is paramount in analyzing the level of DTA. We investigate the degree to which the processes, skills and architecture used in the development process are coherent with each other to provide the required level of support. The three aspects of development capabilities are closely intertwined. Development processes have to be supported by certain types of development methodologies which require appropriate tools and skills. If there is a break in this chain, misalignment within the capabilities of development will arise.

Having internal alignment within components is a prerequisite to any kind of alignment between functions, and therefore has be investigated first.

- 2.1.3 assessing internal alignment/coherence of testing strategy

Similar to Step 2.1.1 for the development activities, an assessment of the internal coherence of the testing group’s strategy components is undertaken here. This involves assessing the balance, fit, match and covariance between the scope of the testing unit, its governance, and its resource endowment. It is important to ensure that it possesses adequate resources to be able to meet the scope of testing activities assigned to the group at the strategic level. It is often necessary to reallocate scope responsibilities at the strategic level to balance resource allocations made. For example, in some IT environments – alpha and beta testing responsibilities are assigned to line functions (based on product lines) or to marketing because of strategic resource issues rather than to the testing group. In such situations, the governance structure for alpha and beta testing also has to be aligned in the sense that testing staff undertaking this responsibilities report to appropriate management level. Another example pertains to the responsibility for unit testing as against integration testing. In many contexts, the strategic responsibility, and the associated resources, for the former are handed to the development group. In other situations, these can just as well be assigned to the testing group. Strategic scope, resources and governance issues are intricately intertwined and there is a clear need for alignment to ensure success.
- 2.1.4 assessing internal alignment/coherence of testing capabilities

Similar to Step 2.1.2 for the development group, an assessment of the internal coherence of the testing group’s capabilities is necessary. This involves assessing alignment between three key aspects that together determine the execution capability of the testing group to perform its role. These are its testing processes; its portfolio of skills and competencies; and its technology architecture and technology. For example – it is imperative that testing processes for load and stress testing be supported by appropriate staff skills and automated testing tools that match. If Mercury’s “Loadrunner” is the software being used, it must have a prescribed role specified in the testing process manual and testing staff must be trained in its use. This appears to be a significant issue in industry. Forrester [43] reports that, while enterprises do purchase test automation tools, they have a difficult time applying them for conducting automation of functional test scripts. In our view, a key reason for this is that automation tools are being purchased without careful consideration of their alignment to testing processes and available competencies for automated testing. In our experience, senior IT leaders are concerned about the execution capability of their testing units and would like to see their testing managers place just as much emphasis on flawless testing execution as on the strategic management of testing in relation to development. The ability to execute testing plans provide much needed assurance to IT managers so that they can rest assured that tested systems are reliable and functional.

Stage 2.2: Strategy Execution Alignment Within Function

Once the internal coherence of the components of the development and testing functions has been assessed, the alignment between the strategy and capabilities components of each function has to be investigated. This is crucial, since, without proper alignment between the strategy and its execution, the goals pursued cannot be attained, because of a lack of support from the actual implementation of strategy. This is a hierarchical requirement where the strategic component dictates what the capability component has to enact.

- 2.2.1 Strategy Execution Capability of Development

Given internal coherence of the components of the development function, the link between development strategy and implementation can now be investigated. The correspondence between the strategic level and the competencies level, on their respective aspects, is investigated and execution misalignments are identified during this stage. This is show as arrow 3a in Figure 2. The strategic choices of development regarding scope, governance and resources have to be matched by the capabilities development execution. Processes, skills and methodologies have to be aligned with the scope of development. Depending on what development has as goals, the tools, knowledge and way of doing things have to match strategic choices. The scope of development dictates execution choices, and the way the function is set up in terms of governance will shape the processes that will take place during the development process. If the development function has a flexible, flat governance structure, then the processes employed during the execution stages have to be equally flexible and adaptable.

- 2.2.2 Strategy Execution Capability of Testing

This involves assessing the overall ability of the testing group to execute its stated strategies. This assessment is done subsequently after the internal coherence of both testing strategy and testing capabilities has been performed independently. This is shown as arrow 3b in Figure 1. Internal coherence amongst the aspects of testing strategy is not a
sufficient condition for this purpose. Neither is internal coherence amongst the aspects of testing capabilities. While testing capabilities can be coherent in relation to processes, skills and architecture, they may not be the appropriate ones for successfully enacting the strategies of the testing group. There can be instances when an internally coherent set of testing capabilities may not be aligned to cover all the types of tests that need to be performed given the specified scope of a testing group’s strategy. For instance, exploratory “gorilla” testing capabilities may be missing from an otherwise coherent set of testing capabilities in an organizational context where the testing group has taken on the responsibility (strategic scope) for testing agile and X-treme development projects.

We know of cases where chief information officers undertook significant reorganization of their IT development shops to ensure that the execution capabilities of their testing groups was enhanced in relation to strategy execution to ensure that they could keep pace with abilities of their development organizations. Our discussions with several outsourcing vendors also reveal that the weak execution capability of organizational testing groups is a key factor in the rapid growth of outsourced testing. While recognizing the not insignificant impact of costs savings in off-shored testing, a desire for effective testing execution capabilities is certainly a factor in such outsourcing decisions. Several vendors of automated testing tools have even started to utilize ROI-based rationales for their services as an efficient and effective way for revamping testing execution capabilities. It is therefore important to ensure that the scope of testing that is strategically negotiated at higher managerial levels, is supported by appropriately aligned testing procedures, competencies and tools. Similarly, as shown in Table 2, the governance models of testing groups have to be appropriately supported by testing procedures, tools and competencies at the execution level. For example, in a situation where testing is to be governed using a decentralized “feudal” model for allocating resources and decision rights pertaining to test planning, there will be a mis-match in alignment if: 1) there are structured and rigid testing processes that are defined by historical legacy, 2) there exist low levels of test planning skills at the execution level for decentralized test planning decisions, and 3) decisions about purchasing automated testing tools and architecture are made centrally at higher levels. Following the same logic, decisions regarding resource allocation for testing at the strategic level have to take into consideration execution issues regarding testing processes, skills and architecture that are pre-existing and desired.

In another organizational environment involving preset release dates for software applications that were generally abided by, coupled with the established sequence of work where testing occurred after development was complete, impeded the ability of the testing group to execute its strategy. Testers had to work under very stressful time constraints to complete their testing in time when the code arrived late. Even though it was aligned in strategic ability and capabilities, this reactive, last-minute “fire-fighting” orientation to testing meant that the testing group often could not execute its strategies in a coherent and efficient manner given that it did not want to be left “holding the ball” when release dates for software were not met.

Stage 2.3: Alignment BETWEEN the Testing & Development Functions:

Preceding steps in our methodology have assessed both internal alignment/coherence within the development and testing functions at both strategy and capability levels. For each of the functions, we have also assessed the ability of the function to execute its stated strategy. At this stage, the focus of the methodology turns to matching the two functions to assess the strategic and capabilities alignment between them. This is show as flow 1 and 2 in Figure 2. As before, this too is done at two levels: i) matching development strategy with testing strategy to assess fit, and ii) matching development capabilities with testing capabilities to assess fit.

- 2.3.1 DT Strategy Alignment:

In this stage we investigate specifically the degree of strategic DT Alignment, which pertains to the degree of conformance between the strategic focus of the development and testing groups on three distinct aspects: - Strategic Scope, Governance and Strategic Competencies.

- 2.3.1.1 DT scope alignment

In terms of strategy, the scope of the two groups needs to be well defined and be complementary. This means that the scope of testing has to be well integrated into the overall scope of development. A well-formalized systems development life cycle that established clear responsibilities and processes for the role that each unit has to play is key to establishing proper scope definitions. For example, a
key aspect could be the understanding that developers/programmers complete unit testing for any module of code that they developed before it was passed on to the testing group. The testing group could then initiate integration and system testing on the basis that each module of code was internally consistent and correct.

Another issue pertaining to strategic scope alignment focuses on the organizational responsibility for developing and testing requirements for new systems. For example, a structured development lifecycle of an organization may task a separate marketing group (outside the IT group) with the responsibility for developing systems requirements in an effort to ensure that business requirements drive the system development efforts. In this case, developers would play an informal role in the requirements development process and would not have much to do with validating the requirements. In this scenario, testers have no role in testing the requirements upfront. This often leads to difficult situations later in the process when code has been written to defined specifications that are based on requirements that have never been validated. A strategic organizational clarification is required, in this case, which would strategically place the responsibility for requirements testing on the testing group and would ensure that appropriate resources are provided for these efforts. If this is not done, the development and testing groups would be misaligned in a strategic sense, with developers arguing that their code is true to the specifications that they had been given yet testers arguing that the new system is not functional from the perspective of meeting business needs.

- 2.3.1.2 DT governance alignment

A second key aspect of Strategic DT alignment pertains to the governance mechanisms used by the two groups. This pertains to the manner in which organizational structure is provisioned, communicated and executed within the two groups. Generally, if one group has a bottom-up orientation while the other has a diametrically opposite top-down orientation, this will lead to misalignment as well. In our experience, development groups are generally bigger in numbers. This means that besides size, the development organization also has more levels in the organizational hierarchy. This variance in levels suggests that there may exist some misalignment in relation to strategic governance with lower level testers and testing managers often having to interact and deal with development professionals and managers who are at higher levels in the overall organizational hierarchy.

Centralization is, in part, a function of the size of the group or organization, and thus, the testing group, with a much smaller number of individuals is generally less centralized than the development group. Also, since testers have comparatively lower status, seniority and, to a certain extent, skill level, they are more likely to have a less centralized decision making structure than the development group. Other determinants of governance include decision rights given to the management of the development and testing functions, reporting and hierarchical structure, as well as policies and procedures regarding stakeholder interaction.

- 2.3.1.3 DT resource alignment

The final aspect of DT Strategy alignment investigated is variances in resource allocation that impact the developer and testing functions. If they differ in terms of strategic resource availability, organizational constraints faced, organizational clout possessed in relation to their stated strategies for building competencies for success, this will negatively impact the level of Strategic DT alignment within the organization. Usually the development group is comparatively better endowed in terms of knowledge, financial resources and human resources. Given its bigger size, longer track-record in the organization it also has more organizational clout and more established practices for capturing and disseminating best-practices. Although this may not necessarily result in misalignments, IT managers have to ascertain that the resources allocated to the testing function are adequate for performing the required activities that testing is assigned. Even though fewer resources are assigned to testing, this does not mean that the testing function should be marginalized and overlooked. A proper resource endowment assigned through budgets, internal project financing, costing and profit centers, etc, is paramount in ensuring proper resource alignment balance between the development and testing functions.

- 2.3.2 DT Capabilities Alignment

This section investigates DT capabilities alignment, which pertains to the execution capabilities of the two functions. It assesses the relative levels of operational capabilities possessed by the two groups that impact their ability to execute the strategies that are assessed in the section above. Three aspects of DT capabilities alignment that support the implementation of the strategic decisions of each group include: 1) the processes involved in
development and testing, 2) the skill sets necessary for development and testing, and 3) the architecture(s) employed by the development and testing groups.

- 2.3.2.1 DT process alignment

The organizational and technical processes involved in development and testing represent the first sub-aspect of DT capabilities alignment. Varying levels of process formalization between development and testing can lead to disconnects in communications and collaboration between the two groups. This is intricately tied to the relative operational methodologies of the two groups which are discussed later. The development group usually employs more formalized processes than the testing department which usually has a more flexible mode of operation. This is partly a result of the different sizes of the two groups as well as the relative maturity of the development function as compared to the testing function. Also, testing theories are generally less developed than structured theories for development and industry best practices in testing are often characterized by ad hoc procedures and the uncoordinated use of brute-force automation tools. This difference in the levels of knowledge and formalization about processes used by development and testing respectively is also reflected in variance in the level of process definitions. The development group usually has better defined processes specifications. The testing group, on the other hand, may have a less clear idea of the start and end points of their processes as well as lack precise mechanisms for completing each process step.

- 2.3.2.2 DT skills alignment

The second aspect of DT capabilities alignment pertains to comparative knowledge and skills available to the development and testing groups. Misalignments here are mainly due to the fact that it usually is much easier to hire skilled developers in the software development marketplace than to hire experienced testers. Formal training programs for development both in academia and industry are readily available but this is not the case in testing. Also, the state of knowledge in development is more advanced than in testing, although there are well established tools (Mercury testing tools) and methodologies for testing.

- 2.3.2.3 DT architecture alignment

The third and final aspect of DT capabilities alignment pertains to the architecture used for development and testing. Here we investigate the tools and platforms used for development and testing, the level of standardization coordination and the technological constraints that are imposed on the development and testing groups. Certain methodologies used in development are more amenable to integrating testing throughout the development process, especially for bringing testing to earlier stages of development. In order to accomplish their goals, developers and testers can use an array of different tools and methodologies. When discussing the architecture of development or testing, the level of standardization of tools and methodologies may be different. The development function generally has more lenience when choosing a certain tool or method to develop its products. This is because of the higher maturity that development usually has, not only in the organization but also in the IT landscape in general. The testing group has less knowledge and maturity, therefore is more standardized in respect to architecture choices. This is also due to the lack of multiple tools or methods for doing testing. This lack of alternatives in testing tools and methods imposes additional technological constraints on the testing group, constraints with which the development group does not have to contend.

3.3 Phase three: Action Plan for DT Realignment

In phase three of our Developer-Tester Alignment Methodology, an action plan for realigning the two functions is created and implemented. The misalignments identified in phase two, are prioritized and solutions for realigning the functions are designed for each individual source of misalignment. Business process redesign, hierarchical reorganization, resource reallocation, human resource development and recruiting, new tool acquisition, are just a few of the possible activities that take place during this phase. Once a detailed inventory of solutions for misalignments has been devised, these are all prioritized on importance and feasibility and a detailed realignment strategy is developed. This strategy is created with input from all stakeholders of the realignment process, and, once the strategy is approved by all stakeholders the resources necessary for the realignment process are procured and allocated towards the enactment of the strategy.
The agreed upon strategy is then implemented by addressing all sources of misalignment identified in phase two. A subset of the tools and techniques presented in the next section can be used to design and implement the developed realignment strategy.

4. An inventory of tools and techniques for DT alignment

This section presents a non-exhaustive list of techniques that can be used to assess and redress the level of alignment between and within development and testing. It is a sample of potential tools and techniques, but any generic tool or technique should be customized to particulars of the specific DT Alignment situation that an organization faces.

**Budget Matching**
During the DT Alignment process, a budget for financial and human resources for the two functions can be prepared using standard budgeting techniques and by investigating the resource requirements of the development and testing functions.

**Cognitive mapping**
This technique graphically represents the causal relationships perceived by the decision makers to exist among the elements of a given environment. This can be used to investigate the relationship and flows which exist between the development and testing functions and also within the two functions, regarding authority, tasks, code and feedback.

**Data Flow Diagramming**
This technique depicts graphically the data flows among external entities, internal processing steps and data storage elements. This can help track the flow of data between testing and development and draw attention to “hand-offs” between them.

**Project Scheduling Techniques**
This is a generic category of techniques that includes techniques such as PERT, CPM and Gantt Charting used for scheduling, controlling and managing projects. These can be used at all levels of the DT Alignment process to ensure the smooth flow of the Alignment process.

**Methodology analysis**
This technique investigates the ISD methodologies and tools as well as the testing methodologies and tools and assest the level of capability alignment between the development and testing functions as far as tools and techniques are concerned.

**Organization Hierarchy diagramming**
A diagram of the organizational structure both for the entire organization and for the two functions – development and testing.

**Pareto diagramming**
This diagramming technique is based on the Pareto principle that a few causes often account for most of the effect. The diagram represents problem causes which are ranked in descending order of importance and indicates which causes should be targeted. This allows analysts to target the more important causes of DT Misalignment and to allocate resources to their resolution based on importance.

**Process flowcharting**
This graphical technique models the flow of activities in a business process by using typical flow charting symbols and methods. This can be used to investigate the processes used by development and testing and identify any misalignments arising from the process flows.

**Reporting Structure analysis**
An analysis of the reporting structure for the development and testing functions, which looks at how decisions are made and disseminated and how the results of the decisions are reported and analyzed. This is used to analyze the scope and governance of development and testing and to identify potential misalignments.

**Skill inventory analysis**
This technique is used to assess the level of alignment between and within functions as far as the skills and competencies of the functions are concerned. A skills inventory keeps track of an employee’s job qualification, education and experience, and provides ways to assess future avenues for developing needed skills.

**Tools and techniques inventory analysis**
This focuses on an inventory of available tools and techniques and how well they support the goals of testing and development. A key variable to assess is if the computer assisted software engineering (CASE) tools for development are aligned with automated testing tools that are used. Most CASE tools take an efficiency perspective and this often contradicts the effectiveness mandate of software testing.
Testing capabilities audit
One useful technique is to undertake a testing capabilities audit. This is a joint review, involving both developers and testers, focusing on the appropriateness and adequacy of the testing capabilities possessed by the testing group. Specific cases of software failures are used as starting points for this audit and fishbone-type analysis tools are used to trace each failure to specific weaknesses in testing capabilities.

Log Keeping
Another method is to keep detailed logs of all “hand-offs” between development and testing and undertaking regular reviews of these interactions from a responsibility/accountability perspective. The lack of such documentation that facilitates objective tracking review is often a sign that developer-tester misalignment may be an issue.

Employee attitude monitoring
Regular monitoring of the relative work-related morale of developers and testers is also critical. Periodic comparative reviews of employee turnover statistics in the two areas as well as comparisons of reward and promotion structures are also effective mechanisms for revealing if developer-tester alignment is an issue in the organization.

In addition to the tools and techniques presented above, it is useful to investigate the multiple types of fit that have been identified in the literature. This fit between functions has six alternative conceptualizations as adapted from Venkatraman [38]:

1. DT Alignment as Moderation: this fit between development and testing is considered to be composite variable that influences the direction and strength of the relationship between each function’s strategy and their outcomes. Here we consider DTA to influence the result of the development and testing activities, such as, a high DTA positively influences the results of the development and testing activities

2. DT Alignment as Mediation: in this case, the fit is a necessary intervening variable between the strategy and success of effort of the development and testing functions. In other words, without having DTA the development and testing activities will not be successful.

3. DT Alignment as Matching: here we investigate the correspondence or equivalence between development and testing strategies and capabilities. For example DTA is envisioned as a correspondence of capabilities, tools and resources in both the development and testing functions and leads to successful implementation of business systems.

4. DT Alignment as Gestalts: this fit focuses on the degree of internal coherence between development and testing strategies and capabilities. DT alignment is measured through comparison to other groups of companies with related development and testing strategies/capabilities. A benchmarking strategy in regard to DT alignment can be employed here.

5. DT Alignment as Profile Deviation: an ideal profile of well aligned development and testing is identified and local deviations from this profile are investigated as possible misalignments. For example, the top 10% of best performing companies can be investigated in relation to their DTA strategies/capabilities and the implementation process of a sound alignment arrangement. Next, differences between this ideal profile and the current state of affairs in a particular organization can be brought to focus.
Table 3: Mapping of Tools and Techniques to the Individual Phases of DT Alignment

<table>
<thead>
<tr>
<th>Phase</th>
<th>Stage</th>
<th>Tools and Techniques</th>
</tr>
</thead>
</table>
| Phase 1: Misalignment identification | Symptoms analysis | Cognitive mapping  
Employee attitude monitoring  
Log Keeping  
Pareto diagramming |
| Phase 2: DTA Assessment | Stage 2.1: coherence WITHIN components | 2.1.1 assessing internal alignment/coherence of development strategy | Cognitive mapping  
Organization Hierarchy diagramming  
Reporting Structure analysis |
| | | 2.1.2 assessing internal alignment/coherence of development capabilities | Development capabilities audit  
Skill inventory analysis  
Process flowcharting  
Data Flow Diagramming  
Methodology analysis  
Tools and techniques inventory analysis  
Testing capabilities audit |
| | | 2.1.3 assessing internal alignment/coherence of testing strategy | Cognitive mapping  
Organization Hierarchy diagramming  
Reporting Structure analysis |
| | | 2.1.4 assessing internal alignment/coherence of testing capabilities | Testing capabilities audit  
Skill inventory analysis  
Process flowcharting  
Data Flow Diagramming  
Methodology analysis  
Tools and techniques inventory analysis  
Testing capabilities audit |
| | Stage 2.2: Strategy Execution Alignment Within Function | 2.2.1 Development Strategy Execution Capability | Cognitive mapping  
Data Flow Diagramming  
Project Scheduling  
Techniques  
Methodology analysis  
Organization Hierarchy diagramming  
Process flowcharting  
Reporting Structure analysis  
Skill inventory analysis  
Tools and techniques inventory analysis  
Testing capabilities audit |
| | 2.2.2 Testing Strategy Execution Capability | Cognitive mapping  
Data Flow Diagramming  
Project Scheduling  
Techniques  
Methodology analysis  
Organization Hierarchy diagramming  
Process flowcharting  
Reporting Structure analysis  
Skill inventory analysis  
Tools and techniques inventory analysis  
Testing capabilities audit |
| | Stage 2.3: Alignment BETWEEN the Testing & Development Functions | 2.3.1 DT Strategy Alignment: | Cognitive mapping  
Organization Hierarchy diagramming  
Reporting Structure analysis |
| | | 2.3.2 DT Capabilities Alignment | Testing capabilities audit  
Skill inventory analysis  
Process flowcharting  
Data Flow Diagramming  
Methodology analysis  
Project Scheduling  
Techniques |
| Phase 3: DTA Action Plan | | Project Scheduling  
Techniques  
Pareto diagramming |

6. DT Alignment as Covariation: this fit is seen as a covariation of attributes that characterize the development and testing functions. Here we look at the attributes of development and testing separately and investigate how they covary or diverge. Misalignments would occur when the attributes of testing and development vary in an unexpected way.

5. Conclusion

In this paper we develop a methodology for measuring and achieving Alignment between the development and testing functions of an enterprise. It focuses on internal and external components of the two functions and how they interact, support and fit each other. By investigating the individual components of alignment – alignment within components (strategy and execution/capabilities) of development and testing; alignment between the strategy and execution component of development and testing; and finally alignment between the two functions themselves on their strategy and execution capabilities – this methodology presents a comprehensive view of the enterprises IT department based on the division between the development and testing functions. In addition to the methodology itself, we also present a comprehensive set of tools that can be used during specific phases of the DT Realignment process and which should help practitioners in choosing the appropriate tool and technique for their specific misalignment issue.

6. References


Abstract

Numerous software development and testing methodologies, tools, and techniques have emerged over the last few decades promising to enhance software quality. While it can be argued that there has been some improvement it is apparent that many of the techniques and tools are isolated to a specific lifecycle phase or functional area. Tool integration advocates promise to further increase software quality throughout the organization though literature remains primarily focused on technical solutions rather than on socio-economic factors associated with tool integration. This study draws on literature in information systems, organization science, and software engineering to develop a conceptual model of factors that influence an organizations’ need for enterprise-level integration of information systems development tools.

1. Introduction

Software permeates virtually all aspects of our lives, from life-saving medical devices to online banking to social networking. Users expect this software to work. However, the reality is that often it doesn’t. Accounts of failed software and its economic impact are commonplace. Software quality is critical to the reliable functioning of business today. Catastrophes such as the air traffic control system in the LA Airport, Denver Airport’s baggage handling system, online voting systems, and other systems cultivate an urgent reminder that more needs to be done to ensure the creation of higher quality systems. Of the $59.5 Billion that software bugs cost the US economy annually, improvements in testing could reduce the cost by about 1/3 [1]. Certainly other improvements to the software development process and integration of software and testing toolsets would provide additional cost savings.

Recent decades have seen a dramatic shift in the systems development and testing landscape. Some of these changes have been the result of technological innovations or industry competitiveness, however, a substantial number of these changes have been in response to the large percentage of failed software projects. A wide variety of software development and testing methodologies, techniques, and tools have emerged over the last few decades with the promise of enhanced software quality. However, many of these artifacts have failed to produce the promised benefits.

Despite considerable investments over the past thirty years, there remains only two major ways to improve software quality. First, you can improve and control the process by which software is written. And, second, you can better specify and measure the outcomes of the software development and testing process.

Software engineering, through such efforts as those spearheaded by the Software Engineering Institute at Carnegie-Mellon University, has begun to layout the “best” process for building software. And, consultancies and vendors have responded to these prescribed methods by offering a litany of software tools that embed the principles of these accepted
methods and their proven techniques. CASE tools and project management package vendors now offer reasonably mature products for documenting and monitoring software projects at a high level. However, these integrated CASE packages do not always provide the granularity needed to control the conduct of the detailed steps of the software development process. Additionally, all too often these tools primarily focus on the best techniques within a single phase of a particular development methodology and give little consideration to the flow of information throughout the entire lifecycle. While many vendors now offer multiple products for different phases of the lifecycle, often these product offerings have been acquired and still lack robust integration throughout the entire development lifecycle.

Software development project managers are often dependent on outcome evaluations of the software development process as the only means to measure software quality. Just as software project managers have quality control problems even when they use modern CASE tools, software developers experience additional problems. These often include not having access or knowledge from other phases of the development lifecycle and thus are without means to most effectively and efficiently make contributions. This situation can result in delays in the development process and/or poor quality code.

Nowhere are the problems of software quality more relevant than in the “software testing” portion of the system development process. It is here where time pressures are the greatest and where testing process methods are less standardized and vary by tester that software development project managers are pushed even more to rely on outcome measures as the principle quality metric. This situation has slowed the codification of best testing practices into many integrated vendor CASE tools packages and has promoted the wide spread use of specialty tools from specialty small software vendors and/or the use home grown techniques and tools. More recently vendors such as Hewlett Packard, Borland and IBM have made strides to offer more integrated tools sets that accommodate differences in development processes.

Within many organizations there is often a lack of communication between functional areas of the organization, teams, business units, or geographical locations. This lack of communication is often referred to as a “silo” or an “island.” Software organizations often also suffer from silos as it regards the awareness, interest, evaluation, trial and adoption of software development and related application lifecycle management methodologies, techniques and tools. As alluded to previously, software development tools have evolved within functional boundaries (e.g. requirements analysis, configuration management, testing, etc.) and often the selection of these tools is limited to the ability of the tool to perform tasks within the particular functional area without consideration for the benefits of integration with tools that support other lifecycle phases.

A fair amount of research has been conducted over the years on the benefits of standardization, integration and data centralization, however, application of these concepts to the software engineering process has received relatively minimal attention. In just the last few years terms such as application lifecycle management (ALM) are becoming popular buzzwords among consultancies and tool vendors and many benefits are being touted for those who implement ALM solutions.

This study examines research related to application lifecycle tool integration, standardization, and centralization and proposes a conceptual model for organizations’ need for enterprise-level integration of information systems development tools. Specifically we identify factors that influence the need for such standardization and integration of software development tools.

2. Related Research

2.1. Software Development Processes

The process of developing software is a “complex, knowledge intensive activity” [2]. Over the years a number of standard software processes, such as process reference models and guidelines, have been developed to describe systematic approaches to developing software at varying levels of detail [3]. Developmental or lifecycle models, such as waterfall or spiral development organize production activities into distinct phases (e.g. requirements, design, implementation, testing, and deployment) and also specify the transition criteria between the phases. While these general lifecycle phases provide general guidelines, other reference models (e.g., ISO.IEC/12207, IEEE/EIA 12207) provide greater detail as to the roles, responsibilities, tasks, and activities during each lifecycle phase [3].

The creation of software products is a design activity. The result of the design process is an artifact. Not only is the creation of the software itself an artifact of the software development process but there are
numerous artifacts that are created as a result of the design process that occurs during each phase in the development lifecycle. Artifacts that are output from one phase often serve as input for the next phase. The following table captures some of the artifacts that are created as part of the software development process [4]:

Table 1 Artifacts associated with lifecycle phases

<table>
<thead>
<tr>
<th>General Lifecycle Phase</th>
<th>Artifact Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feasibility Analysis</td>
<td>Feasibility report</td>
</tr>
<tr>
<td>Project Planning</td>
<td>Project plan</td>
</tr>
<tr>
<td>Requirements</td>
<td>Requirements Specification</td>
</tr>
<tr>
<td>Analysis</td>
<td>System Design Document</td>
</tr>
<tr>
<td>System Design</td>
<td>Detailed Design Document</td>
</tr>
<tr>
<td>Detailed Design</td>
<td>Detailed Design Programs</td>
</tr>
<tr>
<td>Development</td>
<td>Test Plan</td>
</tr>
<tr>
<td>Testing and Integration</td>
<td>Test Report</td>
</tr>
<tr>
<td>Installation</td>
<td>Manuals</td>
</tr>
<tr>
<td></td>
<td>Installation Report</td>
</tr>
</tbody>
</table>

The IEEE/EIA 12207 standard splits software related processes into primary, supporting, and organizational processes [5]. Primary processes provide guidance on the acquisition, supply, development, operation, and maintenance of software products. Supporting processes provide additional information that is necessary to ensure software quality through documentation, configuration management, quality assurance, verification, validation, joint review, audit, and problem resolution. Organizational processes focus on management of the primary process (e.g. project planning, tracking), infrastructure, process improvement, and training.

The coordination of development life-cycle activities including requirements, modeling, development, build, and testing is referred to as application lifecycle management (ALM) [6]. This coordination occurs through three primary mechanisms: 1) enforcement of the development processes in any of the lifecycle phases, 2) management artifacts and relationships between development artifacts produced or consumed by any of the lifecycle phases, and 3) metrics on the progress of the development effort as a whole [6].

2.3. Tools supporting software development

Numerous tools have emerged over the years to support activities throughout the development lifecycle. For each artifact listed in Table 1, there are numerous tools that support the creation of that artifact. Perhaps most familiar are the tools that support the development of the software product itself. For example, Integrated Development Environments (IDE) that provide source code editors, a compiler/interpreter, build tools, and a debugger provide are software tools that support activities in the development phase of the lifecycle. A few of the myriad IDEs available include Eclipse, ActiveState Komodo, and Microsoft Visual Studio. Tools are plentiful for other lifecycle phases too.

Historically, tools have been created for particular activities within a single lifecycle phase; that is, tools tend to support activities and the resulting artifacts within a phase rather than the use of that artifact across phases. While vendors are now touting integrated suites of tools, the reality is that vendor’s offerings are a smattering of products brought into the same “integrated suite” via acquisition and some work better together than others.

Many software development organizations find themselves in a very similar situation; that is, they have various primary and supporting tools that are used to create and manage the artifacts produced in different lifecycle phases; however, often there is limited or crude integration between tools. The adoption of individual products in this heterogeneous set of tools may have been an active, cognizant decision, or it may have been the result of external, uncontrollable (or less controllable) factors (e.g. initiative of higher-level management; a licensing or cost issue).

The decision to adopt a software development tool for a particular lifecycle phase is a complicated process due to possible integration with the existing tools, scalability, usability, customization, support, etc. [7].

The integration or linking of tools that support different aspects of the development process was recognized as a key issue in CASE research nearly two decades ago [8], however we are still grappling with this issue.
What exactly is tool integration? “Tool integration is about the extent to which tools agree. The subject of these agreements may include data format, user-interface conventions, use of common functions, or other aspects of tools construction” [9].

Brown and Penedo [10] posited that the “greatest challenge is to integrate [the successes in individual phases] to produce an effective and integrated automated environment that supports the complete software development life.” They cite that expected benefits of automated support for the software development process include processes that are “more predictable, less expensive, easier to manage, or produce higher quality products.”

A recent, comprehensive review of tool integration literature organized articles from the last few decades into three main categories: descriptive, formulative, and evaluative [11]. The review draws on an excellent annotated bibliography of articles dealing with tool integration within software engineering environments [12].

Interestingly, after identifying the heavy technological focus of the existing academic research on tool integration, Wick and Dewar [11] contend that the problem needs to be re-examined from a socio-economic perspective. More specifically they propose a possible relationship between the net value of tool integration and the level of integration.

To better understand the problem of tool integration in software development process, we conduct a review of past studies focusing on the managerial aspects of information systems development in information systems, organization science, and software engineering disciplines.

### 2.4. Managerial Aspect of Information Systems Development

Cost overruns, project delays, unmet user needs and systems that are never fully utilized are commonplace to most software development practitioners [13]. This chronic situation has been addressed by not only methodology, tools, and techniques, but also using multi-faceted managerial approaches in an attempt to minimize the all-to-common failures mentioned earlier. Information systems development (ISD) literature has adopted several theoretical perspectives on improving ISD performance.

**Risk management**

The practice of risk management involves two main actions: assessing the risk and then controlling for it [14].

Risk assessment can be decomposed into a sequence of the following activities [14]:

1) risk identification which often consists of “producing lists of project-specific risk items that are likely to compromise a project’s success,”
2) risk analysis which assesses the probability and magnitude of loss for each risk items and also takes into account the interaction affects of the individual risk items, and
3) risk prioritization which “produces a ranked ordering of the risk items identified and analyzed”.

Risk control can be further decomposed into the following activities [14]:

1) risk management planning which creates a plan to deal with the individual risk items or the compounded effects of multiple items,
2) risk resolution which refers to elimination or the risk items typically achieved by the carrying out the activities of the risk-management plan, and
3) risk monitoring which “involves tracking the project’s progress toward resolving its risk items and taking corrective action where appropriate.”

ISD literature identifies numerous risk items/factors such as misunderstanding the requirements, not managing end-user expectations, changing requirements, lack of top management support, insufficient/inappropriate staffing and many more [14-17]. Schmidt et al. [16] further cluster numerous risk factors into 14 different groups and identify the source and nature of risk items in that group.

**Coordination Theory**

A major area of investigation in coordination theory is developing a deeper understanding of “the effects of information technology on human organizations and markets” [18]. In context of ISD, the benefits of implementing new systems development and application lifecycle management tools requires more than just installation of new technology. It also requires organizational change.

Orlikowski [19] suggests that “in order to account for the experiences and outcomes associated with CASE tools, researchers should consider the social context of systems development, the intentions and actions of
key players, and the implementation process followed by the organization.” Guinan et al. [20] found that “team skill, managerial involvement, and a little variance in team experience enable more effective team processes than do software development tools and boundary-spanning behavior.”

Rai and Patnayakuni [21] found that CASE training availability, having a CASE champion, and the frequency of job/role rotation positively affect CASE adoption behavior. Interestingly, top management support doesn’t affect CASE adoption, however, it is suggested that perhaps management support may be “more critical for post adoption stages” [21].

Knowledge Integration and Sharing in ISD

The knowledge-based theory of the firm proposes that knowledge is the most strategically important resource of a firm and that performance of firms depends on the creation, management, and use of knowledge [22, 23].

Organizations often become fragmented into groups and often lack communication and interaction with other groups in the organization. These fractured, fragmented groups are considered silos. There are four primary types of silos in IT organizations: geographical, organizational, functional, and technological [6].

Geographical silos are based on the location of the group (e.g., Memphis, Bangalore). Organizational silos are formed along departmental, business unit, or team boundaries. Functional silos form along functional area boundaries. In software development, silos might form around phases of the application lifecycle (e.g., analysis, development, testing). Technological silos are based on specific technological artifacts that are used within an organization (e.g., J2EE, Oracle).

Building on the research that examines knowledge integration and application, Patnayakuni et al. [24] found a positive influence of organizational integrative practices such as participative decision making and job rotation on knowledge integration in the ISD process. Furthermore, they found that knowledge integration in the ISD process positively influenced ISD performance.

Several scholars have also investigated coordination and sharing of specialized knowledge in the ISD process. Because software development is knowledge work and expertise is an important resource, Faraj and Sproull [25] examined the role of expertise coordination. They found that expertise coordination processes—“knowing expertise location, recognizing the need for expertise, and bringing expertise to bear”—had a strong impact on team performance.

Recognizing the importance of knowledge integration in large-scale software projects that are geographically distributed, Espinosa et al. [26] focused on the coordination needs of such ISD teams and the influence of geographic dispersion on the coordination needs. They found that these teams have technical, temporal, and process coordination needs and that geographic dispersion has a negative impact on coordination, an impact that is mitigated by shared knowledge of team awareness and task awareness. Team awareness refers to the knowledge about the others’ activities or events that occur in the team environment at any point in time. Examples of team awareness include knowledge of the progress of the development of a software component or an upcoming deadline. Task awareness refers to a team member’s knowledge of recent task activities such as who did what and the progress of others’ tasks.

Control theory

Past studies looking at control of ISD projects have examined two aspects: the specific control modes that are used (e.g., [27-29]), and the factors that influence the choice of control modes [27, 28].

Control theories suggest behavior control, outcome control, clan control, and self control as four control modes [30, 31]. Behavior control involves defining steps and procedures for performing tasks and evaluating controllees’ performance based on their adherence to the procedures. Outcome control involves specifying desired outputs of the task and evaluating controllees’ performance based on the extent to which they have met the specified target outcome. Unlike formal control such as behavior control and outcome control, informal control focuses on the role played by individual or group norms. Self-control relies on a controllee setting his own goals and self-evaluating his goal achievement. Clan control occurs when a group of controllees sharing common goals develop their internal values and norms and regulate their behavior based on shared values and norms.

According to the literature on control of ISD activities, the appropriate level of each type of control depends on a variety of factors such as task programmability and outcome measurability, and information availability. In summary, control
literature suggests that high levels of outcome measurability, which represents the ability to measure results, are positively associated with the exercise of outcome control [27, 30]. Furthermore, when controllers have knowledge about the ISD process and controller behavior has a high degree of observability, the use of behavior control increases [27, 31].

3. Conceptual or Theoretical Framework

So what benefits can integration of software development tools bring? A recent case study on tool integration and ALM can shed some light [32]. BNSF Railway, one of the largest railroad networks in North America, chose to buy an application lifecycle management system due to the time pressure to modernize its systems and better prepare for internal and external regulation compliance audits. The result is an enterprise-level application lifecycle management solution that integrates requirements management, enterprise software configuration management, process and workflow management, software build management, etc. After the implementation of the ALM project, BNSF is able to completely trace and document the requirements and to securely manage its code in one centralized repository. Furthermore, all application teams follow a standardized development process and geographically dispersed teams can collaborate more effectively. Finally, it is able to build report and evaluate metrics obtained throughout the entire application lifecycle.

Theoretically, what benefits can SDLC tool integration bring? Standardization of processes and tools, centralization of artifacts, and enhanced…

Standardization refers to standardization of the process followed to create software and accompanying artifacts, standardization of the tools used during creation, and also the standardization of the representation of the artifact. The degree of standardization refers to the extent to which management specifies a uniform set of rules and procedures to guide the process [2], standardize the tools, or standardize the representation of the artifacts themselves.

Standardization can facilitate codification of knowledge acquired from the development process and obtaining uniform performance metrics to assess outcomes throughout the development lifecycle. Control theory further suggests that standardization of the software development primary and supporting processes allows behavioral control and the collection and standardization of performance metrics.

![Figure 1 Illustration of tool integration in software engineering for a sample company](image-url)
(especially based on the same tool) allows increased outcome measurability and benchmarking across projects.

Centralization of software development artifacts produced and augmented in different lifecycle phases is often, though not necessarily, a component of an integrated tool suite. Coordination and knowledge integration literature suggest that a centralized repository of artifacts contributes to increased team and task awareness which can mitigate the impact of geographic dispersion on team coordination.

Standard communication interfaces between tools that focus primarily on a single or limited number of lifecycle phases leads to increased interchange of artifacts produced in different phases and allows for augmentation of artifacts throughout the ISD process. Perhaps the most tangible, salient benefit is that of improved traceability of requirements and software components across phases. Based on coordination theory we can expect increased information sharing and its associated benefits.

Standardization of processes and tools, centralization of artifacts, and increased interchange of artifacts between phases mitigates many of the risks in software development. While risk assessment and risk control activities are still a major concern, a standardized and integrated approach to application lifecycle integration provides much of the information needed in order to effectively identify, analyze, prioritize, plan for, mitigate, and monitor risks. Risk management literature suggests that organization can better manage the risks in projects that are complex and have volatile user requirements.

Figure 1 illustrates the concepts of tool standardization and integration in context of a software engineering organization. Groups or projects, a type of organizational silo, are illustrated as rows in the diagram. Application lifecycle phases, a type of functional silo, are illustrated as columns in the figure. Notice that Group/Project 2 is the only group that has integration of tools and associated artifacts across the different application lifecycle phases and tools. In this example, tool standardization is illustrated by the fact that the sample company has a single modeling tool used across all groups/projects.

Based on the control literature, risk-management literature, and other related literature we developed a list of factors that influence the need for formal process control, formal outcome control and information transparency which can be achieved through appropriate integration of software development lifecycle management tools.

**Organization Characteristics**

**Geographic Distribution of ISD Teams**

Geographically distributed teams are groups of people sharing a common purpose and performing interdependent tasks across locations and time with few opportunities for face-to-face interactions [33, 34]. As more and more organizations outsource some or all of their application development activities to domestic and foreign vendors, software development teams are becoming increasingly geographically dispersed. However, both researchers and practitioners have identified that these global teams are even more difficult to manage than traditional, co-located teams, due to coordination problems, lack of trust and shared understanding among other factors [33, 35, 36].

In global ISD teams, mutual knowledge, that is knowledge that the communicating parties have in common and know they share [37], helps establish common ground, improve communication and coordination, and reduce misunderstanding and interpersonal conflicts among individuals. Failure of information exchange between global teams will result in dispersed teams having different pieces of information and potential withholding of local information.

Consequently, organizations whose ISD teams are globally located may have a greater need for information sharing among these teams. Application lifecycle tool integration has the potential to allow global teams to access real-time information and to obtain a more accurate picture of off-site activities. In the meanwhile, integration also facilitates the monitor and control of remote vendors’ activities and progress by upper management.

H1: The greater the geographic distribution of ISD teams, the higher is the likelihood that managers will choose to integrate their software development tools.

**Software Development Process Maturity**

Consistently throughout the software project risk management literature the inadequate use of project management practices shows up as a significant risk item. The lack of an effective development process or methodology leads to software quality problems including issues in appropriate documentation, software and testing, poor estimating, etc [16]. In addition, Tiwana and Keil [17] labeled the use of an inappropriate methodology for a given project the highest relative risk of the six risk factors identified.
H2a: The less mature the software development processes or methodology employed by an organization, the higher the likelihood that managers will choose to integrate their software development tools.

We argue that most any standardized development approach or paradigm, whatever it may be, is better than the lack of a coordinated approach. Project management artifacts in this category include signoffs, project tracking methodology, awareness of overall project status etc.

The standardization of the flow of activities that are part of the application development process as implemented in the adopted tools and integrated with tools throughout the application lifecycle provides a specific development methodology all projects within the organization. As a result, software development and application lifecycle management tool integration provides many of the tools for appropriately tracking project plans, tracing requirements through the application lifecycle, etc. Additionally, this integration allows for greater overall project status awareness.

H2b: The less mature the project management methodology employed by an organization, the higher the likelihood that managers will choose to integrate their software development tools.

Another aspect of overall project management that is identified as a risk in the literature is the “bad estimation” of work due to the lack of effective tools, techniques, or based on inappropriate data [16]. Software development and application lifecycle management tool integration provides for standardization of the gathering and reporting of metrics across projects within an organization.

H2c: The lower the accuracy of past project estimations, the higher the likelihood that managers will choose to integrate their software development tools.

Turnover of ISD Professionals

Because software development is knowledge-intensive work, the most important resource is expertise. In the ISD process a variety set of skills are needed and employees are usually specialized in a particular knowledge domain. The lack of required knowledge/skills remains a significant risk to software projects success [16]. Recognizing the fact that turnover of IT professionals remains one of the chronic challenges facing organizations, many organizations have developed knowledge management systems to capture and codify the tacit knowledge of their employees but failed to fully utilize these systems due to a number of factors. Moreover, the ISD process not only requires applying specialized knowledge for specific tasks but also transferring, translating, and transforming specialized knowledge across individuals and development stages [24]. A high IT turnover rate makes knowledge integration even more challenging.

Integrating software development tools helps codify the specialized knowledge possessed by individuals by storing all technical artifacts and intermediate steps in a central repository, which has the potential to minimize the impact of employee turnover and maintain smooth transition when a new employee joins.

H3: The higher the IT professional turnover rate that managers perceive, the higher is the likelihood that managers will choose to integrate their software development tools.

Project Characteristics

Task Interdependence

Andres and Zmud [38] define task interdependence as “the extent to which a task requires organizational units to engage in workflow exchanges of product, information, skills, or resources, and to where actions taken in one unit affect the actions and work outcomes of other units”. Software development projects vary in their levels of task interdependence. Higher levels of task interdependence require more exchange of information related to requirement clarification and project progress [39]. Therefore, when the development activities in an organization are highly interdependent more inter-project communications are needed. Integrating and standardizing SDLC through tool integration can facilitate such communications and information exchanges and improve software development performance. Hence, we hypothesize that:

H4: The higher the degree of task interdependence that managers perceive, the higher is the likelihood that they will choose to integrate their software development tools.

Requirements Volatility

The volatility of requirements, though somewhat expected, can have a significant impact on success of a project. Perhaps most important is not whether or not there are volatile requirements rather how mature is the requirements management and traceability process in order to ensure that the solution that is
developed meets the needs of the stakeholders involved.

An integrated software development and application lifecycle tool set allows for project similarity for the process of developing the system. For example, estimation model generation, project plan creation, requirements traceability, test integration and reporting are all consistent from one project to another and the artifacts created in one lifecycle phase can flow as inputs to other phases.

H5: The greater the perceived volatility of requirements for projects within an organization, the greater the likelihood that managers will choose to integrate their software development tools.

Figure 2 illustrates the hypothesized relationship of these organization and project factors that influence an organization's need for enterprise-level integration of ISD tools and.

![Figure 2 Factors that impact an organization’s need for enterprise-level integration of ISD tools and hypothesized relationship](image)

5. References


Representing Test Cases for Semantic Web-based Software Testing Systems

Abstract

Recent advances in the World Wide Web technologies have brought about the emergence more powerful tools and applications that can support collaborative activities. Now with the advent of the Semantic Web as the future Web, it is timely to explore the potential use of emerging Semantic Web technologies in developing knowledge-based software testing systems (KbSTSs) that would also be collaborative in nature. This paper uses test case management as a basis for studying the problem of developing KbSTSs over the Semantic Web framework and explores the representational issues pertaining to test cases that are foundational to the development of what we refer to as Semantic Web-based software testing systems (SWbSTSs).

1. Introduction

Software testing has always been regarded as an important activity within any software engineering process. However, despite making tremendous research progress over the past few decades, much of the research results produced have not been fully absorbed by the industry. In line with this argument, a recent review paper [1] claims that software testing is still being considered as ad hoc, expensive, and unpredictably effective. This, as claimed, still poses great challenges in software testing research. Generally speaking, usability of any software testing system can be improved through increased automation. However, a fully automated system that can comprehensively cover all aspects of software testing is still a distant dream. Nevertheless, as now increasingly being explored, knowledge-based software testing systems (KbSTSs) are expected to add more values to testing environments.

Furthermore, with the advent of the Semantic Web concept [2], opportunities are now wide open for development of what we refer to as Semantic Web-based software testing systems (SWbSTSs), through the use of the emerging Semantic Web technologies. Since Semantic Web technologies concern knowledge development and management, SWbSTSs can then be considered as a sub-class of KbSTSs. The aim of this paper is to highlight the potential use of Semantic Web technologies in developing KbSTSs through using test case management problem as a focal point. Specifically, this paper will discuss issues and techniques involved in representing test cases for SWbSTSs. The paper is organized as follows. Section 2 gives the background that discusses the research aim, the underlying assumptions, and the key research questions. Detailed discussion on structuring and representing test cases is given in Section 3 and Section 4 respectively. Finally, Section 5 presents the conclusions and future works in this direction.

2. Background and basic assumptions

The main aim of our research is to produce a theoretical framework to support the designs for SWbSTSs. This research initiative is motivated by first, the potential use of the Web as the platform to support collaborative software testing and secondly, the advent of the Semantic Web as the envisioned future Web. Towards that aim, we focus on test case management problem as the basis, based on the following assumptions:

1. The use of test cases is central to any testing process.
2. Capturing logical relationships between test cases and other relevant testware and software artifacts is key in supporting reasoning on test cases.
3. A good test case management subsystem will be a useful component to any software testing system.

Based on the above arguments, two general research questions become our immediate interests:

1. How best can we utilize existing Semantic Web technologies in developing test case management systems?
2. How to ensure that these test case management systems can be incorporated or be made to inter-operate with the bigger Semantic Web-based software testing systems (SWbSTSs)?
The vision of Semantic Web was first articulated in 2000 by Tim Berners-Lee, who is the inventor of the current World Wide Web during his XML 2000 address (http://www.w3.org/2000/talks/1206-xml2ktbl/slide1-0.html). The Semantic Web idea is further elaborated in [2] where the Semantic Web is envisioned to be the future Web that would consist of semantic extensions to support data connectivity needed for enhancing human-computer as well as computer-computer co-operations. It will be built upon current and future de facto standards for describing and reasoning with data on the Web.

In an interview [3], Tim Berners-Lee mentioned that after several years of effort, the Semantic Web project has managed to provide a good solid layer of Resource Description Framework (RDF)[4] and Web Ontology Language (OWL) [5] which allows systems to be described, and data to be exchanged.

Referring to the first research question above, we contend that effective utilization of Semantic Web technologies must revolve around judicious use of both RDF and OWL. In addition, since Semantic Web languages are and would in general be XML-based, judicious use of XML technologies will also be important here.

As for the second research question, interoperability is ensured as long as the development of these systems utilize the W3C recommended languages and standards and in phase with the evolution of Semantic Web technologies. This is so since the basic premise Semantic Web architecture is to ensure interoperability through standardization. Hence, the immediate goal under this research initiative then is to develop a test case management system that will use Semantic Web technologies in the following sense:

1. Semantic Web languages for representing test cases as well as other related testware and software artifacts.
2. The associated tools to support processing of these various elements.

3. Structuring test cases

Central to test case management system is the notion of test case itself. Although many definitions are found in the literature, all seems to capture the same essential meaning despite the existence of some minor variations. For example, IEEE Standard Glossary of Software Engineering Terminology [6] defines test case as an entity comprising of a set of input values, execution preconditions, expected results, and execution postconditions. Furthermore, a test case is developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.

The following logical structure for test case [7] is consistent with the above definition, i.e a test case consists of:

1. Test Case ID
2. Purpose
3. Preconditions
4. Inputs
5. Expected Outputs
6. Postconditions
7. Execution History
   a. Date
   b. Result
   c. Version
   d. Run By

The extent of how the above structure should be detailed or extended depends on the intended use of test cases. In its simplest form, the above structure would be adequate to support simple inventory process for test cases. However, in order to support analysis tasks especially for test case reuse, more information need to be added. Furthermore, within the context of Semantic Web, the association or incorporation of metadata into test case structure as well as the provision of vocabulary in the form of ontologies are two fundamental aspects to be looked into.

3.1. Test case metadata and basic attributes

Metadata is generally defined as data or information about data. For our purpose, the distinction between metadata and data for test cases is of immediate importance. We refer the former pieces of information as test case metadata and the latter as test case attributes. We use the following principles to achieve this separation:

1. The information or values provided by test case metadata are required either for identification, selection, discovery, or other analyses on test cases; and these are not directly relevant to the act of conducting the required test itself.
2. The information or values provided by test case attributes must be available for conducting the test, once the test case is selected.
We now examine two metadata schemes that are currently connected to the Semantic Web initiative. They are (1) Dublin Core (DC) metadata scheme [8] and (2) the test metadata scheme [9] proposed by W3C Quality Assurance Working Group. The former is being used frequently by Semantic Web community when it comes to metadata, although it has been developed independently from W3C. As for the latter, although still at a W3C Working Group Note status, it is considered to be pioneer in this respect for the area of testing. Within the context of our discussion, we refer the former and the latter as DC scheme and QA scheme respectively.

The DC scheme essentially provides a basic element set that serves as a vocabulary of fifteen properties for use in describing resources [10]. The 15 metadata terms denoting these properties are: contributor, coverage, creator, date, description, format, identifier, language, publisher, relation, rights, source, subject, title, and type.

The QA scheme consists of metadata elements that are claimed to be a minimal set that can be applied to test cases [9]. The metadata terms proposed are: identifier, title, purpose, description, status, SpecRef, preconditions, inputs, expectedResults, version, contributor, rights, grouping, and seeAlso. As claimed by the proposing group, among the uses of the metadata under this scheme are to:

1. track tests during the development and review process
2. filter tests according to a variety of criteria
3. identify the area of the specification that is tested by the tests
4. aid in the construction of a test harness to automatically execute tests

Through a close examination of the intended meaning of all of the above DC and QA metadata terms, we observe the following:

1. Since DC metadata terms are supposed to be applicable to resources in general, all of them are possible candidates to be part of test case metadata.
2. Several metadata terms from both schemes are similar and have common intended meaning. This could be due to the QA scheme adopting them from the DC scheme. They are:

<table>
<thead>
<tr>
<th>Table 1. Common terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
</tr>
<tr>
<td>contributor</td>
</tr>
<tr>
<td>description</td>
</tr>
</tbody>
</table>

3. There are DC metadata terms that are relevant to testing but are not adopted by the QA scheme. They are: creator, date, relation, source, and subject.

4. Some of the QA metadata terms are considered to be part of test case attributes based on the separation principles stated earlier. They are: SpecRef, Preconditions, Inputs, and ExpectedResults. In addition, the attribute Postconditions as normally being included as part of test case definition is excluded from this scheme.

Based on the above observations, we derive a basic core structure each for test case metadata and test case attributes for use in SWbSTs, based on the following principles:

1. Use DC terms as best as possible since it is already well established. The usage is denoted by the prefix dc.
2. Adopt QA terms whenever applicable. The usage is denoted by the prefix qa.
3. Introduce new terms whenever necessary. This is denoted by the prefix xx.

The basic core structures derived are:

1. Test case metadata consists of:
   a. dc:identifier
   b. xx:group_id
   c. dc:creator
   d. dc:contributor
   e. dc:title
   f. dc:description
   g. qa:purpose
   h. qa:status
   i. qa:Version
   j. dc:rights
   k. dc:source
   l. dc:subject
   m. dc:relation
   n. qa:grouping
   o. qa:SeeAlso
   p. xx:TestType (technique)
   q. xx:Execution History (a series of)
      i. dc:date
      ii. xx:result
      iii. xx:tester

2. Test case attributes consists of:
   a. dc:identifier
   b. xx:group_id
   c. qa:SpecRef
   d. qa:preconditions
   e. qa:inputs
   f. qa:ExpectedResults
We shall use the above basic core structures in the ensuing discussion.

### 3.2. Ontologies

The most frequently quoted definition of ontology in the Semantic Web literature is the one by Gruber [11] which refers an ontology as a formal, explicit specification of a shared conceptualization. In simple terms we can view an ontology as providing a vocabulary for the basic terms and relations used to describe certain domain or topic of interest.

A good summary of the various classifications of ontologies is given in [12]. At this juncture, the following types of ontologies which are due to [13] are of our interest:

1. **Upper Level Ontologies** which are domain independent ontologies that describe generic concepts such as space, time, and events.
2. **Domain Ontologies** which describe the vocabulary for a given domain of interest.
3. **Task Ontologies** which describe the vocabulary required to perform generic tasks or activities.
4. **Application Ontologies** which describe the vocabulary of a specific application, particularly pertaining to concepts corresponding to the roles performed by entities in a given domain while performing some task or activity.

Within the context of our discussion, ontologies are needed to provide vocabularies that would be required for the system to understand the meaning (or intended use) of the test cases it has to process. For example, when test case re-use is sought, a set of test cases would be selected as potential candidates when given a set of criteria. However, by examining the metadata provided such as the keywords found in the dc:subject term, and with the help of the domain and application ontologies for the software in question, grouping of these test cases can be done more accurately. Furthermore, again with the help of some ontologies, test cases that are found to be not relevant to the subject or domain of interest can be filtered out. The above four generic types of ontologies are indeed important to most applications. However, to support software testing activities, we need to develop more specific domain ontologies, particularly one that provides vocabulary for software testing itself. A good account of representative ontologies for the domains of e-commerce, medicine, engineering, enterprise, chemistry, and knowledge management can be found in [14]. Although to date, works on software engineering ontologies are making good progress, little has been done on software testing (see [15]). Hence, although these available ontologies are useful and may be applicable to our purpose, the development of a software testing ontology is critical to the success of our research. The ontology design principles and methods [14] and the summary software testing body of knowledge [16] can be used as a basis for developing the software testing ontology for SWbSTSs. Until this is achieved, only limited reasoning on test cases can be carried out.

### 3.3. Linkages to other artifacts

Complete reasoning on test cases would also have to rely on references to other testware such as test basis, test plan, and test design specification as well as other software artifacts such as use case and design models.

The following questions then arise:

1. How much of these linkages should be captured?
2. Would it be possible to find a base set of linkages that can dynamically be extended and added to the test case structure?

We shall leave the resolution of these issues as part of our future work.

Figure 1 shows the conceptual structure of test cases in relation to other entities.

---

**Figure 1. Conceptual structure of test cases**

---
4. Representation techniques

It is important to note that although RDF is developed for describing metadata, it can also suitably be used to represent the data level of entities. Hence, it is possible that both test case metadata and test case attributes be represented by RDF in one single setting. However, in this study, we have opted to use RDF for representing test case metadata and XML for representing test case attributes. This is in line with the “separation of concerns” principle. During implementation, we do not discount the possibility that the two descriptions be combined under one single RDF file, if efficiency can be improved.

We shall now use the following example test cases in the ensuing discussion. These are equivalence class test cases for the Triangle Problem which are taken from [7]. They are:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Expected Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>WN1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>Equilateral</td>
</tr>
<tr>
<td>WN2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>Isosceles</td>
</tr>
<tr>
<td>WN3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>Scalene</td>
</tr>
<tr>
<td>WN4</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>Not a Triangle</td>
</tr>
</tbody>
</table>

Since the above test cases are associated with a certain testing technique (in this case Equivalence Class Testing) and are normally performed together, it seems appropriate to represent all of them in one single XML file as shown in Figure 2. It is also then logical to assume that they can share one common metadata and this is shown in Figure 3. Note that in both cases, xx is implemented as the default namespace.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<test_cases group_id="ECTP1"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:qa="http://www.w3.org/TR/2005/NOTE-test-metadata-20050914/"
xmlns:xx="http://must.edu.my/swbsts/">
  <test_case dc:identifier="WN1">
    <qa:inputs>
      <param name="a" value="5"/>
      <param name="b" value="5"/>
      <param name="c" value="5"/>
    </qa:inputs>
    <qa:ExpectedResults>Equilateral</qa:ExpectedResults>
  </test_case>
  <test_case dc:identifier="WN2">
    <qa:inputs>
      <param name="a" value="2"/>
      <param name="b" value="2"/>
      <param name="c" value="3"/>
    </qa:inputs>
    <qa:ExpectedResults>Isosceles</qa:ExpectedResults>
  </test_case>
  <test_case dc:identifier="WN3">
    <qa:inputs>
      <param name="a" value="3"/>
      <param name="b" value="4"/>
      <param name="c" value="5"/>
    </qa:inputs>
    <qa:ExpectedResults>Scalene</qa:ExpectedResults>
  </test_case>
  <test_case dc:identifier="WN4">
    <qa:inputs>
      <param name="a" value="4"/>
      <param name="b" value="1"/>
      <param name="c" value="2"/>
    </qa:inputs>
    <qa:ExpectedResults>Not a Triangle</qa:ExpectedResults>
  </test_case>
</test_cases>
```

It is important to note that the above XML file can also be associated with a schema such as a DTD or XML Schema file to validate its syntactic structure. The latter has more advantages since it is XML-based and also provide a set of datatypes to enrich the description.

As mentioned earlier, we need to associate an RDF description to the whole set of these test cases for metadata description.

For illustrative purpose, we focus to incorporate only the following metadata elements:

1. dc:identifier {refers to the test cases group_id}
The above metadata elements can be incorporated into an RDF file as follows:

```
<?xml version="1.0"?>
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:qa="http://www.w3.org/TR/2005/NOTE-test-metadata-20050914/"
xmlns="http://must.edu.my/swbsts/"
><rdf:Description rdf:about="#ECTP1">
  <dc:description>This resource contains test cases.</dc:description>
  <qa:purpose>These test cases are used to test a program solving the Triangle problem.</qa:purpose>
  <dc:source>Unknown</dc:source>
  <dc:subject>triangle,equilateral,isosceles,scalene</dc:subject>
  <dc:relation rdf:ID="BVTP1">Test cases using Boundary Value Testing</dc:relation>
  <TestType>Equivalence Class Testing</TestType>
</rdf:Description>
```

Figure 3. RDF representation of test case metadata

From the above example we observe the following:

1. The rdf scheme itself provides a vocabulary of terms that can be used to replace some that we have identified earlier. In this case, we have replaced dc:identifier with rdf:about.
2. It would be more convenient to combine dc:description with qa:purpose into one single element.
3. The keywords in the dc:subject element require math ontology for interpretation.
5. The elements dc:description, qa:purpose, and dc:source are useful for test case analysis especially for re-use purpose. Good keywords must be extracted from their contents and be put in the dc:subject element.

The above observations seems to suggest the following:

1. The basic core structures for test case metadata and test case attributes can still be minimized.
2. Ontology pertaining to the software problem domain is also important for test case analysis purpose.

Finally, there are two more issues that need to be highlighted as far as test case representation is concerned:

1. Is there any need for a test case to be represented individually, and not as a group?. If so, how would this affect the metadata organization?
2. What is the scheme to be used for constructing identifier for test case, test case group, and other testware or software artifacts?

5. Conclusion and Future Work

In this paper, we have highlighted the potential use of Semantic Web technologies from the perspective of representing test cases for wider analytic use. Some test case design and representational issues were discussed, which suggests that further research need to be carried out in order to support the development of full blown Semantic Web-based software testing systems. Future work should also include the development of techniques in dealing with mediation and ontology alignment problems.

6. References

Achievements, Challenges, Dreams”, Future of Software Engineering (FOSE ’07), Publisher, Location, Date, pp. 1-10.
Testing Using Program Instrumentation and Visualization
V. LAKSHMI NARASIMHAN
Department of Computer Science

Western Kentucky University
KY 42101
Email: lakshmi.narasimhan@wku.edu

ABSTRACT

This paper presents a mechanism to perform software testing using program instrumentation and visualization. Through program instrumentation, the problem of test data generation is treated entirely as a numerical optimization problem. As a consequence, this method does not suffer from the disadvantages of symbolic execution or a constraint simplifier in order to analyze a program. Instead, program instrumentation is used to solve a set of constraints without explicitly knowing their form. The system supports not only the generation of integer and real numerical data types, but also non-numerical discrete types such as characters and enumerated types. Thus, the approach combines the fundamental elements of symbolic execution and program instrumentation to produce a system, which offers superior performance for program testing. A suitable visualization environment has also been implemented along with several metrics generator.

Keywords: symbolic execution, program instrumentation, test data generation, penalty functions, visualization environment, metrics generator.

1. INTRODUCTION

As shown in Fig.1, large information systems have millions of lines of code, written in many programming languages and developed by many organizations and people. To test and maintain such systems is a near impossible task. The need for better program testing tools and techniques has long been recognized because it is well known that testing alone can take up over 50% of the software production cycle [1].

Fig.1: Typical Information Systems

IEEE definitions on software testing state that program testing involves executing a test program with a set of input data and comparing the output data with what is expected; any observed differences reveals the presence of an error. To test a program completely, one must exhaustively execute the program with every possible set of input data, which is not practical [2]. Instead, testing is usually limited to a subset of the paths in the program and therefore, care must be exercised in choosing test cases which will maximize the chance of finding errors. In the simplest case test data may be generated randomly, but this method of testing has been found to be unsatisfactory due to the poor test coverage achieved [3].

Program testing techniques based on control flow analysis have been developed which involve selecting test data according to a set of predefined criteria such as statement coverage, branch coverage, and condition coverage. These criteria have been further refined [1] (e.g., several variants of branch coverage such as decision coverage, decision/condition coverage and multiple condition coverage) to allow the testing of compound branch predicates. Besides control flow analysis, other forms of test data generation have been developed including data flow analysis [4,5], domain testing [6],
mutation analysis [7,8] and rule-based approaches [9].

Testing is better achieved if the tester has some knowledge about the structure and function of the program being tested so that the test cases can be selectively hand-chosen. However, large programs are usually developed by a team of software engineers and, in such instances, the generation of test cases becomes very time-consuming and impractical. Efforts have been made to automate this process, but it is severely restricted by the fact that the problem of generating test data to execute any arbitrary path, in general, is unsolvable [10]. As a consequence, many of the conventional testing strategies cannot be fully implemented in an automated system.

This paper presents a system for the generation of test data, whereby the problem of test data generation is formulated and solved completely as a numerical optimization problem. Instead of using symbolic execution, program instrumentation is used to solve a set of constraints without explicitly knowing their form. As a consequence, this method does not suffer from the difficulties in handling loops, subprogram calls, and array references associated with symbolic execution [21]. The system supports not only the generation of integer and real numerical data types, but also non-numerical discrete types such as characters and enumerated types. The approach chosen combines the fundamental elements of symbolic execution and program instrumentation to produce a system, which offers superior performance for program testing. The system has a built-in metrics generator, which generates a number of useful metrics, while the visualizer facilitates code comprehension.

The paper is organized as follows: section 2 reviews testing strategies in general, while section 3 details the method of symbolic execution. Section 4 details numerical method for test data generation, followed by the design of the alternative system developed by us in Section 5. The technique employs a new program instrumentation technique and an optimization procedure for test data generation. Section 6 describes the overall testing and test data generation system, followed by the conclusions, which outlines the possible future work in this area.

2. TESTING STRATEGIES

Testing is typically carried out at various stages as shown in Fig.2. Unit testing involves the smallest unit or module of the software and a typical test is meant to uncover errors in the following: Interfaces, Local data structures, Boundary conditions, Independent Paths, Error Handling paths, etc. The procedure demonstrates the correctness of module design (i.e., Detailed Design) and is white box oriented.

Integration testing concerns the construction of the system or "assembling" the units or modules. It uncovers errors associated with interfacing, based on internal or external specifications. It can be done at three levels, namely, Program, Subsystem and System Integration levels. Both top-down and bottom-up approaches to integration testing are possible.

(See Appendix)

Fig.2: Levels of Software Testing

System testing concerns Functional testing, Requirements validation, Pre-acceptance testing, Recovery testing, Simulation of hardware failures or create data errors, Security testing, Exercising the security aspects/features of the system, Stress testing, Exercising the peak volume of the system, Performance testing, Checking for response time, throughput rates, Configuration testing and Testing on different hardware configurations. Basically, system testing verifies that the system as a whole is structurally and functionally sound.

Acceptance testing is related to contractual obligations in that it deals with comparison of the final system to its initial requirements and its final acceptance. Questions like “is the system ready for conversion or operational use?”, “does it meet the previously defined system specifications and acceptance criteria?” and whether it works the way it should, under the user's operational environment are tested in acceptance testing phase.

Other types of testing includes, but not limited to, Conversion Testing, Installation Testing and Maintenance Testing. Testing can be done in i) top-down, ii) bottom-up or iii) “combo” manner. Under the combo category, one can combine various testing techniques in order to exploit the strengths of various techniques. For example one set of combination approach to testing could be as shown in Fig.3.

(See Appendix)

Fig.3: Combo Approach to Software Testing
The million dollar question, however, is “When to STOP Testing?” There are several approaches to address this question, which includes: setting of typical Stop-Test Criteria, when time and money runs out, test cases no longer uncovers more bugs, until the effort or costs of conducting test outweighs the “returns”, estimates from the test plan, quality Metrics (Measurable) Approach, no high severity bugs, achievement of test coverage goals, trends of reported bugs, decreasing rate with improved coverage, a measure of confidence level and stability of system. However, the process of testing never ends, so long as the software is being used...

3. TEST DATA GENERATION USING SYMBOLIC EXECUTION

Most of the currently available testing and test data generator systems are based on the method of symbolic execution (see [12-15] for instance) in which symbols are assigned to input variables and subsequent uses of input variables are then expressed in terms of these symbols. Path predicates for each line in the test program are then generated which reflect the necessary requirements for a set of input data to cause the execution of a particular line in the program. This process of generating test data is better illustrated through an example.

(See Appendix)
Fig.4: Example Test Program and Corresponding Control Flow Graph

Consider the program given in Fig.4 taken from [16] and its corresponding control flow graph. The path that must be taken for the while loops s8 and s11 to be executed exactly twice, each time exercising the true branch of the if statement s14, is given by the condition TTTTTTTF. If the input variables n and d are initially assigned the symbols X and Y respectively, then after the assignment statements s2 to s4, t=Y and r=X. Consider the requirement for the first while loop (i.e., statement s8) to be iterated exactly twice: In order to initiate at least one iteration it is necessary that r >= t or X >= Y. After the first iteration t is now equal to Y*2 and so for a second iteration it is also necessary that r >= t or X >= Y * 2. To cease subsequent iterations it is necessary that r<t or, in other words, X<Y*4. Therefore, the necessary condition that the first while loop is executed exactly twice is given by:

\[ X \geq Y \text{ and } X \geq Y * 2 \text{ and } X < Y * 4 \]

Similarly to execute both the first and second while loops exactly twice (each time exercising the true branch of the if statement), necessary condition is:

\[ X \geq Y \text{ and } X \geq Y * 2 \text{ and } X < Y * 4 \]

For programs having many conditionals and loops, the path predicate becomes very long (even for this short example it is quite long). However, the path predicate can be rearranged and simplified to remove redundant conditions or statements. For instance, in the above example, the resulting path predicate is: X>3*Y AND X<4*Y, which can be expressed as a system of inequalities \{X>3*Y, X<4*Y\}. The set of test data, which will cause the desired path to be executed, is found by solving this system. If the solution can be found (i.e., the set is consistent) the test path is feasible and the solution to this set will be the input data which will execute the given test path. On the other hand, if the system is inconsistent the test path is infeasible. In the present example, the system is consistent and so the desired path is feasible, and any set of input data satisfying the above conditions will cause the desired path to be executed (e.g., X=3,Y=1 or n=3,d=1).

4. NUMERICAL METHOD FOR TEST DATA GENERATION

Miller and Spooner [17] proposed a method in which test data generation is formulated and solved as a numerical optimization problem. Instead of symbolically executing a test program, instrumentation statements [18] are inserted selectively at various places in the test program. For each statement affecting the control flow, the associated branch predicate is rearranged and used to produce a path constraint of the form ci=0, ci>0, or ci <0, where ci is a data-dependent real-valued constraint value. This value is a measure of how close the ith path constraint is to being satisfied. As an example, consider the if statement given in Fig.5.

Statement 1;
if x = y then
statement 2;
end if;
statement 3;

Fig.5: Example of Program Instrumentation

The above statement has a branch predicate x=y which can be rearranged into the form c=0 where, for example, c = \[x - y\] or c = \((x-y)^2\). This constraint value c then gives a measure of how close x is to y and, as c approaches 0, as required by the path constraint, the necessary condition for the true branch
being executed becomes closer to being satisfied. Note that the \( c_i \) values are arranged so that each \( c_i \) is positive when the corresponding constraint is satisfied.

One can now choose a continuous real-valued function \( f(c_1, c_2, \ldots, c_i, \ldots, c_n) \) such that

\[
\begin{align*}
  f(c_1, c_2, \ldots, c_i, \ldots, c_n) &< 0 \text{ if } i, c_i \text{ is strictly negative} \\
  f(c_1, c_2, \ldots, c_i, \ldots, c_n) &> 0 \text{ if } i, c_i \text{ is strictly positive}.
\end{align*}
\]

Note that the definition of the function ensures that \( f(c_1, c_2, \ldots, c_i, \ldots, c_n) > 0 \) only when all path constraints are satisfied. Test data can now be generated by picking an initial set of data and then applying numerical techniques for constrained maximization, stopping when \( f(c_1, c_2, \ldots, c_i, \ldots, c_n) \) becomes positive. Values of \( f(c_1, c_2, \ldots, c_i, \ldots, c_n) \) for each iteration in the maximization process are computed from the constraint values \( c_i \) returned by the instrumentation statements inserted in the test program.

5. **ALTERNATIVE PROGRAM INSTRUMENTATION**

The test data generation system presented in this paper adopts the approach of using program instrumentation to return the values of the constraints, as given by Miller and Spooner [17], but the optimization of test cases is achieved through the use of a set of penalty functions. Instrumentation statements are inserted selectively in the test program to return various information values to the test data generator on the state of different variables and their test coverage during the program's controlled execution. Typically instrumentation is needed for each control statement and is placed in a position such that it is executed immediately before the conditional in the control statement is evaluated. References [23] discusses iterative methods for testing and [24] discusses another new approach to testing.

(See Appendix)

**Fig.6: Instrumented example test program**

In the case of if statements, instrumentation is inserted immediately before it, but in more complex control structures such as loops, two statements need to be inserted - one immediately before the control structure and one at the end of the sequence of statements contained within the structure. The former is used for the first iteration of the loop statement, while the latter statement is used for subsequent iterations, as control flow does not return to the first instrumentation statement. The structure of the instrumented test program is kept as closely as possible to the original program, but statements which cannot be accommodated for (e.g., case structures) have to be transformed into their equivalent if statements. This concept is better illustrated with an example. Fig.6 gives the instrumented version of the program shown in Fig.4 with the instrumentation statements shown in bold italics.

The variables used for the purpose of instrumentation are as follows: \( i \) gives the number of branches executed at any given time, \( k(i) \) represents the branch number for each branch statement, \( \text{type}(i) \) is the enumerated type \((\geq, >, \text{eq})\) which represents the relational operators \( \geq, >, = \) respectively used in the branch statement and \( g(i) \) gives the constraint value of the branch predicate. Additional information such as timestamps for time profiling, or other coverage information may be easily obtained by including appropriate assignments.

The program is executed under very controlled conditions by the test driver so that the test data executes the same selected path on each execution. This is achieved by the following procedure: 1) The number of control statements executed is monitored by noting whether a specified number of branches (PathNodeCount) have been executed which enables a certain number of branches to be executed during execution without losing control of the test program. 2) The control flow during execution is forced to take a specified path by modifying each branch statement predicate to include two additional conditions. Each branch predicate \( p_i \), is transformed into a modified branch predicate of the following form:

\[
p_i' = (p_i \land \neg \text{ContExec}) \lor (\text{ContExec} \land \text{PathToTake}_i)
\]

where, \( \text{ContExec} \) is a flag which indicates whether or not each branch statement should be forced to take a specified path and \( \text{PathToTake}_i \) is a flag which indicates the particular branch the \( i \)th branch statement should be forced to take regardless of the values of variables in the path predicate. Both \( \text{ContExec} \) and \( \text{PathToTake}_i \) are initialized prior to each execution of the test program.

5.1 **OPTIMIZATION PROCEDURE USED FOR TEST DATA GENERATION**

Generating test data in our system is formulated as a constrained optimization problem. Formally, the general non-linear programming optimization problem can be stated as:
Minimize the objective function (eq.1):

\[ f(x), \quad x \in \mathbb{R}^n, \]

subject to a set of non-linear path constraints (eq. 2):

There is no simple procedure to handle non-linear path constraints in optimization problems. But, by employing a penalty function \[19\], the constrained optimization problem posed by equations (1) and (2) can be transformed into a simpler, unconstrained optimization problem with the objective function being the following (eq. 3):

The weighting factors \((w_i)\) are assigned large positive constants. The term \(w_i \cdot G[gi(x),type_i]\) represents the penalty function of the \(i\)th path constraint with a constraint value of \(gi(x)\) and the relational operator represented by \(type_i\). The positive penalty function \(G\) is chosen such that if the constraint is satisfied, the penalty function is very small and if not it is large. To enable the problem to be minimized, the penalty function is made proportional to the margin by which the constraint is violated. The functions chosen for this system are shown in Fig.7 where, \(\xi\) in case-1 is a small positive constant included to ensure the origin is taken into account properly.

(See Appendix)

Fig.7: Penalty Functions

Note that the objective function is the sum of the penalty functions for each constraint and therefore, the smaller the objective function, the closer the point \(x\) is to satisfying all the constraints. Fletcher and Powell’s optimization technique \[20\] which uses conjugate directions was chosen for this system to minimize the objective function \(f'(x,w)\). Values of \(g_i(x)\) used for calculation of the penalty function are obtained during the controlled execution of the test program through program instrumentation. During optimization, the values for \(w_i\) are increased, thereby causing the effect of the penalty \((w_i \cdot G[g_i(x)])\) to be diminished for points satisfying the constraint and to be increased for those points which violate the constraint. It has been shown by \[21\] that as \(w_i\) increases, the solution to the problem posed by (3) converges to the original problem given by equations (1) and (2).

The discussion so far has assumed that branch predicates are simple, i.e., there is only one relational operator e.g. \(x>y\). However, this system has the ability to handle predicates which involve logical operators such as "AND" and "OR" operators. In cases involving these operators, special rules are applied to generate an appropriate penalty function. Thus, if \(p_1\) and \(p_2\) are two simple predicates with constraint values \(g_1\) and \(g_2\) respectively, a separate procedure is adopted for logical operators.

6. THE OVERALL TEST DATA GENERATION SYSTEM

(See Appendix)

Fig.8: ADTEST Test Data Generation System Architecture

The test data generation system consists of the following three parts (see Fig.8), while the inner operation of the system is shown in Fig.9:

(i) Test Program instrumentation module performs the following functions:
• Test programs analysed by the instrumentation module (parser written using YACC & LEX)
• Re-formatting and insertion of instrumentation statements
• Collection of information on variables and control structure stored in a common databases
• Several software metrics generated for each test program unit (e.g., McCabe, Halstead, Knot analysis, LNCJS, etc).
• Instrumented test program is compiled together with software test drivers necessary to enable the controlled execution of the test program by the test data generator.
• The test driver contains code for type constraint checking and type conversion/s of the input variables before the test program is executed.

(ii) Test Path Generator performs the following functions:
• Individual subprograms or combinations of subprograms selected for testing
• Test paths selected either manually or automatically using various testing strategies
• The control flow graph for the selected subprograms is graphically displayed
and the user can hand-select a test path between each of the control nodes.

\[
g_i(x) = \begin{cases} 
> & 0, i = 1, 2, \ldots, m \\
\geq & \quad \quad \\
\neq & 
\end{cases}
\]

\[
f'(x, w) = \sum_{i=1}^{n} G(g_i(x), w_i, \text{type}_i)
\]

(iii) Optimization Module performs the following functions:

- Optimisation is performed in an attempt to generate test data to execute the test paths produced by the test path generator.
- Once the results have been displayed, new test paths may be selected using the test path generator or a new test program may be selected.

(See Appendix)

Fig.9: Inner Working of the Program Instrumentation System

Some aspects of the system was proposed by Scott [15] for programs written in Fortran, but it has been considerably refined and modified to generate data for Pascal and Ada based systems. The system supports the generation of Ada's integer and real numeric types, as well as non-numeric discrete-type input data structures. In the latter case, such structures are generated by representing them with their ordinal values during optimization and converting the results of the optimization back to their original type before they are displayed. In some cases, additional constraints need to be placed upon the values which they may take (for example a character's ordinal value must be in the range of 0 .. 255, and range constraints for variables). Such constraints must be checked for prior to executing the program and, in addition, at each step during the optimization search.

Note that it is not possible to generate all data types nor may they be desired. For example, it may be possible to actually generate a pointer address, but its value from optimization would be meaningless since memory allocation would not necessarily be contiguous. However, this is not to say that variables of such types may not be used in the program. Any type of variable may be used as long as it is not used in a branch predicate. If a logical or relational operation does need to be performed in a branch predicate, a temporary boolean variable must firstly be used in an assignment statement to record the value of the operation, which is subsequently used in the branch predicate.

The software system supports strings of a limited length by representing each string as a weighted sum of each character element of the string. Since strings are seen as an important element in programming, alternative ways of generating strings are being investigated. On a further note, a complication arises with the overloading of the "<", ">", "<=", and ">=" operators in ADA. Redefining "<" to mean ">" would make test data generation very difficult as it relies upon these operators having their standard meaning. To overcome this problem, the restriction that such relational operators only be used with their standard meanings is imposed and, in our view, this is not too restrictive as it would be poor programming practice to do otherwise.

The system also supports a form of visualization (see Fig.10 for example) and several types of metrics generation.

(See Appendix)

Fig.10: An Example Visualization System (Adapted from [22])

7. CONCLUSIONS AND FUTURE WORK

We have demonstrated a new type of software testing and related test data generation process, which does not involve the complications of symbolic execution. Instead, instrumentation statements are inserted into the test program to dynamically return information about the values of the constraints which must be satisfied in order for the input data to execute a selected test path. This procedure has the advantages that the form of each constraint need not be known explicitly, and problems associated with symbolic execution do not occur. This method is flexible in that additional information such as timestamps and test coverage information, may be easily obtained by adding appropriate assignments in the instrumentation. A program visualizer (and metrics generator) have also been developed over this system.

Future work on this topic will concentrate on the problem of test data generation in real-time programming. Test data generation in this case is
complicated by the fact that, unlike sequential programs, repeated execution of real-time programs with the same input data, may not necessarily produce the same output. Questions such as what type of instrumentation is needed in testing real-time systems and how such instrumentation can be introduced into real time systems without perturbing the complex timing relationships that exist between different processes need to be examined carefully.

8. ACKNOWLEDGEMENTS
The author would like to thank Dr. Gallagher for his help on various aspects of this paper.

9. REFERENCES
Appendix

- Level 1: Black-box techniques only
- Level 2: Black-box and Coverage Measurement
- Level 3: Black-box, White-box and Coverage Measurement
- Level 4: Black-box, White-box, Coverage and Complexity Measurement

Fig.3: Combo Approach to Software Testing

Fig.2: Levels of Software Testing

Program calculates the quotient and remainder of the division of two positive integers $n$ and $d$, and the corresponding control flow graph.

```plaintext
1: procedure quotient(n: Some Integer, d: Some Integer) is
2:     q: Some Integer := 0;
3:     r: Some Integer := n;
4:     t: Some Integer := d;
5:     begin
6:         while $r \geq t$ loop
7:             $t := 2 \times t$;
8:         end loop;
9:         while $t \neq d$ loop
10:             $q := q \times 2$;
11:             $t := t / 2$;
12:             if $t \leq r$ then
13:                 $r := r - t$;
14:                 $q := q + 1$;
15:             end if;
16:         end loop;
17:     end quotient;
```

Fig.4: Example Test Program and Corresponding Control Flow Graph

Statement 1;
if $x = y$ then
statement 2;
end if;
statement 3;

Statements
---------->
Replaced

Statement 1;
c = |$x - y|$;
statement 2;
statement 3;

Fig.5: Example of Program Instrumentations

```plaintext
init()
i := 0;
store(type value, g value, p value)
i := i + 1;
type(i) := type value;
g(i) := g value;
p(i) := p value;
if i = PathNodeCount then
    exit;
end if;
exec()
return (p(i) AND NOT ContExec) OR (ContExec AND Path(i));
```

Fig.6: Instrumented example test program
<table>
<thead>
<tr>
<th>Case</th>
<th>Type</th>
<th>Constraint</th>
<th>G[g(x)]</th>
<th>Effect of G[g(x)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>gt</td>
<td>g(x)</td>
<td>exp(-g(x))</td>
<td>G[g(x)] → 0 as g(x)</td>
</tr>
<tr>
<td>2</td>
<td>ge</td>
<td>g(x) &gt; 0</td>
<td>exp(-((g(x)+ξ)))</td>
<td>G[g(x)] → 0 as g(x) becomes &gt; 0</td>
</tr>
<tr>
<td>3</td>
<td>eq</td>
<td>g(x) = 0</td>
<td>exp(</td>
<td>g(x) + ξ</td>
</tr>
<tr>
<td>4</td>
<td>ne</td>
<td>g(x) ≠ 0</td>
<td>exp(-((g(x)</td>
<td>+ ξ))</td>
</tr>
</tbody>
</table>

**Fig.7: Penalty Functions**

**Fig.8: ADTEST Test Data Generation System Architecture**

**Fig.9: Inner Working of the Program Instrumentation System**
Figure A-1. Main representations used within SEE-Ada

Fig.10: An Example Visualization System
(Adapted from [22])
A STUDY ON FAULT INJECTION-BASED APPROACH FOR DEPENDABILITY EVALUATION

S. M. Masum, S. Saha and M. Yeasin
Department of Electrical and Computer Engineering, The University of Memphis,
TN 38152, USA

Abstract

Fault Injection is an effective solution to the problem of validating highly reliable systems. Fault Injection is the process of corrupting a data state during program execution. Fault injection based testing is the process of determining the effect of that corruption. The testing may consist of simply measuring whether the corrupted state affected a particular output, or the testing may determine whether system attributes such as safety, security, or survivability have been affected. Fault injection based testing is often used in large system development projects. This paper presents a study on fault-injection-based approach for dependability evaluation.

I. INTRODUCTION

Testing is a crucial step in the development of a software-intensive system, as it checks the compliance of a system to the end user requirements [1]. The development of software testing systems must be performed in effective and efficient manner. It is easy to see that an effective testing is a very good indicator of the quality product and efficient testing procedure to ensure the faster development cycle that is an imperative requirement for large organization. The prime objective of the system testing is to cover all forms of the testing techniques related to systems to ensure the successful development and application of software and technology.

With greater reliance on computers in a variety of applications, the consequences of failure and downtime have become more severe. Computers employed in critical applications often incorporate redundancy to tolerate faults that would otherwise cause system failure. A fault-tolerant computer system’s dependability must be validated to ensure that its redundancy has been correctly implemented and the system will provide the desired level of reliable service. Fault injection—the deliberate insertion of faults into an operational system to determine its response offers an effective solution to this problem.

This paper presents a study on fault-injection-based approach for dependability evaluation. The rest of this paper includes a discussion about fault, error, and failure in Section 2; Section 3 presents a review of related works; Fault injection model is presented in Section 4; Section 5 shows the impacts of faults on program behavior in our simulation; At last, Section 6 concludes this paper.

II. FAULT, ERROR, AND FAILURE

A fault is a variation in a hardware or software component from its anticipated function. Faults can occur during all stages in a computer system’s evolution specification, design, development, manufacturing, assembly, and installation—and throughout its operations. Most faults that take place before full system exploitation are discovered through testing and eliminated. Faults that are not detached can reduce a system’s consistency when it is in the field. In spite of the potential for such underlying faults in computer systems, most fault-injection reviews focus on the faults that occur during system operation [2].

Software faults are caused by the incorrect pattern, design, or coding of a program. Underlying faults or bugs in the code can shell during operation especially under heavy or unusual work loads and eventually lead to system failures [2]. For this reason, software fault injection is employed mostly for testing programs or system or software implemented fault-tolerance mechanisms.

When a fault causes an invalid change in machine state, an error occurs. The time between fault occurrence and the first manifestation of an error is called the fault latency [2]. Although a fault remains localized in the affected code, multiple errors can originate from one fault site and propagate throughout the system. These will cause a propagating error after a period of time, called the error latency [2]. When the fault tolerance mechanisms perceive an error, they may commence several actions to handle the fault and contain its errors. Recovery occurs if these actions are successful otherwise the system eventually
malfunctions and a failure occurs.

Figure 1 shows an example to clarify the definitions of fault, error, and failure [2]. Suppose a permanent stuck-at-0 (s-a-0) fault [2] affects a memory bit with an initial value of logical 0. Sometime later, an error occurs when a logical 1 is written into this bit. (If the faulty value had been opposite the initial value of this bit, an error would have manifested immediately with no fault latency.) The next read from the memory bit obtains the s-a-0 value instead of the correct value, 1, thereby detecting an error.

Figure 1: Illustrative example of a fault, an error, and a failure [2].

Proper service continues if the system’s fault-tolerance mechanisms can correct or mask this bit error. If not, service is disrupted. Here, a fault injection-based method has been studied for dependability evaluation which detects the existence of faults and to estimates the goodness of the code. It uncovers faults reliably to develop fault-tolerant software system. A unified approach has been adopted that is based on synergistic approach to improve the effectiveness of fault detection. One attempts to find more efficient testing strategies by fault injection, and the other tries to use the design characteristics of the software to start the testing phase early in the software development.

III. RELATED WORKS

The literature of software testing by incorporating fault-injection technique extends from theoretical in its inception phase to practical now in some commercial usage. There is an important distinction of types of faults or anomalies as described in [22] that should be tried for simulation which essentially gives a theoretical view of the problem. The advantages of such an approach should not be considered from correctness point of view. Rather, the point of what to be injected systematically leads to a safer system. This software technique has its scope in physical systems as well—particularly in a software embedded mechanical system comprised of three modules: (i) Software Control System, (ii) Actuators that perform the desired function, and (iii) Sensors that feed information to the software. But, the bottom-line of the work [22] is not to claim a system incapable of ever producing undesirable outputs, but to claim that it is known to be resilient to some anomalies tried, which may most likely infer that it might resilient to many other anomalies as well.

Fault-injection technique has its potential to be exploited in software systems ranging from small codes to large sophisticated systems. Some works has been carried that focuses on extending the use of fault-injection technique from one-person Independent Software Vendors (ISVs) to mass level [28]. Spanning of user-spectrum is indeed important from software development point of view. Manual injection, as an alternative to automated injection technique, often works as a good firsthand appreciation of the inherent tolerance of a system. Even a small number of injections, in the order of 25-30, work as a good indication to the tolerance of a system with repetitive processes, as pointed out in [28]. Some key places have also been figured out, which are more vulnerable to defects. For example, a fault-injector right after an input, is often very productive to yield a defective case. A mix of small tiny functions like perturbing an input data, or randomized input, is quite good to assess the confidence in a piece of code. Apart from an accessible source code, faults can also be injected to a black-box software module. Perturbing the output of a black-box, interface propagation analysis forces corrupt information into a system as it executes.

Different types of faults were analyzed in the initial phase of software fault-tolerance research [26]. But, essentially there was focus on faults that exists in a system. However, in [27], focus has been set on the characteristic of a program that is vulnerable, which is probable to make faults and which can hardly be found by random black box testing. It turns out that apart from evaluating the correctness of a code, it’s more necessary and effective to predict the erroneous behavior of black box where the code has already been compiled. In doing so, an important notion “sensitivity” has been introduced in [27], which can be thought of a finer granularity of the term “testability”. Testability of a program is its inclination towards exposing faults to a test scheme.
with inputs randomly selected from a particular input distribution. The structure and semantics of the code and the assumed input distribution governs the testability of a system. In contrast, sensitive is meant for a location of a code as opposed to the overall system. The sensitivity of that location refers to the probability of an error in that location to be reflected in the overall system failure. So, sensitivity is related to testability, because testability encompasses sensitivities of all parts of a program under an input distribution. Sensitivity analysis has been used as a dynamic method for analyzing code. In doing so, a three part fault/failure model has been figured out with execution, infection and propagation modules. It has been claimed, though several experiments that such analyses can unearth hidden faulty locations, which could have been undetected by random black box testing [27]. Although it is computationally intensive, it requires no human intervention. This approach adds another dimension to software quality assurance, by pointing out vulnerable codes that has a greater tendency to hide faults during testing phase, so that that code can be re-written. It can also help interpret the test result of black-box testing. More confidence can be attached with code with high sensitivity than that with low one. So, in brief, sensitivity is good at finding subtle bugs, which may have been hidden in conventional testing techniques. Although translating fault-injection from theoretical to practical domain is a complex job, some efforts have been made in the literature goes to a large extent in this regard. PSN (Piscas Safety Net) is an environment created to supplement the traditional testing methods using fault injection techniques and failure tolerance measurements [21]. Specifically, it targets at identifying weaknesses that could cause catastrophes and point out probable weak locations. This is one module of Whitebox Software analysis toolkit. It’s an implementation of extended propagation analysis (EPA), which uses injected faults at software and hardware level. EPA takes a program, an unacceptable outcomes description and an operation profile as input and returns potential weak locations. This has been applied to four different software control applications, where it has proved its value. As stated previously, the notion of failure tolerance has been used as a metric. Failure tolerance is the ability of a program to compute an acceptable result even if the program itself suffers from incorrect logic, and even if it is fed with incorrect data. A simple metric of failure tolerance is used, where a score of 0 means completely intolerant and 1 means totally tolerant. But, the most important is acceptable result, which is a range of acceptable values and is defined by the user from other subsystems requirements. PSN is a command line program which is activated at compile time. PSN instruments the code and performs extended propagation analysis at compile time. Also, the results are dynamically gathered as the software executes. It uses fault injection to alter data states during program execution and monitors outputs whether that is acceptable or not. An output is considered unacceptable if it is incorrect, non-existent or unsafe. The system has been tested in four case studies; two of them are safety-critical. The systems that were tested were (1) Yaw, which is a small yawdamp controller for a Boeing aircraft delivered to NASA, (2) Autopilot, which is a C source code for autopilot control of a Boeing aircraft, (3) Magnetic Stereotaxis System - an experimental device for performing neurosurgery and (4) Advanced Automated Train Control System. The results have been demonstrated in that work, which is very convincing. Some typical fault-injection models have been proposed in [3][4][5][6][7]. From the non-trivial viewpoint, some Machine learning approaches have been proposed for Software Testing [8], Software Modeling [9] and Software Debugging [10]. Ernst et al. [11] aim at detecting program invariants, through instrumenting the program at hand and searching for predetermined regularities (e.g. value ranges) in the traces. Brehelin et al. [8] consider a deterministic test procedure, generating sequences of inputs for a PLA device. An HMM is trained from these sequences and further used to generate new sequences, increasing the test coverage. In [12], the goal is to test a concurrent asynchronous program against user-supplied constraints (model checking). Grammatical Inference is used to characterize the paths relevant to the constraint checking. Xiao et al. [13] aim at testing a game player, e.g. discovering the regions where the game is too easy/too difficult; they use active learning and rule learning to construct a model of the program. A more remotely related work presented by Zheng et al. [10], is actually concerned with software debugging and the identification of trace predicates related to the program misbehaviors. In [11][12], ML is used to provide better input to ST approaches; in [8], ML is used as a post-processor of ST. In [13], ML directly provides a model of the black box program at hand; the test is done by manually inspecting this model. The underlying ideas behind fault injection are not new and there exists much literature describing how to employ them for hardware system validation, software testing and hardware design validation [15][17][18][4]. Unfortunately, the migration of those ideas into practical methods for software validation has not occurred, mainly due to concern about the plausibility of the anomalies injected. For example, in an integrated circuit, the failure classes are obvious:
stuck-at-one, stuck-at-zero etc. But for a software system, the internal data states are not as simple as just a ‘0’ or ‘1’ and hence the number of different data mutations that could be injected is intractable. The key concern surrounding all software fault injection methods is that the information collected may not be relevant for the actual problems that the software will experience in the future, i.e. the results of fault injection will not hold for the real problems that will eventually manifest. Further, the prototyping and commercialization of fault injection tools has been very limited (e.g. MOTHRA[16], SafetyNet from RST, Fault Tolerance and Performance Evaluator from the University of Illinois, etc.). Without automated fault injection tools, the analysis is rarely feasible.

IV. FAULT MODEL AND INJECTION METHOD

Software fault injection includes a family of techniques that instrument the original code with mechanisms, more code, that either modify the existing syntax or force the state of the code to be modified when the software executes. Either way, the code or its behavior is somehow changed. It is this process of modifying events that makes it ideal for predicting what might happen if future events get disturbed. Fault injection process does not tell how good the code is per se, but instead provides a worst case prediction for how badly the code might behave in the future, as opposed to how bad it might be.

The number of different anomalies that software can experience during its lifetime is almost always infinite and unknown. The anomalies that are of interest in here are those that can arise from code defects, caused by programming errors or design defects, human factor errors, or other external failures from hardware upon which the software depends for inputs or from other software. The combinations of anomalies from instances of these different problem sources are intractable. It is becoming common for software engineering standards to enumerate certain classes of problems that must not occur. Here, problems are usually defined in one of two ways: (1) a class of failure that must not occur, or (2) a fault class that can occur but the fault class must be shown to only cause acceptable outputs. However, IEEE standards have been criticized as “adhoc and unintegration” [19].

Since it is not possible to inject every possible anomalous operational scenario and then observe the software’s behavior, it is necessary to somehow ensure that the anomalies injected are representative of the types of anomalies that the software will experience during its life-time.

(See Appendix)

Figure 2: Anomaly detection: (a) Anomaly classes where A is the set of anomalies internal to the code and B is the set of anomalies outside the code and (b) Anomaly classes where A and B overlap [23].

Addressing some more questions is easier a framework is set up within which to approach the problem. There are two key classes of anomalies - (1) internal problems and (2) external problems. Even though both of these spaces are huge, there is only a small number of problems that are actually cared about – specially, those that will affect the software after it is deployed. If it can be shown that it is likely to be able to simulate anomaly classes that will exhibit behavior patterns equivalent to these important few, then the trick has been conjured up, that is needed.

To begin, it may be assumed that no modifications will be made to program P during its analysis. We can relax this constraint later. In Figure 2(a), let Γ represent the space of all undesirable anomalous events that could affect P’s output during P’s lifetime (if they were to occur). In Figure 2(a), A denotes that portion of Γ representing potential internal anomalous events and B denotes potential external anomalous events caused by human factor errors or external hardware and software failures that could be foisted on P during execution. Admittedly, most of Γ’s members are unknown, and hence Γ can be likened to a black-box. A represents either: (1) an effectively infinite and unknown space of anomalies, or (2) an empty and unknown space of anomalies. B represents an effectively infinite and partially unknown space of anomalies. A ∪ B ≈ Γ. Those anomalies in Γ that are not in A and not in B represent state corruptions that are not currently possible in P’s state as it executes with its current input space, but those anomalies could be forced into Ps state as it executes (e.g. by fault injection methods). Obviously, those members in Γ that are in neither A nor B are not anomalies that we care about for purposes of liability.

In some situations, an anomalous event may not occur until both an external problem occurs and an internal logic error is executed. This situation is shown in Figure 2(b) (which we believe is a more realistic portrait of how most instantiations of Γ will truly be partitioned). Here, anomalies in A ∩ B represent those anomalous events caused by bad incoming information and exercised faults. For example, it might be the case that bad incoming data
is required to exercise a fault in a manner that results in an anomalous internal event.

For the purposes of software accountability, the impact of the anomalies that are exclusively in A will be determined, to some degree, by testing the program P in the usual way. Traditional testing then gives us a handle on the liability consequences of the anomalies in A∩ the more complicated case of Figure 2(b), any anomaly in A∩B will require more than traditional testing to simulate. That is, traditional testing will be able to assess the impact of the A anomalies, but the B anomalies require something else.

Fault injection can play a role in providing this other information. Stated simply, simulating anomalies in B is possible using fault injection. This is because fault injection can be used to observe what effect external errors (such as sensor failures) corrupted file information, and human error factors, will have on P. Once again, the more complicated case in Figure 2(b) maps in a straightforward way into this discussion. It is easy to see that in order to simulate anomalies in A∩B, both fault injection and testing must be applied.

B can further be partitioned into b′ and b″, where b′ represents those problems that will occur in P’s future, and b″ represents those problems that could possibly occur in the future but in actuality will not. b′ is of finite size and b″ is probably of infinite size for most P’s. b′ and b″ are unknown both before and after fault injection analysis is performed.

The dotted circle in Figure 2(b) encompasses anomalies from A, b′, b″, and those not in A and not in B. It would be Utopian if the fault injection methods employed on P only happened to simulate members from b′, but to do this requires either incredible luck or omniscience. In any case, since b′ is unknown, criteria for success would be impossible to develop as well. Our approach for simulating the anomalies in the dotted circle in Figure 2(b) is to inject anomalies intrusively using state corruptions, not mutants. The code that we add to modify internal states is termed as perturbation function. A procedure for coding perturbation functions is further described in [20].

Anomalies may be injected in a random fashion. Random sampling from Γ without any knowledge of any of Bs members was applied, yet weaknesses in these systems were discovered. Advanced Automated Train Control (AATC) system, Hughes was able to take corrective actions at 12 places in the code after randomly sampling from Γ. These 12 potential problems had not been discovered by any other software safety methodology. Successes using the same fault injection approach on a medical application are also provided in [21]. In these studies, no members of B were known, yet random sampling from Γ discovered serious potential risks. This shows that fault injection is not only useful for assessing accountability, but when accountability is demonstrated, it can help developers pinpoint where problems were able to propagate from, thus playing a proactive role in decreasing accountability [22].

The next question that arises is whether or not the members of B that are simulated via fault injection methods will display representative propagation characteristics of the remaining space. By representative, we mean that they will share both the likelihood of propagation and a similar output class. Recent research suggests that distinct, incorrect data states (for the same test case) often tend to exhibit similar propagation behaviors, regardless of exactly what the corruptions are. As an example, suppose that at some point in the code some variable should have a value of 100, but has a value of 101. If this incorrect value causes a problem in the output, then it is likely that had the incorrect value not been 101 but 102, this too would cause a problem. One clue as to why this phenomenon occurs is the fact that specific partitions of B are likely to follow specific execution traces, since members of those partitions can only penetrate P at fixed points, and hence will probably follow similar sub-paths of P. If this claim is generally true, the absolute worst case induced by any member of b′ might be discernible from observing what happens when simulating a subset of the members of B. Results substantiating this conjecture are not yet in hand, but given the results from [21][24][25] (in combination with the fact that there are no other liability measurement solutions in our immediate future), this hypothesis cannot be dismissed outright without some justification. In fact, even if future research refutes the hypothesis advocating fault injection for absolute worst case prediction, fault injection will always be capable of building a list containing the leading candidates for that uncertain title.

V. FAULT INJECTION AND ITS IMPACTS ON SOFTWARE BEHAVIOR

Some early observation follows here from a simulation of the Fault Injection on two different versions of Quick sort algorithm. First version is a very basic one which does not have any error or exception handling and the second one has some exception handling component. Theoretically, the second implementation must be more dependable than the first one. The simulation language is Java since it supports Exception Handling and Multi-
threading. The pseudo-code for both the implementations is given in Figure 3.

(Basic implementation of Quicksort)
function qSort (q, low, high)
var array less, pivotList, greater
select a pivot value pivot from q
for each x in q
  if x < pivot then add x to less
  if x = pivot then add x to pivotList
  if x > pivot then add x to greater
return concatenate(qSort(less), pivotList, qSort(greater))

(Advanced implementation of Quicksort)
function Quicksort(q, low, high)
if (low < high)
pivot-location = Partition(q, low, high)
  Quicksort(q, low, pivot-location - 1)
  Quicksort(q, pivot-location+1, high)
else
  throw Exception;

Partition(q, low, high)
pivot = q[low]
leftend = low
for i = low+1 to high
  if (q[i] < pivot) then
    leftend = leftend+1
    swap(A[i], q[leftend])
    swap(q[low], q[leftend])
return (leftend)

Figure 3. Pseudocode for Simulation.

In both the cases, faults are injected into the variables (data) and the variables are monitored simultaneously. Table 1 gives the summary of the simulation results. Total failure denotes the number of times the program terminated abnormally and also at the early stage of the implementation, partial failure means that the faults introduced minor errors which do not halt the program immediately. In other cases, the program executes successfully, but the output may be correct or wrong. From the results it can be noted that the second implementation of the Quicksort performs better than the first implementation. That means that the second version of the Quicksort is more dependable with respect to fault-tolerant concern.

Table 1: Simulation Results.

<table>
<thead>
<tr>
<th>Description</th>
<th>Basic</th>
<th>Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of simulations</td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>

In an extended phase of simulation, faults have been injected into the entire execution time of two versions of Quicksort. We included an error-detection mechanism into this simulation model. An injected fault initially caused a minor error. If the minor error later propagated to and was detected by the model, it became a detected error. A fatal error occurred when a detected error disrupted control flow. The program would then either complete with correct or incorrect results or terminate through a time-out or fatal error. Figure 4(a) and Figure 4(b) report the average outcomes for both work loads. Of the 500 faults injected into each version, 66% was inserted into idle CPU and eventually overwritten. Of those, faults that were not overwritten, approximately 39% leaded to normal program completion, while over 61% produced fatal errors.

Next we measure fault and error latency in memory. We periodically copy the contents of real memory locations into archival storage. The locations were randomly chosen from 2 to 3 recursions in memory of up to 10 Kbytes each. Stuck-at bit faults were then simulated in the sampled variable contents to calculate latency distribution parameters for a representative set of 500 injected faults. In Table 2, the mean fault latency was almost four times greater for s-a-0 (stuck-at-0) than for s-a-1 faults. Conversely, the mean error latency of the s-a-1 faults was about double than that of the s-a-0 faults. Our programs use a fraction of their allocated memory blocks. This would leave many 0s in memory, because blocks are initially cleared when they are allocated. The frequency at which errors are overwritten before any additional errors accumulate can be determined from such measurements of fault and error latency.
We also simulated our operations 10 to 50 times, over periods of up to 50 hours to obtain statistically significant MTBF estimates. We considered two different error arrival rates ($\lambda_1 = l/24$ hours, $\lambda_2 = l/48$ hours) and latencies, based on the analysis of real error data collected from our system. The results graphed in Figure 5 show that injected errors alone did not adversely affect the system’s MTBF. However, when 75 percent of the injected errors were correlated by even a small percentage, the degradation in MTBF was enormous.

Figure 4(b). Category Distributions of Overwritten Errors

VI. CONCLUSIONS

This paper presents a study on fault-injection-based approach for dependability evaluation. Empirical validation was performed using simulation results obtained from two versions of Quicksort code for measuring the degree of fault-tolerance. One of the major problems yet to be resolved is the large fault space. Future investigations are underway in reducing the large fault space associated with highly integrated systems. This will require improved models that homogeneously symbolize the effects of low-level faults at higher rate.

Table 2: Simulation Results.

<table>
<thead>
<tr>
<th>Latency</th>
<th>Mean (Normal Executions)</th>
<th>Standard Deviation (Normal Executions)</th>
<th>Mean (Faulty Executions)</th>
<th>Standard Deviation (Faulty Executions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault</td>
<td>17.1</td>
<td>32.6</td>
<td>42.3</td>
<td>46.2</td>
</tr>
<tr>
<td>Error</td>
<td>42.3</td>
<td>46.2</td>
<td>22.1</td>
<td>46.7</td>
</tr>
<tr>
<td>Total</td>
<td>59.4</td>
<td>46.7</td>
<td>91.9</td>
<td>46.7</td>
</tr>
</tbody>
</table>

REFERENCES


Appendix

Figure 2: Anomaly detection: (a) Anomaly classes where $A$ is the set of anomalies internal to the code and $B$ is the set of anomalies outside the code and (b) Anomaly classes where $A$ and $B$ overlap [23].
Data Mining of Business Processes to Detect Software Errors

Alan Hevner and Balaji Padmanabhan
Information Systems and Decision Sciences
College of Business
University of South Florida
Tampa, FL 33620
{ahevner, bpadmana}@coba.usf.edu

Abstract

It is a challenge to detect software errors in operational systems when no obvious error conditions are evident. Business processes may perform poorly or incorrectly due to software system defects. For example, packages delivered to a certain zip code may be habitually delayed. However, linking these operational conditions to the responsible software defects may be very difficult. The focus of our research is to investigate the use of data mining to discover fault patterns in business processes and then to link these faults with their cause in a software module. We propose an approach that relies on leveraging faults in business process outcomes (e.g. a package arriving late), a mapping of how software modules impact specific business processes, and the use of data mining techniques to learn patterns to identify the potential sources of error.

1. Introduction

Highly automated industries, such as the automobile, shipping, and computer industries, provide numerous high profile examples of errors that are detected during deployment, sometimes triggering product recalls that are costly to both manufacturers and consumers. In a recent article, Garfinkel (2005) presents a list of ten of “history’s worst software bugs”, focusing specifically on errors detected after a system was deployed. One solution to this problem is to invest more in defect detection during systems development. However some defects may still pass through existing filters and the resulting system errors may not be detected until actual system operations. Hence it is critical to have a continuous software system assurance and testing approach that seeks to identify errors after deployment. In most cases, these are the software defects that are the hardest to find.

Even during deployment, some errors may be easier to detect than others. For instance, if a computer system crashes then evidence of an error is clearly established. However consider examples such as a vehicle routing system that provides a driver with a sub-optimal route, or a package tracking system that incorrectly identifies the location of a package. In such cases the actual software defect may remain “under the radar”, but can have significant consequences to both firms and consumers. How can such errors be detected?

In this paper we present a research proposal for addressing this problem using data mining. The approach is based on identifying measurable error patterns that occur during the performance of business processes (such as a package arriving late) and learning data mining rules to identify potential software modules that need to be further investigated for the responsible defects. The following example presents more specific motivation for the proposed solution.

Assume for simplicity a small package routing system, \( S \), that automatically places a package in one of two trucks based on the areas each truck is scheduled to visit. This system, \( S \), depends on two other systems, \( S_1 \) and \( S_2 \), where \( S_1 \) actually scans the package to determine an address and \( S_2 \) takes as input an address and returns a geographical coordinate and route to the address. On the business process side, assume that the specific business process of drivers delivering the packages is \( P_K \) and that the preceding processes in the supply chain are \( P_{K-1} \).

Suppose there is a defect in the software for \( S_2 \) in the processing of a specific zip code, say 90210, leading \( S_2 \) to provide an incorrect geographical coordinate, thereby placing the package in the wrong truck. Over time this may result in delays to some packages intended for 90210 (depending on how soon the driver detects the problem). Clearly there are no obvious errors here, such as a system crash. However there is a software defect that causes packages to be incorrectly loaded onto trucks. In such cases what we do observe are unacceptable process outcomes (delayed packages) associated with \( P_K \).
Without additional data, the unsatisfactory outcome at \( P_K \) may be attributed to a possibly wide range of reasons including human error or any of the prior processes or information systems that were used in the supply chain. However if each process step and each software system kept track of its actions in a data repository, then data mining approaches may be used in learning rules that might isolate the flaw(s). In this example, a rule of the form \( "S_2\text{.handles}(90210) \rightarrow P_K = \text{bad}" \) may directly help identify the source of the software defect in module \( S_2 \).

The subsequent sections provide a framework for performing this research. In Section 2 we discuss a formal representation for the relationship between software system modules and business processes and describe the data that need to be collected for the data mining framework to be effective. Section 3 discusses three pattern discovery techniques that can be applied to these data, specifically trying to learn rules relating to unsatisfactory process outcomes. Section 4 briefly discusses work in the related research areas. Section 5 concludes the paper with a discussion of research opportunities and challenges that our approach provides.

2. Graphs and relationships

2.1 Representing systems and processes

Visualize the system connection graph of technology modules to be a directed graph, \( S \), with \( k \) vertices \( S_1, S_2, \ldots, S_K \). Let \( M \) be the \( k \times k \) adjacency matrix for the graph where \( M(i, j) = 1 \) indicates that there is an edge from \( S_i \) to \( S_j \). The graph \( S \) represents technology modules and any relationships among them. We use the term "technology modules" to refer to entities that can be software and/or hardware. Figure 1 illustrates this graph.

For instance, continuing the example from the introduction, the automated sorting system is a vertex in this graph and has directed edges that come into it from vertices representing a package scanner and a GPS module. Modules that do not depend on any other module are represented as vertices with no incoming or outgoing edges.

Similarly, visualize the supply chain of business processes to be a directed graph, \( P \), with \( q \) vertices \( P_1, P_2, \ldots, P_Q \). Let \( R \) be the \( q \times q \) adjacency matrix for the graph where \( R(i, j) = 1 \) indicates that there is an edge from \( P_i \) to \( P_j \). The graph \( R \) represents business processes and any relationship between them.

Finally let \( SP \) be the graph that contains both the system as well as the business process graphs including relationships between system modules and processes. Hence, \( SP \) is a directed graph with \( k+q \) vertices with adjacency matrix \( T \). The indices (row and column) from 1 to \( k \) represent technology modules and the indices from \( k+1 \) through \( k+q \) represent business processes. Notice that this graph also represents interactions between modules and processes.

While Figure 1 only illustrates edges from systems to processes, this representation permits processes to in turn influence systems as well. For example, a process may provide information in the form of data inputs to a software module.

While business processes and relationships between software modules have been modeled as graphs in past research, linking these explicitly in the
context of software assurance and testing during deployment is novel. Further this representation also provides useful information to support testing in the operational environment. For instance, a process that was recently changed may attempt to use a specific system in unintended ways, leading to unexpected system faults. Likewise, changes to a software module may impact one or more processes in the supply chain, sometimes in unexpected ways. Such impacts can be identified based on the proposed graph.

We use this graphic representation of modules and processes to support a data mining approach for operational software systems assurance. The goal is to leverage system and process data to learn rules relating prior events to specific outcomes.

2.2 Required data

A prerequisite for effective data mining is the availability of quality data from which to observe patterns of behavior and to extract knowledge. Thus, it is essential that the modules and processes described above contain a historical record of their activities in a well understood data format.

As input to our research model we wish to observe the data that are automatically gathered, and mine that data for patterns that might indicate specific supply chain process failures. Of particular importance in the context of software testing are patterns that relate specific systems to unfavorable process outcomes. In this section we describe the data associated with the relationships shown in Figure 1.

Specifically, we associate a database with each vertex in SP. For instance, a vertex representing a sorting system is assumed to have a database that captures all the data that the sorting system processes. Time is an important attribute for software testing given that errors may be attributed to events that occurred earlier in time. For example, failures may occur due to a specific sequence of process and/or system activities. It would be essential to capture and understand that this specific sequence led to the failure and not simply one, isolated system action. Hence one or more relations in each database will have time as an attribute.

Certainly it is possible for some of these databases to be a single relation capturing process/system data. For expositional simplicity in the following discussion we will:

(i) assume a single relational table associated with each vertex, although the general ideas discussed here apply to databases associated with each vertex as well.

(ii) use the same notation to refer to both vertices and the associated relations. Specifically, we will use the term $S_k.x$ and $P_q.y$ to refer to attributes $x$ and $y$ in relations $S_k$ and $P_q$.

3. Pattern Discovery Techniques

Given the research framework defined in Section 2, we seek methods that can discover patterns of the form:

$S_k.zip = 90210, S_k.time = t \rightarrow P_q.outcome = delayed, t < P_q.time < t + \Delta, confidence = 20\%$

This rule specifically states that when software module $S_k$ processes a package addressed to zip=90210 it results, 20% of the time, in a delay in a subsequent process $P_q$ within a certain period of time, $\Delta$. If a shipping company’s goal is to have on-time deliveries as a rule, or even as a guarantee, then package delays 20% of the time to a certain zip code can be unacceptably high.

Current rule discovery methods in data mining do not discover temporal rules across multiple relations in this manner. However, association rules and sequential pattern discovery methods may be applied to learn such rules with some additional restrictions, notably one where a specific $\Delta$ can be chosen upfront, enabling the conversion of multiple relations into a single large flat file (typically the unit of analysis for rule discovery methods in data mining) such that the time stamps of records are within the given time period of the outcome measured. Below we briefly describe these types of patterns.

3.1 Itemsets in data mining

An itemset is defined as a set of items $\{i_1, i_2, \ldots, i_k\}$ that occur together in a single transaction. While the initial application for the itemset concept was market basket analysis, there have been other applications of this technique, such as learning the set of Web pages that are frequently accessed together during a single session.

An association rule is represented as $I_1 \rightarrow I_2$, where $I_1$ and $I_2$ are both itemsets and $I_1 \cap I_2 = \emptyset$. For rule $I_1 \rightarrow I_2$, support is the percentage of all transactions in the dataset that contain $\{I_1, I_2\}$. Confidence is defined based on a measure of conditional probability as the percentage of transactions where $I_1$ is present in which $I_2$ is also present. The “items” in itemsets are usually based on categorical attributes (although they have been used for continuous attributes based on discretization). An example of a pattern that can be learned from association rules is:
In this example we assume that the data have been preprocessed into a flat file such that all the attributes on the left hand side of the rule are constructed from prior time points. Note that the right hand side deals with a categorical condition (a “high” delay).

### 3.2 Quantitative rules

*Quantitative rules* extend the representation of typical association rules to one where the right hand side of the rule is a quantitative expression such as the mean or variance of a continuous attribute (Aumann and Lindell, 1999).


Given:

(i) sets of attributes $A$ and $B$,
(ii) a dataset $D$, and
(iii) a function $f$ that computes a desired statistic of interest on any subset of data from the $B$ attributes.

A SQ rule is defined as a rule of the form:

$$X \Rightarrow f(D_X) = \text{statistic}, \quad \text{support} = \text{sup}$$

Where:

(i) $X$ is an itemset (conjunction of conditions) involving attributes in $A$ only,
(ii) $D_X$ is the subset of $D$ satisfying $X$,
(iii) the function $f$ computes some statistic from the values of the $B$ attributes in the subset $D_X$, and
(iv) *support* is the percentage of transactions in $D$ satisfying $X$.

### 3.3 Sequential patterns

A third type of pattern that can be used in data mining for software assurance are *sequential patterns*. Srikant and Agrawal (1996) define a sequence as an ordered list of itemsets $<I_1, I_2, \ldots, I_k>$. The ordering is important and is used to represent a pattern where a series of itemsets follow one another (usually in time where transactions have time stamps associated with them). Such a representation is useful where patterns relating to the order of occurrences are relevant. An example of such an itemset in this context is:

$$S_k.zip = 90210, \quad P_q.outcome = \text{delayed}$$

However the key challenge that remains for all of these methods is determining the time window within which individual events are required to occur.
in the discovery of critical, hard-to-find software defects that directly impact the effectiveness of the business processes and quality of delivered products and services. The automated use of data mining for continuous quality assurance of essential supply chains has potentially broad impacts for many businesses.

4.2 Activity data repositories

Process monitoring is a component of Business Process Management, and one that is supported in traditional ERP systems (zur Muehlen and Rosemann 2000). Typically these systems store audit trail data about process execution including any relevant metadata associated with processes. McCoy (2002) describes a broader framework for Business Activity Monitoring (BAM) that pulls together process data across different information systems for operational as well as strategic goals. Hence, for organizations that use systems such as ERP systems, good quality process log data can be easily collected.

System event logs, on the other hand, exist for various software systems, but the data captured in these are system-specific. For instance, operating systems store logs related to various system events such as a failed device or a failed login attempt. However, there is little consistency among different software environments on the formats and contents of these event logs.

For this proposal traditional process monitoring logs (for business processes) and captured event logs (for systems) may provide data that can be directly used, but the specifics regarding the formats and the semantic contents of these data logs may vary with the organization and specific information systems used.

4.3 Process mining

Statistical Process Control (SPC) is a well-established statistical methodology (Oakland 1986) that studies process variations from a statistical perspective. The broad goal here is understanding process variations and their impact on some desired output. One or more variables may be monitored over time (in control charts) to determine if there may be a statistical change that might signify problems lurking ahead.

While similar in spirit to our motivation, this approach relies on identifying specific statistical criteria upfront that will be monitored over time. In contrast, methods that use data mining may suggest specific variables (or combinations thereof) that would need to be monitored. In this sense, our method can be viewed as a data mining approach for statistical process control.

There is a more closely related stream of work (Cook and Wolf, 1998, Datta, 1998, Aalst et al. 2004) in the area of business process mining that learns processes or process fragments from observed event logs. The typical context here is one where the underlying processes are not completely documented (but they do exist), however the analyst has access to logs that describe the set of events that have occurred with time stamps. From such data the process mining goals are to discover and better understand business processes.

A key difference in our context is the goal of learning rules relating events to software errors. Further our systems and process representations require that we have a full understanding of the graphs documenting systems, processes and relationships. However in the case where process graphs do not exist, our approach could first build on the process mining literature to learn the processes and relationships before applying our methods to the data.

5. Research directions and challenges

The goal of this paper is to set forth our research vision of how the use of data mining could provide continuous quality assurance of business supply chains and their associated software systems. The identification of patterns leading to unsatisfactory outcomes will support the discovery of subtle but critical software defects. However, the realization of this vision will demand research advances on a number of key challenges. Just as importantly, this research must be performed in the context of real-world systems. Thus, forming an industrial partnership to conduct this research is an essential component of our future directions. In this concluding section we identify some of the specific challenges and corresponding research directions that we will investigate.

First and foremost, high quality data are essential for data mining to be useful. Quality activity data repositories on software modules as well as business processes are not easy to come by, but successful organizations are increasingly keeping track of such data. What types of data are needed, in what formats, and at what level of granularity? These are hard questions to answer and may depend on the kinds of errors that cause processes to malfunction. While the problem appears to be domain and application specific, developing some guidelines mapping error types to the kinds of data needed are interesting research directions to pursue.
Second, good data mining algorithms are also critical. The methods described in Section 3 are well-known rule discovery techniques, but these still have the drawbacks of having to use pre-defined temporal effect windows and flat file inputs. A related direction for research is to develop new pattern discovery methods that are optimized for learning temporal rules such as the ones needed for continuous quality assurance data mining. Two specific extensions that would be valuable are:

1. **Learning temporal windows automatically instead of using pre-defined defaults.** The approaches described in Section 3 will create records containing variables relating systems and process data. However in the preprocessing stage how far back in time should variables be created to model specific process failures? Methods that can determine this automatically will clearly be useful.

2. **Learning rules across relations.** Classical rule discovery approaches in data mining work on flat files. However the data are likely to be in relational tables associated with each node in the graph. While there has been work on inductive logic programming (Muggleton, 1991) that addresses the issue of learning concepts from relational data, this has been limited to small scale applications and has not been used to learn temporal rules. Can other heuristic methods, such as genetic algorithms, be used effectively for this problem?

A third challenge is how to assign ‘blame’ for a poor process outcome. It may be possible to learn several rules that relate some variables to unfavorable process outcomes. These rules may suggest that there is not just one, but several prior systems and processes that each or collectively may have contributed to the outcome. In such cases, how can the exact cause be effectively identified? One solution here is to support a ‘human-in-the-loop’ interface to let domain experts analyze all discovered rules. But depending on the scale of the problem this may or not be feasible. Can good methods for effectively “assigning the blame” be developed?

As a final research challenge, we must find a viable and supportive environment in which to perform this research. To make this research relevant to real-world systems, we are looking to partner with one or more companies who face the real issues of business process quality every day. Working together, we look forward to investigating and finding solutions to these important research challenges.

---

**References**


Predicting Defects in Large Open Source Software

Teodora R. Stoian, Vasile Rus, Sajjan S. Shiva
Department of Computer Science
The University of Memphis
Memphis, TN
{rtstoian, vrus, sshiva} @ memphis.edu

Abstract
Defect prediction is an important topic in software quality processes. We present in this paper our research on defect prediction based on numerical models. Numerical models predict how many defects are in software units as opposed to just predicting the presence or absence of defects performed in binary model. The proposed models were evaluated using experimental data collected from approximately 3 million lines in 3 Mozilla’s Firefox releases. To generate the experimental data, we collected problem reports (PRs) from Bugzilla, Firefox’ defect tracking system, and linked the PRs to modification reports (MRs) obtained from Mozilla’s Concurrent Versioning System (CVS). Various linear and non-linear machine learning algorithms were then used to induce predictors based on size and complexity metrics. The results obtained with the various algorithms are discussed and compared.

1. Introduction

We present in this paper research on defect predictors in large open source software projects such as Mozilla’s Firefox web browser. Defect predictors help project managers identify units of software products that are likely to contain defects. The software development process could become more efficient as the development and testing effort would focus on the identified buggy software units. In this context, the objective of a defect prediction model is to predict the likelihood of present or future defects based on defect information from the past.

An effective defect prediction model could make the software quality task more efficient through defect prevention and removal. It could lead to reduced development cycle time, reduced product costs, and improved product quality. In addition defect prediction can help identify risk prone software units and indicate software problems at earlier stages of the development. The need for accreditation can also be an important factor. Compliance with various maturity models and standards requires the implementation of defect prediction models. Both Software Process Improvement and Capability Determination (SPICE), also known as ISO 15504, and the Capability Maturity Model (CMM), from Software Engineering Institute (SEI), specify the need for a predictable process.

In general, building defect prediction models involves the following steps: (1) modeling the defect prediction problem, i.e. what to predict and in what context (2) selecting and collecting features/attributes as components of the model, i.e. what are the characteristics based on which the predictions are made (3) collect data, analyze it, and refine the model(s) based on the analysis (4) use the model(s) with machine learning algorithms to automatically learn the parameters of the model(s) from past data. The end result of the previous steps are software tools that could be applied to new data, e.g. future releases of a software product, to predict defects in these releases.

The rest of the paper is organized as in the followings. Section 2 presents related work. Section 3 presents the selection of features, i.e. size and complexity metrics that we use to predict our models. Section 4 presents data preprocessing. In section 5 statistical, neuronal networks, probabilistic, and decision tree algorithms were selected and analyzed. Section 6 describes the defect prediction model assessment. The last part of the paper, sections 7 presents the conclusion and directions for future research.

2. Related Work

Many modeling methods have been proposed and used, however, previous research studies have provided little guidance on how to capitalize the results of different approaches. We discuss the previous work along three dimensions that correspond to the major components of any defect prediction framework: feature selection, data collection, and algorithm for predictor induction.
2.1. Feature selection

The first dimension refers to the set of features used to predict software defects. A wide range of features have been proposed in particular for industry-developed software: product metrics, development metrics, deployment and usage metrics, software and hardware configuration metrics (1). Product metrics (e.g. lines of code) and process metrics (e.g. cost and time for completing a task) have been extensively used in an attempt to predict the number of defects in a software system. Among the product metrics, probably the most controversial ones are software size and complexity metrics, explained in detail in Section 4. Being the oldest metrics and the simplest to obtain, they are widely used both in industry and academia to predict defect information (2), (3).

There is still a lot of debate about the relationship between various metrics of code size and complexity on one hand and defect occurrences on the other. Fenton and Neil found that these metrics are good enough predictors for the absolute number of defects but very poor predictors of defect density (4). They concluded that complexity and size metrics alone cannot provide accurate predictions of software defects.

On the other hand, many researchers have found positive correlations between size and defects (3), (5). Graves, Karr, Marron and Siy (6) as well as Ostrand, Weyuker and Bell (7) argue that the defect occurrence is strongly correlated with the size of the file. Hatton approached the defect prediction problem from a cognitive science perspective, arguing that the components that fit comfortably into human short-term memory are most likely to have the lowest defect densities and that components should be neither too small nor too large (i.e. optimal component size is around 200-400 lines of code) (8). Among the most articulate defenders of size and complexity metrics, Nagappan and Ball (2) carried out the largest prediction study on commercial systems. They show that product metrics as well as code churn measures (i.e. the amount of software change taking place within a software unit over time) can be used to identify failure prone modules in Microsoft products (2).

2.2. Data collection

The second dimension is data collection. Data from software repositories is necessary to induce the parameters of the various models for defect prediction. Fenton considers the poor quality of the data as a major cause of inaccurate prediction systems (4). As we discuss later in the paper, data collection has been a real challenge for us too. There are two major types of data used to predict software defects. One type implies data extracted from early versions of a software project which are used to predict defects for future releases (9), (10), (11). Our approach fits within this type. The other type takes into account the actual differences between software versions. In general, the approaches using the second data type are about models for change prediction. The main goal in these approaches is the study of software evolution rather than serve in quality assurance and software testing tasks.

2.3. Algorithms for Predictor Induction

The third dimension of defect prediction modeling refers to the specific algorithms used to implement and estimate the parameters of the predictor models. Among previously proposed algorithms we mention the following: logistic regression used in Zimmermann et al. (10), moving averages used in Li et al. (12), principal component analysis and clustering used in Khoshgoftaar et al. (13), decision trees used in Knab et al. (11), (14), and neural networks used by Khoshgoftaar (14). We discuss below regression and decision trees applied to OSS as being the most relevant to our work.

Knab, Pinzger, Abraham (11) approach applies decision tree learners to source code metrics, modification reports metrics and defect data in seven recent source code releases of Mozilla’s content and layout modules. Their work provides support for Fenton’s claim that size metrics such as number of functions are of little value for predicting defect density. Knab et al. contingent argument is counterbalanced by Zimmermann, Premra and Zeller (10). They mapped Eclipse classes to the number of defects that were reported in the first six months before and after release. Using logistic regression they endorsed that the combination of complexity metrics can predict defects. They suggest that the more complex the code is the more defects it has. A more extensive discussion about the relationship between code size and number of defects can be found in Li, Herbsleb and Shaw (9). They evaluate metrics collected for OpenBSD using established statistical methods: Kendall’s rank correlation, Pearson’s rank correlation, and Akaike’s Information Criterion (AIC) model selection. They claim that the number of messages to the technical discussion mailing list during the development period is the best predictor of defects. Other important predictors they have selected include product metrics collected from the CVS code repository such as the number of inline
comments, the total number of methods, the number of public methods, and the number of protected methods.

3. The problem

We address in this paper the problem of predicting software defects. The premise underlying a defect prediction study is that defect proneness of a file $F_1$ from one (or more) previous release(s) $R_i$, $i = 1...n-1$, with a set of feature-value pairs $X$ is similar with the defect proneness of file $F_2$ from next release $R_n$ provided $F_2$ can be characterized by a similar set of feature-value pairs. The set of feature-value pairs $X$ and the defect proneness of files from previous release(s) can be obtained by exploring information in software repositories. Thus, the unknown variable is the defect proneness of files from release $R_n$.

There are several issues related to the problem of defect prediction that must be addressed. First, we must specify the level of granularity or unit at which we predict defects. The unit could be function, file, module, or even functionality-based prediction, that is, we would predict if the source code related to the implementation of a particular functionality has defects or not. Second, we need to indicate the particular milestones in the lifecycle of a software product that are considered in the prediction. For example, the source files at major or minor release time are inspected or it could be every two weeks or at every recompilation time. Another important issue is the type of prediction: binary, numerical, or structured. A binary prediction type would simply predict if a source code file contains defects or not while a numerical type would predict how many defects there may be in a file, e.g., predicting that file $F$ may contain 7 defects. Furthermore, a more sophisticated prediction type could predict how many defects of a certain severity are in a file, e.g., predicting that file $F$ contains 7 critical defects and 3 minor defects.

In this paper, we build numerical predictors that work at source code file level (our unit of analysis) and use information about these files from major and minor releases. In particular, we use complexity and size metrics to characterize source code files and thus two files are similar if their complexity and size characteristics are similar.

4. The model

To the best of our knowledge, there is still a dearth of empirical evidence linking defect data to size and complexity metrics in OSS projects. Defect prediction can be modeled in various ways. In our approach, we use complexity and size metrics to characterize source files and to predict defects. The selected metrics are part of the larger category of product metrics.

Product metrics describe attributes of the products of the software development process. They could include, for example, the traceability of the design, the reliability and maintainability of the code, and the coverage of the tests. Due to the peculiarities of OSS development we used only code metrics, defined as all those metrics that measure the actual code produced in the development phase of a software process. We collected both control flow metrics and token based metrics. Control flow metrics explore each program unit in a software system and determine the flow of control structure complexity. Token Metrics are calculated by counting tokens in the source code of program units and then taking into account size as a source of variation. We ended up with a set of 32 metrics that we use in our modeling (see the Data Collection section for further details). Table 1 shows the most important metrics (the full list is not provided for space reasons). Complete descriptions of the collected metrics are available from SciTools (15).

<table>
<thead>
<tr>
<th>Category</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Flow Metrics</td>
<td>Cyclomatic Complexity</td>
</tr>
<tr>
<td></td>
<td>Essential Complexity</td>
</tr>
<tr>
<td>Token Metrics</td>
<td>Lines of Code</td>
</tr>
<tr>
<td></td>
<td>Lines of Comments</td>
</tr>
<tr>
<td></td>
<td>Inactive Lines</td>
</tr>
<tr>
<td></td>
<td>Blank Lines</td>
</tr>
<tr>
<td></td>
<td>Declarative Lines</td>
</tr>
<tr>
<td></td>
<td>Executable Lines</td>
</tr>
<tr>
<td></td>
<td>Statement Counts</td>
</tr>
<tr>
<td></td>
<td>Function Counts</td>
</tr>
</tbody>
</table>

Table 1 – Product Metrics

5. Data Collection

We chose the Mozilla Firefox project for building defect predictors. Firefox is an Open Source Software (OSS) web browser and the main product developed by Mozilla. As a non critical software system, it is widely recognized that Firefox contains post release defects.

OSS facilitates the collection of data to be used in defect prediction models. According to Bollinger et al (16), an important requirement for OSS code is that it should be rigorously modular, self-contained
and self explanatory, to allow development at remote sites. Therefore, the data that can be used for prediction models in OSS could be retrieved from the source code version (CVS) repositories and bug tracking systems. On the other hand, OSS development is characterized by lack of a formal process (17), poor design and architecture (18), and development tools that are not comparable to those used in commercial development. Few of the defect prediction approaches in commercial software can be directly applied to OSS development, however results obtained from OSS prediction models can be used in an industry environment.

In order to collect the data necessary to build a defect predictor for Firefox, we must first understand the release procedures, the source code management system, and the defect tracking system.

5.1. Version Selection

Firefox is based on independent Mozilla Core components layered together. The Mozilla source code is organized in several branches. The trunk is the main branch, the central source code that is used for continuous and ongoing development. Trunk builds contain the very latest changes and updates. However, the trunk can also be very unstable at times. When development is started for a specific Mozilla version a new branch is created. At conception, a derived branch contains everything that the principal branch contains (Figure 1). The term release is used in OSS development to refer to different types of releases: major and minor, alpha and beta (15). Firefox branches that correspond to major releases are forked from the associated Mozilla branch and will be used for all future minor releases of Firefox. Specifically, Firefox 1.0 branch was derived from Mozilla Branch 1.7 while Firefox 1.5 from Mozilla Branch 1.8. Due to data availability constraints, we have only considered two major releases, 1.0 and 1.5, and a minor release, 1.5.0.3.

Firebox Branch 1.5.0.3 resynchronized the code base with the trunk, which contained additional features not available in Firefox 1.0. On the other hand, in release 1.5.0.3 the focus was not on adding features but on improving security related aspects, which were bypassed in version 1.5.0. This mix of minor and major releases allowed us to test variation in the performance of defect prediction models when trained on data collected from major or minor releases.
releases. The time frame between the considered releases was approximately equal (see Table 2), which eliminates time as a possible confound.

<table>
<thead>
<tr>
<th>Version</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox 1.0</td>
<td>12/10/2004</td>
</tr>
<tr>
<td>Firefox 1.5 beta 1</td>
<td>09/09/2005</td>
</tr>
<tr>
<td>Firefox 1.5.0.3</td>
<td>05/08/2006</td>
</tr>
</tbody>
</table>

Table 2 – Selected Versions

5.2. Module Selection

The reason behind branching is that components that need to be prepared for a future release are at the same time continuously developed on the trunk. A distinction needs to be made between Firefox-specific source code, i.e. code that does not support any other Mozilla application, and the Mozilla components that support Firefox.

We have selected for our experiments seven Mozilla Core components that are used by Firefox, namely: DOM, Expat, HTML Parser, JavaScript Engine, Network, Security and Layout (19). We have limited the analysis to those components because they have formed the core software that is part of every Firefox installation, even though the browser changes from release to release as specifications are added and removed.

5.3. Metrics Collection

The seven modules we have selected for analyzing are 55% written in C/C++ and 28% in JavaScript. They also include HTML and Java (15%), and other languages such as XML, CSS, Perl, Python, PHP, or even LISP in much smaller percentage. Since the dominant language is C/C++, we have selected only *.c and *.cpp files as follows: 1204 from release 1.0 and 1267 from releases 1.5 and 1.5.0.3 with a total of 874,277, 1,012,931, and 1,019,367 lines of code, respectively (see Table 3). We did not consider the header files (.h files) as they are connected with the corresponding .c or .cpp files.

To derive the product metrics for each source file we used the Understand C++ (15) analysis tool. The tool computes the source code metrics for C and C++ programs and generates report with the metrics. The report contains three categories of metrics: project level, file level, and function level. It also contains object oriented metrics for the .cpp files. Because these metrics are not applicable to .c files we excluded them from our experiments for consistency across all files. The set contained 32 file level metrics.

<table>
<thead>
<tr>
<th>Firefox</th>
<th>V1.0</th>
<th>V1.5</th>
<th>V1.5.0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>23,740</td>
<td>26,704</td>
<td>27,309</td>
</tr>
<tr>
<td>EXPAT</td>
<td>10,746</td>
<td>11,179</td>
<td>11,179</td>
</tr>
<tr>
<td>HTML Parser</td>
<td>25,005</td>
<td>24,611</td>
<td>24,619</td>
</tr>
<tr>
<td>JS Engine</td>
<td>132,321</td>
<td>149,466</td>
<td>151,204</td>
</tr>
<tr>
<td>Layout</td>
<td>206,228</td>
<td>274,890</td>
<td>276,281</td>
</tr>
<tr>
<td>Network</td>
<td>81,969</td>
<td>85,494</td>
<td>85,766</td>
</tr>
<tr>
<td>Security</td>
<td>330,002</td>
<td>377,680</td>
<td>379,668</td>
</tr>
<tr>
<td>Total</td>
<td>874,277</td>
<td>1,012,931</td>
<td>1,019,367</td>
</tr>
</tbody>
</table>

Table 3 - Lines of code for .cpp and .c files

5.4. Bug Collection

The Mozilla project relies on Bugzilla, a defect tracking system, to monitor problem reports (PR), i.e. bugs. A PR in Bugzilla has several pre-defined attributes. Some fields, such as the PR identification number and creation timestamp, are created when the report is first filed. Other fields, such as the product, component, and severity, are selected by the testers when the report is filed and may be changed over the lifetime of the report. Other fields routinely change over time, such as the current status of the report, and if resolved, its resolution state.

Studying the lifecycle of a bug facilitated linking the Bugzilla PRs and CVS Modification Reports (MRs). The status and resolution fields define bugs as evolving entities that change over time. When a tester enters a new bug in Bugzilla the status of the bug is set to UNCONFIRMED. The Mozilla quality assurance team will look at it and confirm the bug exists and changing its status to NEW. After a developer looks at the bug and either accepts it or assigns it to someone else, the bug’s status becomes ASSIGNED. Once the bug is fixed, its status changes to RESOLVED. Finally, the quality assurance team verifies that the bug was indeed fixed and the status is set to VERIFIED and then CLOSED. If the quality assurance team is not satisfied with the solution, than the bug is REOPENED and the process starts again.

A report can be RESOLVED in various ways. Bugzilla PRs indicate this in the resolution field. If the bug was solved and this resulted in a change to the code base, the bug is resolved as FIXED. When a developer determines that the bug is a duplicate of an existing report then it is marked as DUPLICATE. If the developer is unable to reproduce the defect, then the resolution is set to WORKSFORME. If the report describes a problem that will not be fixed, i.e. it is not an actual bug, the report is marked as WONTFIX or INVALID.

In Bugzilla terminology, a bug can be anything that needs to be tracked (20). Some entries are not real bugs, i.e. defects, but rather enhancements. When
analyzing a report in Bugzilla, the quality assurance team rates severity of the bug using one of the following labels: blocker, critical, major, normal, minor, trivial, or enhancement.

While Bugzilla contains information about defects, it does not contain information about the location of the defects in the source code. Instead, this information is captured in the CVS log files. CVS Modification Reports (MRs) keep the complete history of any file in the project, including when and what was modified. Bonsai (21), Mozilla’s web interface to its CVS repository, can be used to retrieve MRs related to source files, comments associated with the files, and the timestamp of the commit message. Each comment acknowledges the people who submitted the change and contains relevant PR identifications numbers (if any). Every number that appeared in a MR’s comment field was a potential link to a bug, indicating that that commit message solved a PR. We selected the number as a candidate for a bug id if the following two conditions were met: the number had the length less than 6 digits and the comment message contained the keywords bug, bug id, id or # before the number.

Table 4 below summarizes the output of the data collection phase. The output consists of three files containing the PRs, the MRs, and the values for the size and complexity metrics for each source code file considered. The first two lines in the table explain the source of the data and the tools used in the data collection, e.g. for collecting PRs and MRs we developed small Perl scripts. The last line presents the source code for the selected Firefox modules and versions used to gather the data. For instance, we collected the PRs from the Mozilla Trunk source code, the MRs from the Mozilla Branch source code, and the metrics from the Firefox Branch source code. At the end of the data collection phase, the PRs file contained: the identification number, the status, the resolution state, and the creation and modification timestamps for each PR. The MRs file contained the PR identification number and the corresponding source code file name. The metrics file contained the source code file name and the 32 metrics associated with it.

<table>
<thead>
<tr>
<th>Files</th>
<th>PRs</th>
<th>MRs</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Bugzilla</td>
<td>CVS Log Files</td>
<td>CVS Source Code</td>
</tr>
<tr>
<td>Tools</td>
<td>Perl Scripts</td>
<td>Perl Scripts</td>
<td>Understand C++</td>
</tr>
</tbody>
</table>

### 6. Data Integration and Analysis

In this section we discuss the main issues we encountered when preprocessing data. We address the issues of data integration and data reduction and transformation. For data integration we developed PERL scripts while for data reduction and transformation we used WEKA data mining toolkit (22).

#### 6.1. Data integration

The main problem with using software repositories in defect prediction is the lack of integration of the CVS history files and defect tracking systems. We linked the PRs with MRs using the PR identification number available both in the MRs in CVS and in the PRs in Bugzilla. A real challenge was to associate the bug reports in Bugzilla with the specific Firefox releases. The data collection process took place at moment $t_3$ and the goal was to collect bugs that were in the source code the moment of the release, $t_1$ (Figure 2). This is not trivial as the following example illustrates. Suppose at the time of the release $t_1$ a defect was in the source code. If the defect was solved after the release, say at $t_2$ or $t'$, the bug at $t_3$, when we collected the data, is labeled as being resolved.

We approximated the defects that were in the source code at the time of a Firefox release $t_1$, to be all the bugs with creation timestamp before $t_1$ AND:
1. with the status CLOSED, RESOLVED or VERIFIED after $t_1$ OR
2. with the status NEW or ASSIGNED and with the committed to CVS timestamp after $t_1$ OR
3. with the status NEW, ASSIGNED or REOPENED at $t_3$. 

### Figure 2 - Linking MRs to PRs

<table>
<thead>
<tr>
<th>Bugzilla</th>
<th>CVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$ = Bug Creation Bugzilla Timestamp</td>
<td></td>
</tr>
<tr>
<td>$t_1$ = Firefox Release Timestamp</td>
<td></td>
</tr>
<tr>
<td>$t_2$ = Bug Committed to CVS Timestamp</td>
<td></td>
</tr>
<tr>
<td>$t'$ = Bug Resolved Bugzilla Timestamp</td>
<td></td>
</tr>
<tr>
<td>$t_3$ = Data Collection Timestamp</td>
<td></td>
</tr>
</tbody>
</table>

84
We considered both case 1 and case 2 above because we are dealing with an open source environment. It may happen that a bug was solved, the commit message exists in the CVS history file at \( t_2 \) but the bug status was not modified in Bugzilla records (case 2 above). It may also be the case that the commit message in CVS is not reflecting the change performed, it does not have a PR identification number associated with it, even if the change resolves a problem and it is reported in Bugzilla at \( t_1 \) (case 1 above). We also selected only the PRs with the severity marked as blocker, critical, major, normal, minor, and where applicable with the resolution set to FIXED. The problem with our approach is that there may be defects in the code undiscovered at moment \( t_1 \) and that will be reported after the release. Because there is no way to tell to which release the bugs belong we simply did not consider them.

The output file of the integration procedure contains the source file names and number of bugs for each file. Finally, the output file is merged with the complexity metrics file. The result is a file containing instances for each source code file. Each instance contains the source code file name, the corresponding values for the complexity metrics, and the number of defects in the source code file.

### 6.2. Data reduction

One major issue with modeling a problem with a large set of features is the high dimensionality of the resulting representation. The dimensionality can be reduced by exploiting correlations among the selected attributes. Two attributes that are correlated are basically redundant and keeping both does not lead to better performance.

Principal component analysis (PCA) is a statistical approach that generates new attributes, by considering the weighted sums of the original attributes. Basically, PCA reduces the dimensionality of datasets by eliminating the correlations among some of the considered attributes. The new attributes are orthogonal to each other. Generally, a small number of principal components can capture the effect of the larger initial set of attributes. We applied PCA to our set of 32 attributes. A new dataset was obtained with 10 data vectors as attributes, among which the first 5 accounted for 85\% of the data variation. Each dimension was a linear combination of some of the 32 metrics.

### 6.3. Data transformation

The datasets we obtained during data collection have an unbalanced distribution. (Table 5) Often, it is useful to compare various methods/algorithms on balanced data sets in order to get a fairer picture of the algorithms’ strengths. We thus derived from the original data set a new balanced data set. We used a reduced sample of the dominant class (defectieveless) class, the most represented category in the class attribute. However, undersampling can be a bad idea, since it is important to keep the conditions as close as possible to reality. So we undersampled only for the training set, doing the evaluation of the learned model on the original unbalanced data.

<table>
<thead>
<tr>
<th>Firefox Version</th>
<th>% Defective Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>26%</td>
</tr>
<tr>
<td>1.5</td>
<td>16%</td>
</tr>
<tr>
<td>1.5.0.3</td>
<td>15%</td>
</tr>
</tbody>
</table>

Table 5 – Percent defective files in the total number of files

In addition, a discretization step was necessary to further prepare the data for analysis because of a distinct category of learning algorithms that can only handle discrete attributes. We transformed the number of continuous attributes values by dividing the range of the attributes into equal intervals. We considered both equal-width and equal-depth partitioning. Equal width partitioning with an unbalanced data set may lead to an interval distribution where outliers may dominate. To avoid this, we chose frequency partitioning and divided the data into intervals, each containing approximately same number of files.

Finally, we discretized the class attribute. We partitioned the defect data into three intervals: files with no defects, files with at most two defects, and files with three or more defects. For the rest of this paper we refer to files with more defects as files containing 3 or more defects, while files with fewer defects are defined as files containing one or two defects.

### 7. Algorithm Selection and Experimental Setup

We compared various algorithms for deriving the parameters of the proposed model for predicting defects. The ones listed below provided the best results and the discussion will focus on them:

- Linear Regression (LR) also used by Bernstein, et al in (23)
- Model Tree Learners, such as M5P also used by Bernstein et al in (23)
- Random Forests (RF) also used by Jiang et al in (24)

We used implementations of the algorithms from the WEKA data mining toolkit (22) and conducted the experiments summarized in Table 6.

<table>
<thead>
<tr>
<th>Exp</th>
<th>Training Data</th>
<th>Testing Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Firefox 1.0</td>
<td>Firefox 1.5</td>
</tr>
<tr>
<td>2</td>
<td>Firefox 1.5</td>
<td>Firefox 1.5.0.3</td>
</tr>
<tr>
<td>3</td>
<td>Firefox 1.0 &amp; 1.5</td>
<td>Firefox 1.5.0.3</td>
</tr>
</tbody>
</table>

Table 6 – Training and Testing Data

Two experimental setups were designed. In one setup, we predict the actual number of defects in a file (experiment A) while in another setup we predict the class or level of defectiveness of a file: no defects/fewer defects/more defects (experiment B).

8. Results

As discussed previously, there is a lot of debate with respect to whether size and complexity can predict defects. We argue that there is value in size and complexity metrics with respect to defect prediction and that research should rather focus on to what extent can size and complexity predict defects or in what particular cases we can predict defects based on size and complexity metrics. In this context, we present our interpretation of the results.

The results of the algorithms using continuous values of defects were reported using correlation coefficient (R), mean absolute error (MAE) and root mean squared error (RMSE):

- \( R \) ranges from -1 to 1 and measures the statistical correlation between the predicted and actual values. A higher number means a better model, a value close to 0 meaning there is no correlation at all and a value of -1 pointing to an inverse correlation.

- \( MAE \) is the average between the difference of the predicted valued and the actual values in all test cases; this is the average prediction error. It does not have a fixed range, but is geared to the values to be predicted.

- \( RMSE \) is the sum of the squared differences between predicted values and the corresponding actual values. The range of the mean squared error is geared to the ranges of predicted values, similar to the mean absolute error.

<table>
<thead>
<tr>
<th>Exp</th>
<th>TP</th>
<th>FP</th>
<th>Pr</th>
<th>Re</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 ( \rightarrow ) 1.5</td>
<td>0.95</td>
<td>0.44</td>
<td>0.69</td>
<td>0.95</td>
</tr>
<tr>
<td>1.5 ( \rightarrow ) 1.5.0.3</td>
<td>0.40</td>
<td>0.06</td>
<td>0.72</td>
<td>0.40</td>
</tr>
<tr>
<td>1.0+1.5 ( \rightarrow ) 1.5.0.3</td>
<td>0.61</td>
<td>0.02</td>
<td>0.89</td>
<td>0.61</td>
</tr>
</tbody>
</table>

Table 7 shows the results of Experiment B (discrete data) using the RF algorithm. The RF classifier was trained to predict classes with no defects (top row in each sub-experiment, which is indicated in the leftmost column), classes with fewer defects (middle row corresponding to each sub-experiment), and classes with more defects (bottom row in each sub-experiment).

1. Files with more defects can be more accurately predicted than files with fewer defects.

It is noted from Table 7 that files with more defects (bottom row in each sub-experiment) can be predicted with a higher accuracy. For instance, in sub-experiment 1 [Exp(1) in Table 7] we obtained \( \text{precision}=0.89 \) and \( \text{recall}=0.61 \) for files with more defects as compared to \( \text{precision}=0.72 \) and \( \text{recall}=0.40 \) for files with fewer defects.

2. Non-linear prediction models are superior to linear ones.

For the experiment A (continuous data), we ran LR and M5P algorithms (Table 8). The correlation coefficient obtained was encouraging for the M5P decision tree learner. The value of 0.66 shows an existing correlation between number of defects on one hand and size and complexity metrics on the other hand. The MAE and RMSE are relative to the maximum number of defects in a file (15 defects in our case).
Previous research results suggest that non-linear defect predictors are superior to linear ones (25). We tested this hypothesis by considering both linear classification algorithms and non-linear ones. Our results too show that M5P outperforms the LR, i.e., non-linear predictors perform better than linear ones, supporting the results in (25).

Table 8 – Experiment 1 results: the LR and M5P algorithms

<table>
<thead>
<tr>
<th></th>
<th>Experiment</th>
<th>R</th>
<th>MAE</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>(1) → 1.5</td>
<td>0.58</td>
<td>0.98</td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td>(2) 1.5 → 1.5.3</td>
<td>0.58</td>
<td>0.97</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td>(3) 1.0 + 1.5 → 1.50. 3</td>
<td>0.58</td>
<td>0.99</td>
<td>1.56</td>
</tr>
<tr>
<td>M5P</td>
<td>(1) 1 → 1.5</td>
<td>0.56</td>
<td>1.00</td>
<td>1.56</td>
</tr>
<tr>
<td></td>
<td>(2) 1.5 → 1.5.3</td>
<td>0.64</td>
<td>0.88</td>
<td>1.48</td>
</tr>
<tr>
<td></td>
<td>(3) 1.0 + 1.5 → 1.50. 3</td>
<td>0.66</td>
<td>0.88</td>
<td>1.47</td>
</tr>
</tbody>
</table>

9. Conclusions

While we do not wish to draw strong conclusions from a single study, we believe that our results suggest that size and complexity metrics can be attractive features to be used for defect prediction, due to their relative simplicity.

Our future work will concentrate on improving and automating the data collection phase because it is one of the most expensive tasks. Also, we plan to investigate applying quantitative techniques from the social sciences (risk theory, utility theory, and decision theory) to defect prediction. Predicting errors introduced in products developed by large populations of software experts should involve consideration of the social aspects of software.

10. References


http://www.bugzilla.org/docs/2.18/html/lifecycle.htm l.


Abstract: Software engineering process incorporates different theoretical and applicable computer science domains (i.e. theoretical computer science, graph modeling etc.) to assure appropriate behavior of the developed software. Software testing is particularly devoted to find defects in a software system. Inadequate test plans and procedures are often positively correlated with time delays, cost overruns and security vulnerabilities. To avoid time delay and cost overruns, significant manpower and efforts are often employed into software testing. However, in cases of over-budget and behind schedule scenarios, software testing efforts are often cut short resulting in software security vulnerabilities. Hence, the modular automated software testing system that can detect common programming errors and security bugs, which are otherwise expensive to find, will translate to optimized and safe code.

This paper reviews the challenges in using machine learning techniques for identifying software defects. The foci of the paper is to present a comprehensive analysis and review on approaches suitable for large scale software testing and will facilitate a modular and highly scalable architecture that may be used across a variety of software products with minimal changes to the testing package. While reviewing past achievements, this paper also outlines the implementation strategies currently under investigation.

1. Introduction

The nature of exponential growth in large-scale software and their usage in science and society is in the midst of remarkable change - change that is stimulated by a combination of new scientific and societal needs for understanding and developing complex systems and rapidly evolving hardware technologies that needs to be programmed in innovative ways. Modern life is ever more dependent on software. But even after couple of decades of successful software development journey, software bugs are still the sources of major problems. Earlier studies show that testing can take up to 50% or even more of the development cost and efforts [1]. However, in cases of over-budget and behind schedule scenarios, software testing efforts are often cut short resulting in software security vulnerabilities. Also, so far there had been not a very efficient and automatic way of testing software against vulnerabilities and security threats. Despite tremendous efforts over last two decades, the solution remains illusive. One of the probable cause is the versatility of programming aspects (i.e. programming models, languages, scripts). A number of tools have been developed which are very domain specific (i.e.; MOPS for C codes [2], DIDUCE for Java codes [3]). Identifying and locating defects in general purpose large-scale software projects is cumbersome and can quickly get out of control with growing project sizes as the task becomes expensive with sophisticated testing and evaluation mechanisms. On the other hand, measuring software in a continuous and disciplined manner provides many advantages such as accurate estimation of project costs and schedules, and improving product and process qualities. Detailed analysis of software metric data also gives significant clues about the locations of possible defects in a programming code.

To meet the growing demand, the developers moved a lot towards reusable object oriented coding instead of classical structural coding scheme, and widely accessible web applications development. With object-oriented approaches, testing should center on objects, classes, generic classes and super-classes, instead of subprograms. Moreover, in the presence of encapsulation, the only way to observe the state of an object is to call a method; thus there is a fundamental problem of observability when testing within an object-oriented framework. Web applications are ubiquitous and
increasingly more sophisticated, and so they are getting popular for almost all major online businesses. As with most security issues involving client/server communications, web application vulnerabilities generally suffer from improper handling of client requests and/or a lack of input validation checking on the part of the developer.

So, with its growth in complexity and size, software testing is no longer a mere process. Software testing is not only about finding defects or bugs in the software; it is the completely dedicated discipline of evaluating the quality of the software. Much of testing is still being done manually, and the process is intuitively guided. Manual software testing is very repetitive, and thus very unattractive to software engineers, and also leaves a whole lot of scopes for bugs unfixed and also unmanageable in the context of large scale complex software development. Hence, it is vital to develop suit of tools for an automated or semi-automated way of testing software to improve or eliminating bug finding as well as to cut down time and cost. In order to automate the process, it is essential to generate oracles from the specification, and generate test cases to test the target software against the oracles to decide their correctness. In the case of object oriented and web based software testing, a full-scale system that can achieve such goal in the development process is non-existent.

Software testing tools and techniques suffer from the lack of generic applicability and scalability. Nevertheless, defects that are identified in previous segments of programs can be clustered according to their various properties and most importantly according to their severity. If the relationship between the software metrics measured at a certain state and the defects’ properties can be formulated together, it becomes possible to predict similar defects in other parts of the code written. The software metric data gives us the values for specific variables to measure a specific module/function or the whole software. When combined with the weighted error/defect data, this data set becomes the input for a machine learning system. Any invariant feature extractor from source codes (i.e. Daikon [4]) can be extended to extract object oriented invariant properties (i.e. multiple inheritance, copy constructor, polymorphism) as well as web based invariant features (i.e. specific annotated tags). Invariant feature vectors can be used for modeling using a host of machine learning techniques to develop robust models for identification of error properties.

II. State-of-the-art in Software Testing

Software testing can be categorized into three broad categories: i) Test case generation based software testing (black box testing) ii) model (or graph) based software testing (white box or structural testing), and iii) pattern based software testing. There are some other approaches for software testing for specific intended purposes, like software fault tolerance using fault injection method, security bugs detection using security scanning, recovery testing, grey box testing (combination of black and white box testing) etc. The following subsections describe the basic principles and brief overview of various approaches in software for the sake of completeness.

A. Black Box Testing

Test case generation based testing works by generating possible test cases and testing against possible paths of the software code. Test data selection is generally a manual process - the responsibility for which usually falls on the tester. However this practice is extremely costly, difficult and laborious. Therefore, exhaustive searching is never a possibility [19]. This is not possible as the number of combinations supported in nearly all programs can be considered, for all practical purposes, infinite. It is easy to note that testing all inputs, all timings, and all paths etc. are practically impossible. Even though this way of software testing has always been considered as incomplete, still this method has been used most extensively, started from manual testing towards development of automatic generation of test cases. One approach to the problem has been to (attempt to) simplify it away, by saying that you achieve “complete testing” if you achieve “complete coverage”.

The use of meta heuristic search techniques for the automatic generation of test data has been a burgeoning interest for many researchers in recent years [20]. Meta-heuristic search techniques are high-level frameworks which utilize heuristics in order to find solutions to...
combinatorial problems at a reasonable computational cost. A very promising approach to generate test cases is introduced which uses genetic algorithm to generate test cases [21]. Genetic algorithm uses its mutation and selection procedure to select new test cases based on the initial test case to test the software under testing for appropriate behavior.

B. Model-based Testing

Model based testing (i.e. MOPS [2]) has several advantages over test case based checking. Model checking is particularly useful because it tells exactly why a property fails rather than just raising an alarm which gives the developer a good hint to fix it. Also, the advent of model checking has introduced a new spin on formal methods: beyond their applications for verification of systems, model checkers are even more useful for finding bugs in systems [2]. Model based testing like MOPS exhaustively searches the control flow graph of a program to check if any path may violate a safety property. Security properties are defined exhaustively by the programmer analyst team.

A number of static analysis approaches have been adopted in the past to detect specific software security vulnerabilities. For example, MOPS [2], developed at the University of California at Berkeley, is a testing tool which is used to validate the conformance to the defensive programming. This testing prototype is currently being used in security critical programs developed in C programming language. The immediate extension of MOPS would be to use a database consisting of rules of defensive programming that can be used by MOPS to check for security bugs. The construction of the database of rules can be done in collaboration with the real developers who not only know the system better, but also are aware of the common mistakes that they need to check to ensure the sanity of the code.

Berkeley Lazy Abstraction Software Verification Tool (BLAST) [5], developed at the University of California at Berkeley, is a model checking testing package based on abstract-check-refine loop. It establishes the idea of building an abstract model, and checking the desired property. In case of failures, refine the model and start over. Using the lazy abstraction method and driven by the model checker, the system builds and refines a single abstract model on demand, so that different parts of the model may exhibit different degrees of precision just enough to verify the desired property [5].

Java pathfinder is normally used to locate violations of properties like deadlocks or unhandled exceptions by systematically exploring all potential execution paths of a program. For example, for a simple program using locks and unlock utilities can easily deadlock itself is two consecutive calls to lock are made without a call to unlock in the middle.

Splint [6] is a tool used to statically check for security vulnerabilities and programming mistakes. It is capable of detecting unused declarations, type inconsistencies, use before definition, unreachable code, ignored return values, execution paths with no return, likely infinite loops, and fall through cases.

CQUAL [7] is a type-based analysis tool that provides a mechanism for specifying and checking properties of C programs. It is primarily used to verify authorization hook placement in the Linux Security Model framework [8], and to detect format string vulnerabilities [9].

SLAM [10, 11] is a pioneer project that uses software model checking to verify temporal safety properties in programs. It validates a program against a well designed interface using an iterative process. The model checker determines the reachability of certain states during each iteration. The Boolean abstraction of the source program verifies the path given by the model checker. If the path is infeasible, additional predicates are added and the process enters a new iteration. SLAM, however, does not yet scale to very large programs. Jensen et al. model checked a special class of security properties in Java using only control flow analysis [12, 13].

Ernst et al. [14] aim at detecting program invariants, through instrumenting the program at hand and searching for predetermined regularities (e.g. value ranges) in the traces. Brehelin et al. [15] consider a deterministic test procedure, generating sequences of inputs for a PLA device. An HMM is trained from these
sequences and further used to generate new sequences, increasing the test coverage.

In Vardhan et al. [16], the goal is to test a concurrent asynchronous program against user-supplied constraints. Grammatical Inference is used to characterize the paths relevant to the constraint checking. Xiao et al. [17] aim at testing a game player, e.g., discovering the regions where the game is too easy/too difficult; they use active learning and rule learning to construct a model of the program. Prediction models estimate the reliability of the system, and investigate the effect of design and testing process over number of defects. Studies show that the metrics in all steps of the life cycle of a software project as design, implementation, testing, etc. should be utilized and connected with specific dependencies.

C. Pattern-based software testing

Machine learning algorithms have been proven to be practical for poorly understood problem domains that have changing conditions with respect to many values and regularities. Since software problems can be formulated as learning processes and classified according to the characteristics of defect, regular machine learning algorithms are applicable to prepare a probability distribution and analyze errors. Decision trees, artificial neural networks, Bayesian belief network and clustering techniques such as k-nearest neighborhood are examples of most commonly used techniques for software defect prediction problems. A first step to this end is to build self-aware systems, using ML to automatically model the system behavior. Along these lines, various ML approaches have been proposed for Software Testing, Software Modeling [17] and Software Debugging [22]. Ernst et al. [14], and Vardhan et al., [16], used ML techniques to provide better input to software testing approaches. Brehelin et al., [15], used ML-based approaches as a post-processor of software testing. In Xiao et al., [17], ML directly provides a model of the black box program at hand; the test is done by manually inspecting this model. In Yuriy et al. [18], proposed a technique for identifying program properties which indicate errors, they used program invariant extractor for such properties extraction and modeled them using ML techniques.

Supervised machine learning algorithms can be used over program execution to detect the number of the faulty runs, which will lead to find underlying defects. Executions are clustered according to the procedural and functional properties of this approach. Machine learning is also used to generate models of program properties that are known to cause errors. Support vector and decision tree learning tools are implemented to classify and investigate the relevant subsets of program properties. Note that most of the properties leading to faulty conditions can be classified within a few groups. Technique consists of two steps (i) training and (ii) classification. Fault relevant properties are used to generate a model, and subsequently used to select properties that are likely to cause errors and defects in the software.

Unsupervised machine learning such as clustering over function call profiles are used to determine which features enable a model to distinguish failures and non-failures. Dynamic invariant detection is used to detect likely invariants from a test suite and investigate violations that usually indicate erroneous state. This method is also used to determine counterexamples and find properties which lead to correct results for all conditions.

Defect prediction models can be classified according to the metrics used and the process step in the software life cycle. Most of the defect models use the basic metrics such as complexity and size of the software. Testing metrics that are produced in test phase are also used to estimate the sequence of defects. Another approach is to investigate the quality of design and implementation processes, that quality of design process is the best predictor for the product quality.

Two types of research can be performed on the code based metrics in terms of defect prediction. The first one is predicting whether a given code segment is defected or not. The second one is predicting the magnitude of the possible defect, if any, with respect to various viewpoints such as density, severity or priority. Estimating the defect causing potential of a given software project has a very critical value for the reliability of the project.
Given a training data set, a learning system can be set up. This system would come out with a score point that indicates how much a test data and code segment is defected. After predicting this score point, the results can be evaluated with respect to popular performance functions. The two most common options here are the Mean Absolute Error (MAE) and the Mean Squared Error (MSE). The MAE is generally used for classification, while the MSE is most commonly seen in function approximation.

Training is a preprocessing step that extracts invariant properties of programs containing known errors, converts these into a form amenable to machine learning, and applies machine learning to form a model of fault-revealing properties. During classification, the tool applies the model to properties of new code and selects the fault-revealing properties, and then the programmer uses those properties to locate latent errors in the new code.

The same invariant properties extraction tool can be used to extract object oriented properties described above, such as information hiding, encapsulation, inheritance and polymorphism. An experiment can be designed to extend the same architecture and test it on object oriented and web programming invariant properties extraction. The simple implementation model it uses is shown in Figure 1.

**D. Genetic Algorithm Based Software Testing**

Optimization goal is always common in almost every domain. Therefore, research on optimization techniques had always been of interest. Most common optimization techniques (i.e. Greedy etc.) often have problems with their functions which are not continuous or differentiable everywhere, multiple peaks or noisy. In search of robust and problem free optimization techniques, biological and physical approaches became of increasing interest over time, which includes the former evolution of Neural Networks and Genetic Algorithms. Genetic algorithms (GAs) represent a class of adaptive search techniques and procedures based on the processes of natural genetics and Darwin's principal of the survival of the fittest. There is a randomized exchange of structured information among a population of artificial chromosomes. GAs are a computer model of biological evolution. When GAs are used to solve optimization problems, good results are obtained surprisingly quickly. In the context of software testing, the basic idea is to search the domain for input variables which satisfy the goal of testing.

Genetic algorithm (GA) has been used extensively in different applications as an adaptive search method. In software testing, Genetic Algorithm is mostly used for automatic generation of test data, mostly based on white box methods [23]. GAs form a method of adaptive search in the sense that they modify the test data from one generation to the next, in order to optimize a fitness function. In contrast random testing generates test data without using any knowledge of previous test data. Based on the results from the software in test, GA selects additional test cases which tests the behavior (around initial test cases) of the software that helps identifying test cases that’s causes the system to fail (if any). Whether or not any failures are experienced, this GA approach supports increased test automation and provides increased evidence to support reasoning about the overall quality of the software.

**E. Invariant Source Code Features of Machine Learning**

- Training Data (i.e. labeled correct programs and programs with known errors)
- Test Programs (i.e. test source codes)
- Program Analysis (i.e. source code)
- Classifier (i.e. Support Vector Machine, Decision Tree, etc.)

**Figure 1: Training and Testing Model for Latent Code Error Extraction**
Invariant properties of source code that distinguishes an erroneous source code from the correct one are to be extracted from the code under observation. The technique generates machine learning models of program properties known to result from errors, and applies these models to program properties of user-written code to classify and rank properties that may lead the user to errors. Given a set of properties produced by the program analysis, the technique selects a subset of properties that are most likely to reveal an error. There had been several recent efforts in this. Fault invariant classifier [18] is one such effort which automatically determines the properties via dynamic analysis, so the user only needs to provide a program and some inputs. The implementation is able to recognize fault-revealing properties of code—that is, properties of the erroneous code that are not true of a fixed version. This fault invariant classifier uses a tool called Daikon [4] to extract properties from programs; next step is to convert program properties into a form acceptable to machine learners, and finally create and apply machine learning models. Daikon detects properties at specific program points such as procedure entries and exits; each program point is treated independently. The invariant detector is provided with a trace that contains, for each execution of a program point, the values of all variables in scope at that point. Features extracted in this way are represented as vectors and finally modeled using machine learning techniques as Support Vector Machine [24], Decision Trees [25] etc.

One similar research was to use machine learning over program executions, with the assumption that it is cheap to execute a program but expensive to verify the correctness of each execution [26]. Their goal is to indicate which runs are most likely to be faulty. They use clustering to partition test cases; similar to what is done for partition testing, but without a guarantee of internal homogeneity. Executions are clustered based on “function call profile”, or the number of times each procedure is invoked. Verifying the correctness of one randomly-chosen execution per cluster outperforms random sampling; if the execution is erroneous, then it is advantageous to test other executions in the same cluster. Their experimental evaluation measures the number of faulty executions detected rather than number of underlying faults detected.

A similar approach [27] uses dynamic invariant detection to find program errors. They detect a set of likely invariants over part of a test suite, and then look for violations of those properties over the remainder of the test suite. Violations often indicate erroneous behavior.

F. Software testing Approaches for Web Based Applications

Dynamic web Application needs a robust testing approach that will significantly improve the reliability, scalability and compatibility [28]. Web application can be modeled using regular expressions. The primary elements in the regular expressions are atomic sections which can be dynamically combined to create composite web pages. An atomic section consists of static structure and dynamic data contents. Atomic sections can be composed in three ways: sequence, selection, and aggregation. Links and control transfer in a web application have been modeled as derivations. Every derivation can be used to construct test cases. For simplicity, it is assumed that web applications are stateless. Regular expressions for complex web applications are based on HTTP and HTML principles. But these composite expressions and derivation can be extended to model XML, Java Server Pages and Active server pages.

G. Software testing Approaches for Object Oriented Programming

Components based (object oriented) modular design programming is a proven effective and popular approach for designing large scales software systems [29]. Automatic approach has been proposed to extract exact model from Object Oriented Components. Instead of one Finite State Machine (FSM), multiple separate FSM sub-models are used to extract the behavior of a component or interface class. In this approach, a FSM has been used for each field of a class, with one state for each method that sets the fields. Then restriction has been imposed on states in which some methods only read the field. In other words, the field is spitted in to two or more communicating sub models. Multiple
FSMs facilitate to model large scale software system containing millions of lines of codes. Two types of model extractor are experimented. The first is a dynamic instrumentation technique that records legal method sequences from programs. The other one is static analysis that infers pairs of methods that cannot be called consecutively. Various test suits may be constructed from automatically extracted models without knowing details of implementations. Proposed approach may be efficiently used to generate test cases based on extracted models. Additionally, extracted models may be compared with required system model that facilitate to capture errors.

III. Comparative Analysis
This section briefly highlights the utility and pitfalls of approaches discussed in the previous section.

Model Based Checking: MOPS [2] is a program analysis tool that uses temporal safety properties (i.e. violations of ordering constraints) explicitly to verify whether they are properly respected by the source code. During compile time MOPS uses model checking process for security properties using Finite State Automation (FSA) and the program as Pushdown Automation (PDA). Any violation or presence of a security property in the FSA reachable in PDA is an indication of existence of any vulnerability in the code. It can be illustrated using the following example. So, what MOPS does is based on this state automata generates formal grammar. So, finally MOPS gets a collection of formal grammar which is followed for vulnerability catches. When a test source code is passed through the system, then MOPS checks that code against all grammars and if it finds any violation then generates an error message. Finally, based on those error messages, bugs can be fixed. So, total strategy is based on grammar formulation and the core research lies on that. The most important advantage of this model is that a group of FSA, each defined for different vulnerabilities, can be combined together to come up with an integrated solution for a super set of vulnerabilities. Modular structure of this approach gives it the highest flexibility to add new rules as required and develop a generalized solution for software testing. Model checking using MOPS has been reported to be both feasible and useful for real programs. After performing extensive case studies and analyzing over one million lines of source code, it has discovered over a dozen security bugs in mature, widely deployed software. These bugs escaped notice until during their development and release, because the security properties were somewhat subtle and non-trivial to check by hand. This demonstrates that MOPS allows detecting non-obvious security violations efficiently and automatically.

Automatic Test Case Generation (using GA): Genetic algorithm approach to software usage testing has been widely is used to explore the space of input data and identify and focus on regions that cause failures. Genetic algorithms can be used as a tool to help a software tester search, locate, and isolate failures in a software system [23]. The use of genetic algorithms supports automated testing and helps to identify those failures that are most severe and likely to occur for the user. Strategic use of genetic algorithm [21] not only helps the tester to isolate failure clusters, but also provides the developer with more information concerning the faults in the software and the input values that triggered them. The developer can then use this information to search, locate, and isolate the faults that caused the failures. The result can improve efficiency of both the testing and the development teams and can support subsequent improvements in the software development process. Genetic algorithm was tested against several conditions and in all cases it was proved to be useful, particularly generating populations with following two niches properties:

a. Very likely to occur and resulted in a high failure intensity and
b. Similar, but different, test cases help to identify the failure’s root cause.

Test cases for stress testing are also clustered via GA algorithm [30]. Response times are the input for GA. ESIM has been used as testing environment. ESIM is a test automation environment for embedded software written in
the C programming. In this approach, GA and tested program run separately in their own ESIM-tasks which communicate with each other through simulated ESIM hardware ports. GA sends each input to the tested black-box program and measure the response time. The response time is the fitness value for the GA. The response time is non-deterministic i.e. it is not always same for same inputs. So, here GA has been used to cluster the slowing down loops. After enough iteration, all slowing down loops obtain higher fitness values. Then, among all possible solution paths, the particular paths associated with fitness values after certain number of generations may be tested number of times to detect flaws. In the experiment, there are $2^{32}$ possible solutions/paths. Among all, 289th path has the maximum response time based on GA fitness value. The path has been selected to be tested multiple times desiring flaws in it.

**Automatic Anomaly Detection:** DIDUCE (Dynamic Invariant Detection U Checking Engine) [3], extracts invariants dynamically from program executions and at the same time continually checks the program’s behavior against the invariants hypothesized up to that point in the program’s run(s) and reports all detected violations. When a dynamic invariant violation is detected, the invariant is relaxed to allow for the new behavior and program execution is resumed. To evaluate the effectiveness of the system, DIDUCE was applied to four significantly complex Java projects. The first program, Simulator, is a proprietary timing–accurate simulator for a class of sophisticated memory systems being considered for a multiprocessor-on-a-chip implementation, the second, Mailman, is an open–source email management utility, the third is the JSSE (Java Secure Sockets Extension) code, and finally, the fourth program, Joeq, is a Java Virtual Machine system. In all cases, DIDUCE helps locating algorithmic errors that fail to handle corner cases correctly. As DIDUCE isolates the context in which an error occurs, the programmer can analyze the problematic scenario better and zoom in to the problem easily. DIDUCE helps user find errors in inputs, unfamiliar codes, and even un-instrumented components. The latter is achieved by noting violations of invariants governing the interface to un-instrumented domains.

**Path Exploration using Machine Learning:** This approach, published as EXIST, for Exploration -eXploitation Inference for Software Testing [31], proceeds by iteratively generating candidate paths based on the current distribution on the program paths, and updating this distribution after the path has been labeled as feasible or infeasible with the goal is to maximize the number of distinct feasible paths found along the process, as opposed to learning a concept or a fixed policy. EXIST is first validated on the real-world problem, including 36 nodes and 46 edges. The ratio of feasible paths is about $10^{-5}$ for a maximum path length $T = 250$. The reported result is the number of distinct feasible paths found out of 10,000 generated paths, averaged over 10 independent runs. With respect to Statistical Software Testing, this approach dramatically increases the ratio of (distinct) feasible paths generated. This automatic intelligent exploration of paths can be a baseline for testing using adaptive test case generation.

**Fault-Revealing Properties using Machine Learning:** Fault invariant classifier reported in [18] discusses issues related to identifying the fault-revealing properties of source code. First, a program analysis tool produces properties of the target program. Second, a classifier ranks each property by its likelihood of being fault-revealing. A user who is interested in finding latent errors can start by examining the properties classified as most likely to be fault-revealing. Since machine learners are not guaranteed to produce perfect models, this ranking is not guaranteed to be perfect, but examining the properties labeled as fault-revealing is more likely to lead the user to an error than examining randomly selected properties. The user only needs one fault-revealing property to detect an error, so the user should examine the properties according to their rank, until an error is discovered, and rerun the tool after fixing the program code. Experimental evaluation of the Fault Invariant Classifier included twelve subject programs. Eight of these are written in C, and four are written in Java. There are 373 faulty versions of the twelve programs. Fault Invariant Classifier was very
effective in classifying fault revealing properties; also ranking and selecting the top properties was found to be more advantageous than selecting all properties considered fault revealing by the machine learner. For C programs, on average 45% of the top 80 properties are fault-revealing. For Java programs, 59% of the top 80 properties are fault revealing. Not all fault-revealing properties necessarily directly lead a programmer to the error, but in a preliminary study, most did. Therefore, on average the user only has to examine 3 of the properties to be led to an error. Another significant attempt is to use machine learning technique to detect anomaly/fault in the deployed software is to use fully observable Markov model, built using the probability distributions of observations [32]. Observations are as the form of predicate states and instrumented line numbers. Predicated states and instrumented line numbers are produced in the instrumentation phase. After the instrumentation phase, the whole program has been executed to calculate the probability. Then learning phase takes place in which predicate states and instrumented line numbers are clustered using machine learning techniques. Then a Markov model is constructed using the probabilities. This model is then included in the software. In deployed section, if unseen events occurred, anomaly will be detected. Two case studies have been experimented in the paper. In the first study, four NANOXML versions are used. Each version comes with seeded faults, which are activated individually. Four versions have 33 seeded faults. Each version has between 214 - 216 test cases. When a fault has been activated for a given version, the whole version has been instrumented and executed to produce predicate states. Then the predicate states are used as input for learning section. After the desire model was built, the failing test cases from that particular version’s test suite were then executed on the instrumented versions and the predicate state traces generated were input to the anomaly detection module, which consists primarily of the model, to perform online anomaly detection. In the experiment, for the 33 seeded faults across the four versions, all failing executions in 24 seeded faults have been detected as anomalous. Furthermore, the technique was able to detect some of failing executions in 7 other seeded faults as anomalous.

**IV. Conclusions**

This paper presents a comprehensive review on various aspects of large-scale software testing. In particular, issues related to development of software testing tools and techniques to overcome the limitations of applicability and scalability were studied. Several implementation ideas are also discussed in developing such tools. For example, the relationship between the software metrics measured at a certain state and the defects’ properties will be used to predict similar defects in other parts of the code. The Daikon will be extended to extract object oriented invariant properties (i.e. multiple inheritance, copy constructor, polymorphism) as well as web based invariant features (i.e. specific annotated tags). Invariant feature vectors will be used for modeling using a host of machine learning techniques to develop robust models for identification of error properties.

**Acknowledgements:** Authors gratefully acknowledges the support received from Software Testing and Excellence Program (STEP) and startup grants from the Department of Electrical and Computer Engineering for partially supporting the research.

**References**


Abstract

Testing has emerged as a distinct responsibility within software development. Rather than being viewed as a sequential line responsibility in software development, this paper argues that it may be better conceptualized as a service responsibility. Testing as a service has three key aspects: 1) as a service to developers, 2) as an indirect service to end users of software products, and 3) as the service component of software products. This paper draws on the SERVQUAL literature in information systems and the International Standards Organization (ISO) standards for software quality to propose a structured quality measurement for testing as a service. In the paper, this TESTQUAL framework is described and its components defined and exemplified. It also discusses practical applications of TESTQUAL and research model for investigating its antecedents and impacts is presented.

1.0 Introduction

Recent thinking about the nature of software architecture in organizations has been significantly influenced by models such as service-oriented architecture (SOA) and software-as-a-service (SaaS) (Goth, 2008). This trend is partly the result of the fact that it is often difficult for developers in disparate organizations to determine which component of software is interfacing with others. At the same time, end users, who have become more technologically sophisticated, have significantly higher expectations today about the quality and functionality of software. Traditional testing practices, which have not changed significantly over the years, are not able to satisfy these emerging requirements. It is therefore imperative to consider the relationship between developers, testers and end users from newer perspectives. One such new perspective is that of “testing as a service”, which is in line with the general trend in IT management to decompose the value an IT unit in terms of a structured set of service offerings rendered to customers both internal and external to an organization. In both academic and professional circles, this is often referred to as the “IT services” paradigm. This paradigm argues that organizations need to consider each value bundle provided by the corporate IT unit from a “service provider and service consumer” relational perspective. This then leads to a focus on measuring and managing the service quality in such relations.

Testing can be easily viewed as a service provided to software developers in relation to error minimization, validation and verification. Without the services provided by testers, the burden of error finding, verification and validation would fall onto the development function which may not be equipped in terms of capabilities and technical resources to
properly handle a comprehensive testing process. This is especially critical as it has been argued that developers are more technically oriented while testers are more process and business oriented (Cohen et al, 2004). A dual service aspect relates to end users who purchase and utilize software products. Testers ensure that the product of software development is of high quality and meets the business needs of the users. They provide this as an indirect service to the users.

This service perspective to testing is an important paradigm shift from the old perspective where testing is simply a technical quality assurance line function. In this new paradigm, issues such as software usability, user satisfaction, mapping to business needs, information systems coherence, and etc., become part of the testing agenda leading to a more holistic role for testers beyond just technical testing of code. In order to measure and manage the service provided by testing, business and IT managers require a tool that allows for the quantification of the quality of service provided. This paper attempts to fill this need by adapting an academic and practitioner validated service measurement tool – SERVQUAL – to the software testing context.

2.0 Traditional Approaches to Testing

The method of choice for developing complex systems is the Systems Development Lifecycle Cycle (Mahmood, 1987) which has been used most for complex integrated systems that are testing intensive and require large investments to develop. This traditional SDLC approach is usually composed of consecutive stages, where the output of each step becomes the input for the subsequent stages. Although the steps are recursive, in practice, once a stage has produced its output, it is considered complete. In general, the individual stages of the SDLC are Analysis, Design, Development, Integration and Testing, and Installation and Acceptance Phase. Traditionally, in SDLC frameworks, testing is usually relegated to a technical verification and validation of the code components. Testing usually focuses only on technical issues of whether the code of the product runs correctly, does not have compatibility issues with other software components, and only briefly checks if the coded product complies with the gathered requirements. The requirements themselves are seldom tested with the users and the problem specification is seldom verified to check if it maps correctly to the business need.

This traditional way of conducting testing as code quality assurance does not provide the in-depth overview of the software development product as a whole that is needed to properly fulfill the needs of business. Technical excellence is no longer enough; while code that works is a minimum requirement of any software product, it no longer is enough to satisfy the expectations of business users. The testing-as-service perspective comes to answer the limitations of a purely technical approach to testing, by injecting the business and human factor in the testing and development process.

Also, traditionally, the common practice of software testing was that testing is performed by an independent group of testers after the functionality is developed by developers but before it is shipped to the customer. Past studies revealed that testing is separate from the development activities due to this sequential working mode of developers and testers (Whittaker, 2000), which is illustrated in Figure 1. In figure 1, there are two groups of arrows with opposite directions: the arrows pointing from left to right are the software products (or code) from Developers to Testers, and from Testers to End Users; the arrows in the other direction are feedback from Testers to Developers (feedback 1), from End User to Developers (Feedback 2), and from End Users to Testers (Feedback 3). The developers, testers and end users are temporally and physically separated in the software production processes; testing is merely viewed as the line function sitting somewhere between the development activities and full implementation for users’ usage, and the facilitating role of testers for software development does not exist.
One major drawback of this sequential development and testing activities leads to the conflict arising between developers and testers, mainly conflict of time allocated to these two disparate system development activities (Cohen et al., 2004). Time is persistently scarce resource in software development, therefore, developers and testers have to constantly compete for more time; often times testing is postponed and planning testing time reduced in order to be able to meet the delivery schedule. The sequential software development practice often results in compromises of quality of product that result from not enough time being spent on testing the code, but more importantly, on testing the requirements and mapping of software specification to the actual business needs. In this out-of-date testing practice, testers are innately not equipped with the mindset of providing quality service to the developers; they are very much focused on the mechanical side of testing - they measure their productivity with number of test cases executed per person per unit time, total bugs found in a module, number of customer reported bugs, number of test cases executed per person per unit time, etc. There’s pressing need to position testers differently, so that the limitations of the traditional testing practice can be overcome.

When developers and testers work in separate locations and at different SDLC phases, communications and personal relationship can be dysfunctional. To reduce the intellectual and physical distance of developers and testers, management has to structure them into an integrated team consisting of testers, developers, and end users. A common way is to form heterogeneous teams that are comprised of pertinent stakeholders. Another modern practice is to start software testing as early as possible, and in one extreme, to start software testing at the same moment the project starts and continues until the project finishes. The major reason for involving testers as early as possible is that the earlier a defect is found the cheaper it is to fix it; this is also an effective mechanism for developers and testers to gain common grounds and reduce the conflict.

New software development methodologies such as prototyping, Rapid Application Development (RAD) and Joint Application Design (JAD) give the testers as well as end users an unprecedentedly significant role than traditional waterfall SDLC. The development team, testing organization and end users are brought together closely throughout the entire system development process, both logically and physically. Logically, software products are iteratively worked on among them, shuffling through them multiple rounds even on one day. They might need to carry conversations on a daily basis during the software development phases. Physically they work in close proximity, - may sit side by side in one room. The interaction between developers, testers and end users are totally distinctive from the way they interact with one another in the sequential software development practice. When the management signals that testing is an important component of the software development process to the project team, and stresses testers as the providers of testing service, testers can take a much more proactive role in participating in the software development project. In this testing-driven system development paradigm, all stakeholders can benefit from the win-win synergy that arises from repositioning of testers as service providers. The conceptualization of testers as service providers is the basis for the next section.

### 3.0 Testing as a Service

Figure 2 shows the service relationships among developers, testers and end users in the new paradigm of testing as a service. Software testing is broad, encompassing a wide spectrum of different activities, from testing a small piece of code by the developer (unit testing), customer validation of the system (acceptance testing), to the monitoring at run-time of a network-centric service-oriented application. There are different sets of objectives for testing at different levels (unit test, component test/subsystem test, integration test) and at different stages of software productions cycle; it’s becoming requisite that testers can work together with both developers and end users to reach these different objectives. The testing as a service paradigm promises to have the capability of conceptualizing all these activities as distinct services to pertinent constituencies.

Figure 2 shows that testers provide key services to two distinct constituencies; service to developers (Service I) and service to end users (Service II). There is an additional indirect service (Service III) that focuses on testers’ role in relation to assuring quality and functionality of the final software product.
Service I is provided to developers and pertains to error detection, verification and validation services. This service provides feedback to developers in relation to errors in the code they have written to ensure that the code works. For example, unit testing by testers is a service that provides instant feedback thereby ensuring that individual modules work right. Another aspect of this service is exemplified by code review and functional testing services that ensure that code written by developers is consistent with business and software specifications.

A third aspect of this service is exemplified by boundary testing, stress testing and load testing services that focus on ensuring that the code written by developers will work in the intended, real world business context for which the software is being developed.

Service II captures all categories of testing services that testers provide to ensure that users will receive software that meets all their needs. Specific examples of these services include requirements validation at early stages plus alpha testing, beta testing and usability testing at later stages of software development. In requirements validation, testers need to understand end user expectations about the software so that appropriate testing plans and strategies can be implemented to guard the users’ interests. Testers will analyze the requirements that are gathered in system analysis phase to arrive at test cases so that they can establish the linkage of between the user requirements and the test cases. User requirements validation is akin to services performed by testers in regard to business needs validation (functional requirements), software performance (performance requirements), and security (security requirements). Different applications with varying sets of requirements often require a diverse set of testing services. Alpha testing services are mechanisms by which testers engage users directly in validation as users are brought into controlled testing environments. Beta testing services engage users through controlled software distribution processes tied to feedback loops to enable users to test software in real world conditions. The service role of testers here is often to act as facilitators in dynamic feedback loops designed to customize and localize software to specific user environments. This is a key service that testers provide to users.

Service III focuses on tester contributions to actual product that is delivered to users by developers. In one sense, testers provide certification and assurance services as the final arbiters of quality of the software product. Business organizations that depend on software driving their central business processes are unlikely to adopt and implement software systems unless they are assured that the software has been tested and certified by qualified testers using sound testing methods. Software products are expected to demonstrate all the attributes that can measure up to pre-established standards, and testers have a key service responsibility in this regard. Many of these attributes pertain to technical standards that have to be met for certification to be awarded. For example, ISO 9126 lays out specific attributes for demonstrating quality of software products. Often in specific industries where there are life threatening or other consequences of software failure, certified testers have to confirm that implemented software products meet strict thresholds for reliability, security and stability. In outsourced software development such testing services are often prescribed in the form of service levels for software performance. Often such key testing certification services are best provided by independent third party testers who specialize in specific business contexts.

Service I, II and III as illustrated in Figure 2 and explained above, form the foundation of the TESTQUAL tool for testing service quality that is proposed in this paper as a holistic measurement of service quality for testing environments. The next section reviews service quality measurements from various service environments with a focus on the SERVQUAL and ISO approaches. Subsequently, Section 5 will describe the application of these measurements to the testing context as laid out in this section to develop the TESTQUAL tool.
4.0 Service quality and standards

The conventional testing metrics primarily measure testers’ productivity and the outcome they deliver in technical terms (e.g., numbers of fixed bugs). The social aspect of testing service has largely been ignored. Section 3 showed that testers provided service from three distinct yet complementary perspectives (developers, end users, and the software product) and includes both technical and social aspects of testing service quality. Sections 4.1 and 4.2 will elaborate the literature on SERVQUAL and ISO standards to provide a foundation for developing a structured scale to measure the service quality of testing (this is to be discussed in Section 5).

There are two primary characterizations of service quality in the literature. One is the Nordic perspective (Grönroos, 1982, 1984), which uses global terms to define service quality in teams of functional quality and technical quality. Grönroos (1982, 1984) argued that functional quality is generally perceived as being more important than technical quality in the service context. Functional quality revolves around how the customer receives the technical outcomes, so the customers perceive service quality based on the manner in which it is delivered. On the other hand, technical quality revolves around what the customer actually receives from the service. In another words, functional quality refers to the format (or process-based view) of service, and technical quality refers to the content (outcome-based view) of the service. In relation to the testing service provided to developers, the process view focuses on how the testing service is provided through the social interaction while the outcome view focuses on the specific content of the service provided; both dimensions may be equally important. On the other hand, in relation to the testing service provided to end users, the focus of measurement would not be how testers provide the services to them, but the nature of contents of the service especially in terms of outcomes. This difference of focus is determined by the working structure of software development group whereby testers work closely with developer (therefore the quality of social interaction should be considered) and only have limited (indirect) contact with the end users.

The alternative perspective, which is more prominently used in North America, emphasizes service encounter characteristics that are perceived by the service recipient – such as tangibility, reliability, responsiveness, empathy, and assurances (Parasuraman, Zeithaml, & Berry, 1988 and 1991).

This conceptualization of service dominates the marketing and information systems literature and is known as the SERVQUAL rubric. SERVQUAL includes a widely-utilized measurement instrument for assessing service recipients’ expectations and perceptions of service deliveries. Nybeck, Morales, Ladhari, and Pons (2002) have noted that the SERVQUAL measuring tool remains the most complete attempt to conceptualize and measure service quality. Our present exploration of service quality in the software testing context is primarily based on this SERVQUAL perspective while being consistent with the duality promoted by the Nordic perspective: of technical (outcome) and functional (process) aspects of service delivery in relation to testing. In our view, the service provided by testers to developers has to be more oriented toward the process-based view of testing service provision, while the service to end users and service in relation to the software product takes more of an outcome-based orientation.

4.1 SERVQUAL literature

Parasuraman, Zeithaml, and Berry (1988, 1991) initially developed the SERVQUAL instrument to measure the gap between customer expectations and services received, based on interviews and focus group meetings with managers and customers from large service companies in various industries. They found that customers basically used the same criteria in evaluating service quality and these criteria spanned virtually all types of service in the consumer retail sector. This multiple-item instrument quantifies customers’ global (as against transaction-specific) assessment of a supplier company’s service quality. The scale measures the service quality of service personnel along five dimensions: reliability, responsiveness, assurance, empathy, and tangibles. They define these five dimensions as: 1) tangibles - physical facilities, equipment and appearance of service personnel, 2) reliability - ability of service personnel to perform the promised service dependably and accurately, 3) responsiveness - willingness to help and provide prompt service, 4) assurance - knowledge and courtesy of service personnel and their ability to inspire trust and confidence, and 5) empathy – a caring orientation and the provision of individualized attention to service recipients.

There has been some debate over the usefulness of the SERVQUAL’s gap measure (perceived service quality minus expected service quality) pertaining to the conceptual and empirical relevance of SERVQUAL versus performance only service scores (SERVPERF). Parasuraman et al. (1991) have noted that the SERVPERF scores produced higher adjusted R² values when compared
to SERVQUAL’s gap scores for each of the five dimensions. Van Dyke, Kappelman and Prybutok (1997) also questioned the SERVQUAL gap measurement approach, the interpretation and operationalization of the SERVQUAL expectation construct, and the reliability and validity of SERVQUAL dimensionality. On the other hand, Pitt, Watson and Kavan (1995 and 1997) have demonstrated that the problems of reliability of difference score calculations in SERVQUAL are not nearly as serious, and suggest that the marginal empirical benefit of a perceptual-based (SERVPERF) service quality measure does not justify the loss of managerial diagnostic capabilities found in the SERVQUAL gap measure. Moreover, there are studies that advocate dual levels of service expectations. PZB (1994) recommend two different comparison norms for service quality assessment: one is the level of service a customer believes can and should be delivered, which is “desired service”; the other is the level of service that the customer considers acceptable, which is “adequate service” or “minimum service” (see Figure 3). The reasons for separating the expected service into these two levels is that customer service expectations are characterized by a range of levels between desired service and adequate service, rather than a single point. This band between desired service and minimum service is termed as the Zone of Tolerance (ZOT). The ZOT SERVQUAL instrumentation has been empirically validated by Kettinger and Lee (2005) in the information systems context (see Figure 3). By including all the three level of service quality (desired, minimum and perceived), the ZOT service approach can help provide a more valid basis for diagnosis and judgment concerning service deficiencies and service quality management (Kettinger and Lee, 2005).

![Figure 3: Kettinger and Lee’s (2005) dual expectation comparison standards: desired and adequate IS service as adapted from V. Zeithaml, L. L. Berry and A. Parasuraman (1993).](image)

While SERVQUAL has an established place in the marketing literature, it has been introduced to the information systems field to assess the quality of electronic services in the last decade. Zeithaml, Parasuraman and Malhotra (2000) developed an electronic version of SERVQUAL for measuring e-service quality through a three-stage process using exploratory focus groups and two phases of empirical data collection and analysis. The resulting e-SERVQUAL dimensions are efficiency, reliability, fulfillment, privacy, responsiveness, compensation, and contact. Gefen (2002) extended the SERVQUAL conceptualization to electronic contexts and found that the five standard SERVQUAL dimensions could be reduced to three when conceptualized for the context of online service quality. These three e-SQ dimensions are tangibles, a combined dimension of responsiveness, reliability and assurance, and empathy. Tangibles were found to be the most important dimension in increasing customer loyalty, while the combined “responsiveness” dimension was considered most critical in increasing customer trust. Gefen (2002) also advocated that e-SQ requires scale development that extends beyond merely adapting offline scales.

As e-service generally refers to web-based services, Liu and Arnett (2000) identified service quality as an important measure of Web site success. In their empirical study, service quality is measured as quick responsiveness, assurance, empathy, and follow-up service. Santos (2003) used focus groups to investigate the dimensions of web-based e-service quality, and identified two major dimensions—
incubative dimension and an active dimension. The incubative dimension includes ease of use, appearance, linkage, structure, layout and content. The active dimension includes reliability, efficiency, support, communication, security, and incentives. Variations of the SERVQUAL scales such as WebQual (Loiacono, Watson, & Goodhue, 2002, 2007), SiteQual (Yoo & Donthu, 2001), and eTailQ(Wolfinbarge and Gilly, 2003) have also been developed and applied generally for the contexts of e-retailers providing service to online consumers. A key objective of this paper is to develop a variation for software testing services provided in the software development context.

4.2 Software Quality

The quality of software developed is a principal concern of every stakeholder in a software development project. Software quality should be interpreted in light of the concept of “purpose of use”, considering both internal attributes (product characteristics) and the external attributes (aim of the use) (Issac, Rajendran, & Anantharaman, 2003). Therefore, assessment of software quality should take into account the following multiple viewpoints. From the perspective of developers, software quality can be defined by the conformity to functional and performance requirements that are explicitly fixed, conformity to development standards that are explicitly documented, and conformity to implied characteristics expected of all software developed in a professional manner (Pressman, 1988). From the users’ perspective, software quality can be defined with respect to all the properties that satisfy correctly and efficiently the present and future real needs of buyers and users of software (Issac et al., 2003). These perspectives can be economic as represented by managers, be both technical and social as represented by developers, and be utility-like as represented by users.

Although the importance of customer-perceived quality is recognized in many fields (especially in manufacturing, marketing and service organizations), there exists no rigorous framework for measuring customer-perceptions of software quality. Issac et al. (2003) proposed a conceptual framework for customer-perceived software quality and an instrument to measure it. This is based on the Total Quality Management (TQM) framework and consists of six dimension; namely, product quality characteristics, operational effectiveness, process quality, employee competence, client focus, and infrastructure and facilities.

It is accepted that software quality does not improve unless it is measured (Reichheld & Sasser, 1990). Measures of software quality are often difficult to articulate and no single measure is adequate by itself. Organizations such as the International Standards Organization (ISO) have developed and promulgated general guidelines and standards of practices for software quality. There is an international standard for the evaluation of software quality such as ISO 9126. As described under the ISO standard ISO 9126, there are many technical requirements for measuring software quality, such as capability, reliability, efficiency, portability, maintainability, compatibility, and usability (See Table 1).

There is also an alternative to the service quality approach that is the focus of this paper. This is the process improvement approach that is the basis for methodologies such as the Capability Maturity Model Integrated (CMMI) and Six Sigma that have also been applied to software development with varying levels of success. However, these approaches do not focus directly on the software testing responsibility so that our view will not incorporate them in this paper.

5.0 TestQual: Aspects, definitions and criteria

Testers are under increasing pressure to demonstrate that their services are customer-focused and that continuous performance improvement is being delivered. The traditional software testing metrics cannot satisfy the need of measuring up to the developers’ and end users’ expectations on both testers and software applications. Based on the SERVQUAL literature and ISO for software quality, we made the first attempt to establish a structured scale to measure the quality of testers’ service in the context of software testing – TestQual (see Table 2). TestQual consists of three segments, with each dedicated to measure service quality to developers, service quality in relation to software product, and service quality of indirect service to end users.

5.1 Service quality to developers

Even though many tools and automated methods can facilitate lots of the testing tasks, the overall testing result ultimately depends on the interpersonal interactions of people co-producing the software (Cohen et al., 2004). The social interaction between developers and testers become evidently necessary in modern testing practice. The measurement for service to developers mainly draws upon the SERVQUAL literature, which primarily taps the human aspect of service quality, as SERVQUAL was originally intended. Six aspects of TestQual that can be used to measure service to developers are identified, namely, Tangibles, Reliability, Responsiveness, Assurance, Rapport and Excellence.
Definitions for Reliability, Responsiveness and Assurance are directly adapted from the original SERVQUAL. The definition of Tangibles is distinctive from what would be directly adapted from the one that were defined by PZB (1988, 1991), which in here refers to a set of attributes about physical testing environment, testing deliverables, and documentation, instead of referring to the physically appearance of testers. There’s one aspect called Rapport that is similar to Empathy in original SERVQUAL, which refers to caring and individualized attention to the developers. Since developers are focused n the technical nuances of software design and testers are more focused on user requirements (Cohen et al., 2004), they can be at odds with each other and conflict between developers and testers can arise. Conflict can adversely affect work process, the quality of work, and the quality of work life. When confrontations arise because of the conflict, courtesy and effective communication can be good methods for them to resolve the conflicts, and the existence of harmonious atmosphere can help the two groups with different sets of minds be cooperative and keep tight pace with each other. Dedication/Excel is the aspect that has not been discussed in SERVQUAL literature. Developers are usually the ones who often “flex the intellectual muscles” to seek “for novels ways to achieve results” (P. 78, Cohen et al., 2004). Testers are viewed to be more rigid, and they need to verify and validate the user requirements that are already documented. On top of having considerable level of understanding of the users’ needs, they are expected to be proactive in helping developers in terms of software design and coding, be creative in giving better solution that can fulfill users’ needs when possible, and they need to wholeheartedly dedicate to the core mission of the software under development.

5.2 Service quality of indirect service to end users

This aspect of service relates to confirming the expectation of users regarding software quality. This is called an “indirect” testing service since it is performed through the medium of the software product which the users interact with. The dimensions that are investigated under this section pertain to four individual characteristics of service quality: tangibles, assurance, agency oversight and user satisfaction.

Tangibles refer to the set of attributes about the physical environment of software use that users come in direct contact with, as well as the actual software deliverables and documentation. These attributes, although characteristics of the software, pertain indirectly to the satisfaction of the users with the use of the software, not the software itself. If the software is easy to use, performs the tasks needed of it, and is pleasing in an aesthetic way, this would lead to user satisfaction. Also the documentation regarding the use of the software has to be of quality in order to increase the user satisfaction with the software. Testers need to oversee all these issues and, based on the quality of the service they provide in making sure these attributes meet and even exceed the expectation of the users, the quality of the service provided by them to the users is higher or lower.

Assurance refers to the guarantee that the testers offer users in regard to the performance of the software that they use. This guarantee pertains to the reliability, accuracy and consistency of the software’s performance. Also the guarantee assures users that the software meets security and privacy standards that are accepted by the organization. All these attributes have to meet user expectation and testers make sure that the software that bears their approval meets the quality standards that the organization and third party standards organizations recommend for software quality.

Agency oversight pertains to the overall responsibility of the testers to ensure that all attributes related to and characteristic of the software and its use are of high quality, meet the requirements of the users, and are connected to real business needs. User satisfaction is the holistic view of indirect satisfaction which look directly at the use of the software and the satisfaction that the users gain by the use of the new system. Alpha and beta testing puts the users in direct contact with the software and allows measurement of their satisfaction. This is the ultimate test of software, before its release in the production environment. The testers have a paramount role during this stage and the service provided by them deals with final error checking and any concluding validation and verification issues.

5.3 Service quality in relation to the software product

The main focus of testers is the software product created by developers. This is where testers provide the most value by supplying error checking, verification and validation services directly to the developed software. The attributes of this service follow international (ISO 9126) standards for software quality that ensure that the characteristics that the software exhibits are up to par with industry benchmarks and meet the minimum requirements imposed by users. The attributes of the software product related service are: functionality, reliability, usability, efficiency, maintainability, portability and scalability.

Functionality is a direct reflection of the ability of the software to perform the intended tasks that it was developed for. Testers ensure that the capabilities of the software are adequate to perform
the intended tasks that the software will be used for. Issues of suitability of the software for the tasks at hand, accuracy of inputs and outputs, interoperability with systems already in place, compliance with internal (organizational) and external (industry and governmental) standards and security and privacy requirements are tested and certified as part of offering this service. This is an assurance service that the software will perform as needed.

Reliability investigates the capability of the software to maintain its performance level over different conditions and time. If functionality looked at performance, reliability looks at sustainability of performance. The maturity and consistency of the software implementation in regard to the predictability of software outputs is tested and the recoverability from systems failures and from data loss is verified. Also the fault tolerance of the software is tested, since good business IT systems have to have a certain level of fault tolerance to deal with unforeseen events.

Usability is the degree of effort required for learning and using the software by its intended users. The learnability of software use and the understandability of the interface, software use procedures and documentation are important issues that are tested here. Also the operability of the software is checked to see if the typical users are able to properly operate the software. The service provided here has two parts because it looks at the software itself, but takes into account the user in his/her interaction with the software.

Efficiency refers to the performance and resource use of the software given certain circumstances. Testers verify the responsiveness of the software when answering requests from the users, as well as the responsiveness of the interface. Also, given an environment in which the software activates, testers ensure that the amount of resources consumed by running the software is acceptable and does not lead to resource shortfalls when the software is run in conjunction with other software.

Maintainability pertains to the modularity of software design and the effort required to modify and upgrade the software with additions and improvements. Also the stability of the software in the environment in which it operates has to be tested to make sure that adequate levels of operation are maintained. The behavior of the software also has to be traceable through logs, functional dumps and other tracking methods. Tracking of software behavior and of errors and failures that arise allows better changeability and is directly related to the testability of the software.

Portability is the degree to which the software is transferable to multiple environments and configurations. As the business environment evolves, and as new hardware and software platforms are developed and implemented, the software developed and tested has to allow for transferability to the new environments and platforms and has to be adaptable enough to do this with the minimum effort. Testers provide a service here that pertains to ensuring that, given future business and technological development, the software developed will allow easy installation/reinstallation, can be replaced and transferred with minimum effort and can easily adapt to new environments.

Scalability is the last attribute that characterizes the tester provided service in relation to the software product. Scalability refers to the ease by which the software can be enhanced and expanded, given new business needs, greater demands and changing requirements. Testers need to ensure that the software that they test is flexible enough to handle growing transaction loads and requirements enhancements without needing any major modifications.
6.0 Antecedents and outcomes of TestQual

6.1 Antecedents of TestQual

We categorize the antecedent of TestQual into three groups: organizational level (service culture/climate and resource allocation), project level (methodology, scope of testing, communication and leadership) and individual level factors that include testers’ capabilities, expertise, and personalities.

One of themes in service quality suggests that organizations must create and maintain a service climate in order for employees to effectively deliver service (Schneider, 1998). Management can help promoting such a service oriented culture/climate that emphasizes that testing organization is a service provider and developers and end users are their internal customers.

Resources including time allocated to testing and training provided to testers also determines the service quality that testers can deliver. Time is always a scarce resource in project management and time itself is a requirement for effective testing. Mature testers who have received sufficient training for strengthening both testing expertise and business understanding can be more effective in provision of testing.

SDLC is the traditional way of system development, and execution-based testing is usually conducted with the primary objective of finding defects in the code after designing phase under this methodology. The problems that are found at this late stage of software development are a lot more costly to fix than if the problems are found in the requirement phase. Therefore, involvement of testing resourcing as early as possible within iterative development life cycle is prerequisite to conquer the disadvantages of traditional SDLC. Modern system development methodology based on prototyping such as RAD, JAD and eXtreme programming can provide the testers with the necessary working environment and job structure to provide quality testing service. Project manager should use a system development methodology that incorporates and considers testing from inception through project completion.

The Scope of testing comprises the responsibilities that testing has in relation to what, when and who tests various parts of the software developed. Unit test, integration, alpha testing, beta testing, requirement testing are some of the few types of tests that testers can run on the software. Depending on where tester responsibility starts and finishes, the scope of testing may be more or less extensive, and the service that tester need to provide...
varies. When the scope of testing enlarges, the testing service tends to be more of strategic effect on the whole processes of the system development.

Lack of leadership has been a problem in system testing, resulting in an out of control testing environment (Frolick and Janz, 1998). With more management support or involvement in testing practice, the management’s leadership style is also an important antecedent to the quality of deliverables of the project constituencies. Participative or authoritarian leadership leads to different outcomes, where a participative style is more amenable to infusing a proactive service-oriented mindset to the testers, by allowing them to have an effective say in the various milestones of development processes.

One of the key findings of the project management is that the communication flow structure of business processes was a particularly strong determinant of most of the quality and productivity problems, more so than either the activity flow or the material flow structure of the business processes (Kock, 2006). The determining role of communication flow applies to testing practice as well. The communication flow between developers and testers should be free of barrier, thus horizontal communication is preferred to the vertical, which does not require the superiors’ approval. Communication media (face-to-face, email, and etc.) is also important. As Cohen et al. suggested the physical co-location of developers and testers can help the testing service be provided more effectively and efficiently.

Testers’ personal characteristics that include capabilities, experience, and personalities can also determine the service they provide. Chilton et al. (2005) note that the personalities types are an important factor in successful team performance and suggest that complex task in software develop need a balanced team opposite personality types. The fit between the developers’ preferred cognitive style and the developer’s perception of the cognitive style required for the existing job environment is instrumental in increase his/her job performance (Chilton et al. 2005). Accordingly, we expected that the existing of fit for the testers can positively impact the quality of their testing service as well. One potential instrument for evaluating testers is the Myers-Briggs (MBTI) instrument that includes four sets of dichotomies: attitudes (extravert vs. introvert), function information gathering function (sensing vs. intuition), decision-making function (thinking vs. intuiting), and lifestyle (judging vs. perceiving).

6.2 Outcomes of TestQual

Software product quality will be the first outcome that will derive from higher quality testing service that is provided to the development team. Testing is just like Quality Assurance (QA) service that monitors the whole software development process. The software design, coding and integration will be greatly enhanced by testing service.

Development time will be dramatically reduced when testers can coordinate with developers and users, and provide adequate and even superior service at proper time during software development life cycle. As noted by many testing researcher as Frolick and Janz (1998), high quality software testing service, if provided from project inception, can not only increase software quality, but also overall system development cycle time.

Testing service such as requirement validation, alpha testing, beta testing, particular testing service that is requested on the certain desired characteristics of the software product can make the software product more error-free and strictly conforms to the users’ requirements, thus can increase user satisfaction when the system is put in implementation and up running. Increased software product quality and development time reduction can also contribute to user satisfaction.

Based on the Job Characteristic Model (JCM) (Hackman and Lawler, 1971; Hackman and Oldman, 1975), feedback in the job that a worker performs and felt meaningfulness of the job can be effective in increasing his/her job satisfaction. Testing activity as service provided to the developers can be viewed as constant feedback from testers. Testers’ focus on the user requirement can reinforce the developers’ felt meaningfulness of their software development job tasks, as developers often times are heavily oriented toward the technical side of software development, and the fundamental goal of satisfying users’ need and being aligned with organizational objectives is ignored. So developers’ job satisfaction can improve because of the intensive feedback from testers and enhanced felt meaningfulness toward the job they perform. On the other hand, users’ requests on the desired features of a new application are fulfilled, which is facilitated through the user-centered testing service. Therefore, end users’ appraisal of their job experience can be elevated to a more pleasurable emotional state. End users’ job satisfaction could also be partially derived from better software product quality and reduced software development cycle time.

7.0 Conclusion

7.1 Implications for academia

This paper builds on the established the well known SERVQUAL literature as well as ISO of software quality. TestQual that is developed in this
paper adapts and incorporates SERVQUAL to measure service quality in a new context of software development, showing that testing service quality should be measured on three aspects. This further expands the Service concept to an internal human-contact area, which was not previously thought of as a service. Since many academicians are looking at concepts related to Service Oriented Architecture (SOA), our paper is the first attempt to add confirmation to the idea of the software development process (of which testing is part of) as a service to business, as well as bringing very technical matters closer to the stream of service research.

7.2 Implications for practice

The contribution to practice that this paper provides is twofold. On one hand, the paper posits a new way of viewing testing as a service provider, which is different from the paradigm where testing is a simple technical verification and validation function that does not tie business needs with technical implementations. This new testing as service perspective, in our contention, will improve software development process by allowing structured and implementable input from all stakeholders in the development processes. Testers that act as more than technical specialist can provide better verification and validation services that are no longer limited to technical issues and that permit the mapping of the technical product onto the reality constellation of business needs and requirements. TestQual can assist the testers in identifying cost-effective ways of closing service quality gaps and of prioritizing which gaps to focus on, given the constraint of scare resources. TestQual can also serve as the structured guideline to train new hires of testers in terms both technical aspects of delivering the quality testing service and social aspects of interaction with other important stakeholders in the project.

The second practitioner contribution of this paper is an implementable tool that allows IT and business managers to quantify the level of service quality offered by their testing function. If managers adopt the testing as a service perspective, they need to be able to measure the level of service provided by the testing organization. TestQual tool can be administered to the testing service receivers in the development processes, thus the level of service quality of the testing function can be quantified. Once the perceived level of service or ZOT of service quality is quantified, managers can identify areas of deficiency of the service organization. ZOT can particularly provide richer information than perception-only or expectation-minus-perception scores. Accordingly, managerial corrective

measurements can be taken to address the testing service deficiency or failures.

The testing as service paradigm help the testers, developers and users recognize that they are dependent on one another in a win-win or lose-lose relationship. When the testers have a clear sense of the expectation and perception on them, they can be more likely to provide adequate and even superior services to their “customers”, thus the project team as a whole is one step closer to successful completions of the quality product. Management must support the testing effort not only with funding, but also make the effort to build and promote testers-as-service-providers climate in software development project team.
<table>
<thead>
<tr>
<th>Aspects</th>
<th>Definitions</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.</td>
<td>• Suitability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Accuracy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Interoperability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Compliance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Security</td>
</tr>
<tr>
<td>Reliability</td>
<td>A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.</td>
<td>• Maturity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Recoverability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Fault Tolerance</td>
</tr>
<tr>
<td>Usability</td>
<td>A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.</td>
<td>• Learnability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Understandability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Operability</td>
</tr>
<tr>
<td>Efficiency</td>
<td>A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.</td>
<td>• Time Behaviour</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Resource Behaviour</td>
</tr>
<tr>
<td>Maintainability</td>
<td>A set of attributes that bear on the effort needed to make specified modifications.</td>
<td>• Stability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Analyzability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changeability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Testability</td>
</tr>
<tr>
<td>Portability</td>
<td>A set of attributes that bear on the ability of software to be transferred from one environment to another.</td>
<td>• Installability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Replaceability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Adaptability</td>
</tr>
</tbody>
</table>

Table 1: ISO standard ISO 9126
<table>
<thead>
<tr>
<th>Aspects to developers</th>
<th>Tangibles</th>
<th>A set of attributes about the physical testing environment, and testing deliverables and documentations</th>
<th>Implementable Infrastructure and testing tools</th>
<th>Quality of test processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability</td>
<td>Ability of testers to perform the promised service dependably and accurately</td>
<td>Coverage Consistency</td>
<td>Maximize bugs correctly identified, minimize false positives</td>
<td></td>
</tr>
<tr>
<td>Responsiveness</td>
<td>Willingness to help the developers at critical times and provide prompt service</td>
<td>Readiness Promptness</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assurance</td>
<td>Knowledge and skills of testers and their ability to inspire trust and confidence of the developers</td>
<td>Capability Trustworthiness Integrity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rapport</td>
<td>Caring and individualized attention to the developers</td>
<td>Courtesy Communication Conflict resolution Harmony</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dedication/Excellence</td>
<td>The initiative and creativity of testers in providing service</td>
<td>Proactiveness Creativity Dedication</td>
<td>Not just show effort but suggesting a solution to developers</td>
<td></td>
</tr>
<tr>
<td>Service in relation to software product</td>
<td>Functionality</td>
<td>Attributes confirming the ability of the software to perform the intended tasks</td>
<td>Suitability Accuracy Interoperability Compliance Security</td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td>Attributes confirming capability to maintain performance level over conditions and time periods</td>
<td>Maturity Recoverability Fault Tolerance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usability</td>
<td>Attributes confirming effort required for learning and using by targeted users</td>
<td>Learnability Understandability Operability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td>Attributes confirming the performance and resource intensiveness of software given stated circumstances</td>
<td>Time Behavior Resource Behavior</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maintainability</td>
<td>Attributes confirming modularity of design and effort needed for modifications</td>
<td>Stability Analyzability Changeability Testability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Portability</td>
<td>Attributes confirming transferability of the software to multiple environments and configurations</td>
<td>Installability Replaceability Adaptability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalability</td>
<td>Attributes confirming ease by which software can be enhanced and expanded</td>
<td>Flexibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirect service to end users: Confirmation of the expectation of users of the</td>
<td>Tangibles</td>
<td>A set of attributes about the physical environment of software use by end users, software deliverables/output, and software documentation quality</td>
<td>Aesthetics</td>
<td></td>
</tr>
<tr>
<td>Assurance</td>
<td>Testers ensure the software performs as expected</td>
<td>Reliability/security Validation to new users</td>
<td></td>
<td></td>
</tr>
<tr>
<td>software quality</td>
<td>consistently, accurately and securely according to the expectation of users</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Agency Oversight</td>
<td>Testers are agents of users to control the quality of software development</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Satisfaction</td>
<td>Testers ensure software successfully meets and exceeds the needs of the user</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha testing</td>
<td>Beta testing</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: TestQual

References:

1. Chilton et al. (2005)
Applying TDD to Web-Application Development – Does Variety Help or Harm?

Wolf-Gideon Bleek, Sebastian Pappert
University of Hamburg
Dept. Informatics, Software Engineering Group
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
bleek@informatik.uni-hamburg.de, Sebastian.pappert@gmail.com

Abstract

This paper presents a case study which covers experiences gained from a Web development project employing agile techniques. Web development involves a number of technologies introducing a variety which is difficult to handle. Beside other agile techniques the project’s participants followed test-driven development (TDD) in the project’s various technology areas. TDD is used in addition to unit testing to positively influence the design of program artifacts. In doing so, they encountered problems, pitfalls and failures and developed strategies to have an overall test-driven approach. We report on the stony way exercising Test-Driven Development in a typical technology mix of a Web development project to learn about obstacles surfacing while pursuing design through Test-Driven Development. The case study reveals three strategies that programmers follow: avoiding to subclass from a framework, moving code to easy testable classes and keeping hard-to-test artifacts small.

2. The Context – A Typical Web Development Project

To better understand the context of this problem area we firstly draft the object of this case study, the development project. Secondly, we introduce the underlying software engineering concepts relevant for this research. And thirdly, we will outline the development environment.

2.1. The Project

The experiences reported in this paper centre around a development project of a Web application1. This application has been developed since 1999 firstly in PHP. During that period it was not developed using unit tests or explicit agile techniques. Due to the popularity of Java and the lack of architectural tools that can help continuously monitor and refactor the old system we initiated a migration to the Java Web platform.

In this new Java Web development project (called the “migration project”) we incrementally develop a Web-based Java application to gradually replace modules of the old system while running concurrently. The development project adopts an agile process which is mostly inspired by Scrum [2] and XP [3] using stories and tasks, stand up meetings, pair programming, shared code etc. and most importantly tries to develop by sticking to the Test-Driven Development rhythm (see [4], [5]).

1 The CommSy project can be reached at www.commsy.de.

116
The Java Web development project started out with a group of approximately 20 people. It was set up as a migration project with the long term goal to replace the running PHP system. As an initial decision Java technology was selected to create the new system. Main reasons for this technology decision were, among others, the rich set of available tools to monitor and ensure quality of the code base during development and an already existing knowledge among key project persons.

2.1.1. The Team Structure.

The project is carried out as an open source project [6] that is supported by a number of different groups. Among these groups are students participating in class projects, researchers and paid programmers. All of these meet at least twice a week working together. One key characteristic of this development group is that besides a constant group of about four people there is a constant fluctuation of team members. The typical length of stay in the development project is approx. one year. The alternating participants usually have little to no experience with TDD but do have some experience with agile development methods (e.g. pair programming).

2.1.2. Basis for Evaluation.

This research bases on static code analysis and individual code analysis and is supplemented by the experiences of the project’s participants gained through interviews. Data on the Java project has been gathered since 2004. For evaluating the source code we have repositories available that allow retracing the actual source code configuration of each development step. We are therefore able to investigate in detail how the source code developed over time. Measures are calculated according to relevant points in time of the development project (e.g. before or after a major refactoring). The repository is available for both the PHP project and the Java project. However, results in this paper concentrate on the Java development only.

2.2. Related work

This paper focuses on an agile development process and evaluates the relation between Test-Driven Development, its effects on design and the number of technologies involved.

2.2.1. Unit Testing.

Unit testing [7, p. 239], [8] in general is an essential part of a basic quality assuring procedure. For each source code unit developers write a test that runs automatically. This tests all public features of the unit and will break if any feature changes behavior. Each unit test complies with a framework so that all tests can be collected and run sequentially. Due to this automation process we are able to run tests each time required (e.g. before a refactoring or code is added and afterwards).

In accordance with Beck the following set of general criteria applies [10]:

- Isolation: A unit must be tested solely without using other operative code. No side-effects must occur. The order of testing must not matter.
- Simplicity: Writing tests must not complicate things in terms of programming language or technology involved. The development environment must (seamlessly) integrate writing and executing tests.
- Automation: Tests must run automated without user intervention.
- Speed: Tests must not consume much time in order to be executed frequently.
- Learnability: Developers must easily understand tests and their frameworks.
- Design: The technology must to a large extent allow test-driven development to design the interface of an artifact.

2.2.2. Test-Driven Development.

Test-Driven Development is an agile practice that goes beyond unit testing. As the name suggests it focuses on a process where tests drive the development. But despite its name, TDD is rather a design technique than a testing technique. This is accomplished by writing the test first and reacting by implementing the required operations in the artifact [11]. In consequence one designs the interface of that artifact reflecting the requirements of the test. In a typical development cycle, at first the test requires functionality that does not exist. The test cannot compile at this time and after fulfilling the syntactical requirements the test will fail during run. After that the implementation is written to suffice the test. Finally refactoring cleans up the artifact under development (see [4], [5] for an example of the TDD cycle here called the “TDD rhythm”)

We favor TDD because it designs the interface of a program artifact from the test’s perspective, reduces or avoids YAGNI\textsuperscript{2} [3, p. 42], implements and tests

\textsuperscript{2} YAGNI: You Aren’t Gonna Need It
code in small steps and the project’s repository consists of well tested classes. In a complex project or an environment where new people are introduced on a regular basis it is of importance to have interfaces that are small and/or easy to understand. TDD supports achieving these kinds of interfaces. Furthermore, we feel that the overall design of the artifacts’ interfaces has a better quality [11] in terms of cohesion (high) and coupling (loose).

TDD can be practiced at initial development as well as during further development of a system. It should be well applicable for Web applications. Pursuing TDD only needs a unit testing framework that supports running tests repeatedly at the developers’ workstation.

2.3. A Typical Variety of Technology

Web development projects tend to cover a significant number of technologies. This is not in contrast to other development projects. However, with the introduction of Web technologies the variety spans on multiple levels. Not only exist different formal languages for program and data description (Java and XML), there are also different programming languages (i.e. Java, JSP, EL), different control flow and state mechanisms (i.e. Servlets, Java objects) and the complex standards for visual presentation (i.e. HTML, XHTML, JavaScript).

While on the one hand this variety helps to cope with ambitious requirements it also demands many qualifications from all project participants and poses a problem for a uniform testing strategy.

Taking aside technologies for special requirements and of course XHTML which is standard in today’s Web applications, the Java platform alone introduces Servlets and JavaServer Pages (JSP), the Expression Language (EL) within the JSP and Tags (which utilize JSP and XML). All of these can be referred back to pure Java code but they certainly have their reason to be available.

2.3.1. Servlets.

Servlets are Java’s way of directing the flow of control into the hands of the application. Therefore, the application’s entry point is an overridden method in a subclass of the Servlet class. This is straightforward for an experienced programmer of object oriented languages but has its pitfalls. Although you can implement this subclass stateful it is not recommended to do so because the container re-uses the Servlet objects in different contexts. Therefore, a set of objects handed over through the method’s parameter called request, response and session represent the state. These objects model the state of the application since multiple instances can run at the same time utilizing the same Servlet object.

2.3.2. JavaServer Pages.

While JSPs are written in an XML style language they may also contain segments of Java code. This mixture makes these documents hard to understand since one segment can reference variables that have been introduced in a previous segment at the beginning of the document (we call it the “mixture problem”). These properties of the technology can be better understood when looking at what happens behind the scenes: The Servlet container translates the JSPs into pure Java code implementing a Servlet subclass. Then it compiles the code in the background and the control flow continues after that in the compiled subclass. Therefore, JSPs behave like Servlets. This has to be kept in mind when writing JSPs since each Java snippet embedded in the XHTML document can access a common namespace and through it the Web application’s context.

2.3.3. Expression Language.

The Expression Language (EL) is an enhancement of the JSP environment that helps resolve the “mixture problem” stated above. Using the EL avoids introducing Java code snippets into the XHTML document. The EL gives access to Java objects and Java operation calls. However, its syntax is cryptic (so that it complies with XML requirements) and EL implies that data transfer between code in the Servlet and the JSP is handled by Java Beans (but this does not pose a problem since it is best practice). However, the naming scheme and identifying the corresponding elements is not obvious.

2.3.4. JSP Tag Libraries.

The tags introduced into the JSPs are a way of reducing duplicated code. They resemble subroutines and can be written using JSP syntax and EL or they can be provided as Java code. These tags are listed in an XML document to be available for reference.

2.4. The Development Environment

---


The integrated development environment (IDE) used for the case study’s project is Eclipse\(^6\). This allows programmers to work effectively with the technology mix outlined above. Plug-ins exist for the chosen technologies and support working with them.

The project’s integration server\(^7\) not only offers a shared code repository but also serves as a build and test server. Each check-in triggers a complete compile and test cycle and installs the new version as a test system. Reports on each build cycle (regarding build steps, tests and test coverage\(^8\)) are available through a central development Web server.

### 3. Applying the TDD-Rhythm

In the case study, project development follows agile development practices, especially unit testing / test infected programming and aims at implementing the TDD rhythm as depicted by [4], [5]. This cycle enables people to quickly generate functional code and they achieve a high quality design. But as pointed out in Section 2.3 in this project participants have to deal with a number of technologies (program artifacts) typical for Web development projects besides pure Java classes. The most prominent among these were:

- Database Mapping
- Servlets and Filters
- Servlet Container Classes
- JavaServer Pages
- Custom Actions (Tags)

These different technologies hinder the team members implementing the TDD rhythm. The first problem is that unit testing frameworks are not as easily available for these technologies as for pure Java code. Secondly, the different unit testing frameworks do not integrate seamlessly into the development environment. Thirdly, different handling mechanisms make it cumbersome to run tests as often as necessary for TDD. And finally, even though testing frameworks are ultimately available, they may not directly support a TDD strategy. In the following subsections we will systematically investigate how the criteria listed above (section 2.2.1) apply to the different kinds of technologies that we encountered in the project.

#### 3.1. Pure Java Classes

Pure Java classes are a well supported area of TDD and provide the basic reference line for other technologies and their artifacts. This type of technology forms approx. 50% of the system in this case (not considering JavaServer Pages, see below in section 3.4) and is mainly used to implement domain values, data items, and utility classes. In this project, the test coverage for this type of technology is nearly 58%. However, not all tests have been created within a TDD rhythm.

All of these classes can be tested in isolation. Tests are written in the same language within the development environment using the JUnit framework [8], [15]. JUnit is fully integrated into the Eclipse IDE. This provides basic automation at the workstation and it is quick enough to be run repeatedly. Except for the usage of JUnit, developers had nothing additional to learn. The interface of classes can be designed by applying the TDD rhythm.

#### 3.1.1. Experiences

The type of technology and the type of classes written facilitate identifying the corresponding test. It is also clear what to test and how it is to be tested.

It is imperative that tests are easy and quick to run in order to keep programmers performing the rhythm. If tests are an obstacle during the development

---

\(^{6}\) Eclipse IDE: http://www.eclipse.org

\(^{7}\) In this case CruiseControl is used: http://cruisecontrol.sourceforge.net/

\(^{8}\) Test coverage is metered by Cobertura: http://cobertura.sourceforge.net.
process these tests are usually the first thing to be dropped.

The project’s experience is that TDD can be applied smoothly but new team members only reluctantly adhere to it.

3.1.2. Reflection and assessment.

The analysis of these classes’ tests shows that even though a significant number of JUnit tests exists and the test coverage is good some tests which were intended as pure unit tests were in fact integration and acceptance tests. One significant problem here is that tests are not written for isolated classes (e.g. by utilizing mock frameworks).

Besides the technical qualification for writing a test with JUnit, programmers also need training in how to write a meaningful test. Neither TDD nor the unit testing framework can assure this.

Regarding the design aspect of TDD, programmers can make use of the full potential of TDD. There were no restrictions on the interface of classes.

3.2. Servlets and Filters

Servlets are the centre of Java Web applications and control the interaction with the user. They consist of two parts: a class file implementing the behavior and a configuration that makes the Servlet accessible through the Servlet container and configures the application. This configuration depends on the Servlet container. For the Tomcat Servlet container the configuration is stored in XML files. Filters are a light-weight realization of the Intercepting Filter pattern [16, pp. 144-165] and very similar to Servlets in implementation and configuration.

In this particular case, Servlets and Filters are less than 5% of the system. Of these classes the coverage is only 20%. Tests are written using the Cactus testing framework [17] following a JUnit mechanism but allowing the object’s execution in a Servlet container environment.

3.2.1. Experiences.

A Servlet object relies on a request, a response, and a context object which the container provides. This implies a substantial initialized object structure to perform the test. Either that object structure needs to be replaced by mock objects or the tests have to be executed in a Servlet container using a framework like Cactus. The project follows the latter approach.

Regarding simplicity tests can be written in the same language in which Servlets are programmed. However, it involves introducing a new testing framework which should comply with established standards: writing and executing the tests needs to be integrated in the development environment. In this case, tests can be automated because they are also JUnit tests making them part of the project’s test suite.

Due to the fact that part of the test is executed on “the client side” and another part is executed on “the server side” tests get more complex but are still understandable.

There are multiple levels of problems testing the Servlets: the container characteristics of the request and response objects, the sequence of operations in the main Servlet, and the session representing the status of the calling context.

3.2.2. Reflection and assessment.

This particular project adopts a Servlet structure to work as a single entry point into the Web application serving as a collecting instance that hands over to other (non Servlet) parts of the system. This complies with the MVC2 pattern using a so-called “front controller” [16, pp. 166-180]. The goal is to reduce the Servlet’s function to collecting the necessary information, hand them over to a transforming instance and collect the result and output for transfer to the user.

Even though Servlets are a centre point of Web application development a common approach for Web applications is to avoid them. The strategy is to keep the number of Servlets of an application to a minimum.

Most of the operations of a Servlet are inherited. Application specific actions take place in one or two methods. Testing these methods is an obstacle because there is a lot of context to initialize and reproduce in order to perform the test. Moreover, the operation usually gets complex during development.

Because the interface of Servlets and Filters is predefined, in this respect TDD cannot contribute to the design of the interface but to the quality of the code. The Servlet’s interface does not evolve through development since there are only dedicated hook methods to specify. A good strategy is to move code from these classes to places where it can be tested properly and classes’ interfaces do matter and can be designed.

Instances of the Servlet classes are not first-class objects. They are not allowed to have instance variables since the Servlet container may re-use them, they run non-deterministic in different threads, and are challenging to test in full isolation. The latter is due to the fact that complex request, session, and response objects have to be initialized to enter a Servlet method. The major drawback is the single method to which control is handed over when the

9 http://tomcat.apache.org/
Servlet is entered. This method may degenerate to a huge main method of the application (the code repository of the project shows this effect in older versions) encapsulating most of the functionality and hiding it from dedicated testing.

Servlet programming requires access to the requestor session object which hands over vital information about the current status of the application as well as the requested action and its parameter (they replace the instance variables one would typically find in classic objects). The processing in the Servlet’s method retrieves and evaluates elements and stores new elements. While this is normal programming style within a Servlet environment it poses a significant burden on testing because it obfuscates the interrelation of these elements. Implicit preconditions and interdependencies occur: e.g. a value needs to be available in the session object before another can be calculated. While in classic code this can be determined and resolved at compile time in a Servlet the situation will arise at run-time. This does not support a TDD style because dependencies are rather in the sequence than reflected by the class’ interface. In consequence, programmers should avoid putting application specific code into these classes.

3.3. Servlet Container Classes

Servlet container classes (SCC) are essentially pure Java classes that read or manipulate the Servlet container’s context objects request, response, and session. These classes serve as a mediator between the Servlet classes and the pure Java classes. The coupling with the Servlet API is existent but not as extensive as the Servlet classes are coupled with the Servlet API [18]. SCC make use of types introduced by the Servlet API but they do not rely on inheriting classes or interfaces of the Servlet API. Approx. 30% of the system is in this category with coverage similar to the Servlets.

3.3.1. Experiences.

Since the syntactical design of SCC is equivalent to pure Java classes all experiences of the developing process of pure Java classes apply to SCC. The testability of these classes solely depends on the design. Unlike Servlets and Filters, SCC are not bound to be run inside a Servlet container. However, the usage of context objects requires at least a minimal implementation of the context classes that the function under test operates on. The context classes are rather complex and therefore the simplest solution is to use the already introduced framework Cactus. In this case study the response object is never manipulated in SCC which reduces the complexity of SCC compared to Servlets. The request and the session objects can be seen as a secondary interface. Without limitations for type and amount of the data in the context objects, tests describe the secondary interface. Developers can reuse the previously learned knowledge of writing tests for Servlets and the tests are seamlessly integrated into the JUnit framework.

3.4. JavaServer Pages

JSPs are basically XHTML documents with special tokens that allow insertion of programming constructs. During run-time these JSPs are transformed to Servlets. While the XHTML text is translated to println commands on the response stream, the inserted statements and code are directly passed through. From the programmers’ point of view JSPs look like a completely different programming environment since they have to utilize a different formal language and have a (prepared but implicit) environment which they can (or must) use to access data (e.g. for outputting values).

3.4.1. Experiences.

In terms of testing we need to investigate the question whether a JSP is a proper unit for testing. We need to identify the public operations, to find out what interface actually is developed or, with respect to TDD, designed. JSPs certainly need testing and can be seen as a unit of the whole application. However, the artifact has no explicit interface or – if seen as a Servlet – there is a predefined and fix interface.

Moreover, a JSP can hardly be tested in isolation. Like Servlets it needs an environment which is provided by request, response and session objects. Bean objects need to be available at compile time.

Interaction with JPSs is possible through a Web request which triggers the compilation and returns an XHTML stream. This stream is either an error report that indicates problems during compilation or runtime or it is the generated XHTML which can then be checked for syntax and content.

Due to the request, the compilation, and the stream result testing JSPs is significantly complex. The test may break for various reasons and unit tests cannot easily be integrated into the overall testing framework.

A major question – as with Servlets – is whether JSPs fall in the area of artifacts that are reasonably developable through TDD. Since there is no interface to design most things to test are standard with all JSPs (see above).

Rudimentary tests for JSPs can be automated. The test can request a page described by a JSP and
evaluate the XHTML stream. If errors exist the test may fail. However, this kind of test is significantly slower and requires an application server to run. It is also hard to extend for tests that go beyond these characteristics.

If testing is performed there are multiple levels of what can be tested: there is XHTML testing regarding the syntax, testing of the control flow (e.g. are iterations over collections performed correctly?), and whether all necessary data are available for processing (e.g. is a specific element necessary for display provided by the session object?).

Due to the nature of JSPs we need to investigate if it is at all reasonable to pursue a TDD approach. How could TDD help during development, since the strong sides of TDD are the design of a units interface and to ensure correctness against the test? We do not see how TDD can contribute to e.g. testing the initialization of a variable. Typical problem categories comprise the existence of getter operations and whether a Bean has been properly initialized.

Since we have a hard time formulating preconditions for a JSP we are hardly able to design tests in a unit test fashion. In the evaluated project a design rule to keep algorithmic code away from the JSP (and from Beans and tags also) has been developed.

3.4.2. Reflection and Assessment.

As we have shown above, JSPs do not have an explicit interface that can be tested or even designed. Still, these units of the application need testing on various levels but a test-driven approach is not the adequate way of development for these elements. Since JSPs do not offer separate methods for which we can assign pre- and post-conditions the cause for or location of an error cannot be identified easily.

Unit testing beyond syntax errors is hard to achieve (while integration and acceptance tests are easy). To reach a maximum level of unit testable elements it is important to not let algorithmic code get into JSPs. This should be handled by classic Java classes and elements that have a testable interface.

3.5. Custom Actions

The JSP specification provides developers with a basic set of standard actions to enhance JSP pages with logic hidden behind XML syntax. These standard actions can be extended by custom actions either by creating a Java class that follows the tag extension mechanisms of the JSP specification or by writing tag files in a JSP syntax.

3.5.1. Experiences.

In particular, the implementation of custom actions as Java classes confronts developers with two points of view for the interface. First, the Java interface for the class file is only of interest to the Servlet container and bears no possibilities for design decisions. Second, the XML interface is important for the actual user of the action. The Cactus framework provides mechanisms similar to the one for Servlets to test the Java interface of custom actions. It is also possible to test the XML interface by embedding the action in a simple JSP and utilizing the testing mechanisms for that type of artifacts. This approach integrates easily in the development process of the other units tested with Cactus but comes with the cost of splitting test cases into two separate files. The input for a test case is defined in the JSP whereas the output is verified in a Java file. A different approach would be to use a framework that allows writing test cases for custom actions completely in the JSP syntax, e.g. TagUnit. TagUnit does not work together with the JUnit framework and therefore does not fit into the established development process. It can be executed automatically in an Ant script but unlike Cactus TagUnit is not supported by IDEs and the integration into a build server would need manual adjustments.

3.5.2. Reflection and Assessment.

As we have shown above, similar to Servlets, custom actions do not offer any degree of freedom with respect to the interface that is to be fulfilled. Therefore, the major strength of TDD is not applicable. This only leaves the question of how the best testing approach can be achieved. While implementing custom actions in JSP syntax is compact and easy to write it requires discipline and understanding when it comes to realizing the test. Additionally, test coverage cannot be measured easily. Therefore we would strongly suggest implementing custom tags in Java.

4. Evaluation

In this section we will generalize the experiences of the different technologies collected in the empirical study and take a look at the concepts. After that we distinguish different levels of testing and finally explicate the contribution of the different testing approaches to these levels.

4.1. General Problems

http://tagunit.sourceforge.net
Web applications introduce a set of technology that complicates testing and in particular TDD thus not allowing taking advantage of the design contribution TDD provides. As shown above, each technology requires a different view on testing, a different unit testing framework and poses new challenges on how to develop code.

On top of that, even with the proper testing framework for each technology the approach to testing is slightly different. Programmers are required to learn the differences and have to implement different strategies of testing (local unit, client-side, server-side).

For example testing Servlets and Filters has not been well understood by programmers in this project in comparison to pure Java classes because:

- the designation of Servlets and Filters is not clear,
- programmers are confused by the control flow and the change of means of expression (application server, Filter, Servlet, JSP, Java class),
- in contrast to “normal” unit tests the number of test methods is increased (up to seven),
- Servlets and Filters make use of implicit objects that are handled differently in each of them,
- execution during tests takes place on the client and on the server side (people have to know that while writing the test and analyzing a broken test),
- testing “normal” Java classes can be run by a keystroke or mouse click while testing Servlets and Filters requires a fully set up and running environment.

As a consequence, these tests are reluctantly implemented and operated.

Not all testing frameworks allow testing the unit in isolation. As shown by the JSPs we are not able to mock all objects of the JSP’s environment (e.g. other JSPs included, custom actions).

With respect to simplicity it is ambiguous to introduce a number of additional testing frameworks. While on the one hand they enable testing of other technologies they on the other hand demand additional learning and may confuse due to (slightly) different handling.

At least the aspect of test automation can be accomplished for all technologies discussed. This is currently not an open field. Also, speed is not a relevant topic even though testing JSPs imposes additional compiling.

Design with respect to the interface of an artifact is a weak topic in the field of Web application development and TDD because most of the interfaces of the technologies introduced are fixed. To access functionality interfaces have to be met and developing them further is counterproductive.

### 4.2. Levels of Testing

Even though all testing frameworks make use of the unit testing approach they do not – in fact – have the same level of testing. The least common denominator of all frameworks simply is that they test a single artifact. For this artifact test methods can be specified. Whether these test methods provide tests for each method of a class or offer different request types for a Web page is not specified further.

In this project we have identified at least three different levels of testing:

**Method by Method Test.** Java classes allow tests that provide a test method for each (relevant/public) method of the unit under test. TDD can govern the process in a way that influences what methods emerge and how their signature is designed.

**Single Entry Test.** Some tests, like e.g. JSP tests, simply request the result of that specific unit to evaluate the result. In some instances the request has parameters that may influence the result. A common theme to these tests is that there is only one entry point.

**Disguised *-Test.** While many of the so called tests come as unit tests they aren’t in fact unit tests. They bear the name to signal that they can be integrated in a JUnit run. HttpUnit, for example, is not a unit test in comparison to JUnit simply because it does not test a unit in isolation.

### 4.3. Coping with Web Technology – Overcoming the Major Problems

Since the case study identified a number of problematic characteristics of the technologies involved in terms of TDD it also outlines strategies to minimize or overcome these problems.

**Strategy 1: Avoid Subclassing the Framework.** Keep the number of classes that subtype from a framework’s class as low as possible because the interface of such classes is pre-defined and designing it further is not intended. Since testing Servlets is not easy and TDD mostly not applicable they should – from the standpoint of TDD – be avoided. This has been achieved in the project by reducing the number of Servlets to a minimum. Currently the project contains five classes that inherit the Servlet interface.
This will be reduced to three leaving one actual Servlet and two super classes.

**Strategy 2: Move Code to Testable Classes.** The second and corresponding strategy is to introduce objects of Java classes that are not joined with the Servlet class. Move as much code as possible out of artifacts that pose a testing problem and implement using pure Java classes. Delegate from the special classes as early as possible. In this way, full TDD can be applied: the object has a relevant interface, the object has a state and basic test principles apply.

**Strategy 3: Keep Hard-To-Test Artifacts Small.** Keep the size of artifacts that cannot be tested small (in contrast to a few large ones). As discussed above, testing JSPs is a similar problem as with Servlets but with the added handicap that JSPs are typically not segmented into methods. While for Servlets a valid strategy can be to avoid them this may be more complicated for JSPs because they are a good way for formatting output. Other solutions, like transformation frameworks introduce an ample amount of learning and complexity.

Therefore, the project’s strategy is to keep JSPs as small as possible to have the unit under test manageable. In this case study there were 15 JSPs with an average size of 149 and a peak of 310 LoC before refactoring. After following this strategy the number increased to 40 with an average size of 72 and a peak of 190 LoC.

5. **Discussion**

Analyzing the project has shown that each new technology introduced into the development process is likely to require a specifically tailored test framework. For those technologies where frameworks are available they have at first been used to implement tests but not all have proven an optimal solution. For some technologies sufficient testing support simply is not achievable with acceptable effort. In any case, special knowledge is necessary not only for handling the technology but also for writing the tests.

For a project team whose goal is to have sufficiently tested artifacts for all technology areas and establish a good design for all artifacts strategies have to be established that can be used during software development. The strategies presented above were identified through the project’s analysis. Their common theme is to reduce the code written in technology that is hard to test and move code to technology that offers an easy to use and established testing framework.

One deficiency of the depicted collection of strategies is that one cannot get rid of the different testing frameworks completely. The team still needs to have the knowledge about the frameworks and their testing strategy available in the project.

What has been achieved? Rather than pursuing a way to make TDD available for all technology areas the study shows that it is advisable to carefully investigate whether new technology needs to be introduced. If so, it needs to be determined how much must be written using this technology and how much can be moved to “normal” code.

In this respect this research has identified in what areas new technology hinders designing the artifact through TDD. It also presents a viable strategy to elegantly reduce the restraining effects of new technology. In total, by concentrating on the artifacts that can be designed by TDD developers can make the most of it.

Testing is not the only domain in which moving to a well established base is helpful. There are a number of tools available for pure Java code which is hardly found for the other technologies. Examples are, to start with, test coverage, static code analysis (measuring, identifying bugs), dynamic code analysis (profiling). Even though some areas may be covered in the future it will always be easier to work with a main technology in contrast to a set of different technologies.

The approach outlined in the evaluation can possibly be transferred to other programming languages and to other areas of software development. However, this research is limited to the Java environment and a single Web development project.

6. **Conclusion**

In this paper we have outlined a case study of a Web development project in Java. This project makes use of the standard Java technologies like Servlets, JavaServer Pages, the Expression Language, Tags and Custom Actions for implementing a Web application. With a test-driven approach one needs special support in the area of unit testing. The case description shows that each technology besides pure Java classes has its drawbacks and requires special support (in the form of a framework) to implement tests and in consequence profit of the design capabilities of TDD. Some drawbacks hinder a TDD approach and some unit testing frameworks do not allow literal unit testing.

The case study shows approaches on how to cope with these deficiencies. The first approach centers on avoiding special Web related technology. Servlets are
an example where reducing the number of classes and reducing the lines of code has effectively allowed for more testing. This is, in part, achieved by the second strategy that accompanies the first. Moving code to separate testable classes. JSPs are an example of another strategy. They can practically only be tested as a single entity. To make testing more precise these entities have been reduced in length. The number of artifacts more than doubled but it had a positive effect on modularization and testability and has had hardly any effect on the total number of lines.

We can conclude that the key message of this case is a most obvious one: reduce the number of technologies to simplify testing during development. In addition, both strategies, avoiding problematic technology and splitting hard to test artifacts in small pieces, are part of a feasible way that leads to a better testable architecture. This also allows circumventing the problem of driving the development of artifacts that do not have an interface (in this case JSPs). Since designing the interface in terms of methods is not applicable, at least small artifacts stand in.

With regard to the design of artifacts the message is to focus on established artifact types and avoid elements with the characteristics shown here.

The identified approach combining these three strategies leaves the problem that by utilizing a set of problematic technologies there is always the chance of growing artifacts of these types that evade an established and practical testing strategy. Further research on this needs to be done.

References

Defect Management System for Software Development Process

K V Dinesha,
Professor, I.I.I.T Bangalore
26/C, electronic city, Hosur Road, Bangalore 560 100
Email : dinesha@iiitb.ac.in; dineshakv@yahoo.co.in

Abstract

Defect management forms an integral part of every quality system. Defects tell about the health of the system. One of the main focuses of SEI CMM level 5 is defect prevention and use of defects for improving software process. In this article we discuss defect management life cycle from a practitioner’s perspective. We hope that this article provides background, guidelines and framework for setting a defect management system for the software system environment of an organization. This article discusses what defects are and their different aspects, organization attitude towards defects, different methods of analyzing defects, how defect analysis and classification can help in improving the ongoing project and organization process, and what to expect from defect management software. The objective of this article is to assist an organization in building process part of its Defect Management System.

1. Introduction

Defect management forms an integral part of every quality system. In short, defects tell about the health of the system. One of the main focuses of SEI CMM level 5 [1] is defect prevention and use of the feedback in the form of defects for improving software process. In this article we discuss defect management life cycle from a practitioner’s perspective. We hope that this article provides background, guidelines and framework for setting a defect management system for the software system environment of an organization. This article discusses what defects are and their different aspects, organization attitude towards defects, different methods of analyzing defects, how defect analysis and classification can help in improving the ongoing project and organization process, and what to expect from defect management software.

Defect Management System has two components: People and Processes. The quality of work done by people is closely related with the culture of the organization and the quality of processes is tied to the extent of automation and hence the contribution of Tools to the process. In this article we focus on the processes associated with Defect Management System. The processes in an organization to some extent depend on the culture of the organization. We have discussed the culture aspect within the limited context of defect management processes. We have not dealt with the Tool aspect.

At present, the best practices associated with software defect management system are scattered in many books, journals and practices followed in different organizations. The objective of this article is to provide a set of best practices from which an organization can build its own processes for a Defect Management System. The article consolidates the best practices followed by many of SEI CMM level 5 companies and best practices suggested in books and journals.

1.1. Terminology

A software artifact is the output of (or input to) a process in software production. Different processes correspond to different phases (like requirements, analysis, design, testing, coding, integration, maintenance etc) of software development. From this perspective program source code is one of the artifacts, an output of the coding phase.

In the context of defect management system, we broadly encounter the terms defects, bugs and errors. In this article we use the terms errors and bugs synonymously. People use the term error (or bug) in the context of mistakes found in the source code. When we talk of defect, we mean the broader context, i.e., mismatch between what is expected and what is achieved in any of the software artifacts [2]. The following are a few examples of defects:

1. A wrongly captured requirement has a defect in requirement document.
2. A design which does not deliver the specified requirements has a defect in design document.
3. A source code document which does not adhere to the coding standards, naming conventions etc., has a defect in source code.
4. A system which does not satisfy the required non functional specifications (like performance, reliability, maintainability, etc.) has defects in its architectural / requirement artifacts.

2. Attitude towards Defects

It is impossible to build any system with zero defects [3]. This has some important implications.
1. All the software applications have defects no matter how thoroughly they are tested.
2. When we certify that an application is ‘defect free’ it means that our testing team (or test suite) could not find the defects in the system.

In the context of the above, the following should form the basic organization culture.
1. To err is human.
2. To learn from error is also human.
3. To learn from other’s errors is divine.
An organization should show a healthy and scientific attitude towards defects. The organization culture [1, 4, and 5] should nurture the following:
1. Analyze every defect from customer satisfaction perspective.
2. Accept that it is natural to do mistakes.
3. Have a mechanism to report defects.
4. Assure to the employees that defect analysis is not performance analysis of the developers. However, a defect identified by an independent third party or in the field could be used as a measure of quality of work and performance of the employees.
5. Treat the defects like a scientist or a doctor. An organization should have a system to classify defects and learn from the defects so that such types of defects are not introduced in future by anyone in the organization.

3. What to do when one discover defect [6]
The first thing one does is to fix the defect. However, if we want to learn the most from the defect, we need to do a few more things:
1. Enter the details of the defect in a tool.
   Typically the following details are captured in a tool
   a. Description of the defect
   b. The defect location [name of the module/component]
c. The priority for defect fixing: [must be fixed urgently, nice if it can be fixed in current release, can be fixed in the next release- an item in the list of known bugs]
d. The Severity of the defect(impact of defect on the system)
e. The development phase in which the defect is discovered
f. The phase to which the defect belongs : [requirement phase, analysis, design, etc]
g. The ‘defect finder’ and the ‘defect fixer’
h. The software artifacts that were changed to fix the bug
i. A description of the changes done to fix the bug

2. Look for similar types of defects in other parts of the system:
Example 1: Let us assume that while reviewing an architecture document we find a defect related to mismatch of interface between a legacy system and the new component to be developed. We should also look for similar mismatches between other interfaces and the new system.
Example 2: Let the GUI for a system be in German. We find that there are some spelling mistakes in the GUI labels for one screen. We should look for this type of error in all other screens (at least those screen developed by the same group). Such defects can be minimized by organizing a training program on German language vocabulary around GUI words.
Example 3: We find that in a source code a particular flag was wrongly initialized. Look for initialization of flags in all programs developed by the same developer (and developers with similar background).

3. Look for defects hiding under defects: Many times correction of a defect enables the flow to the parts of the system which were not covered earlier due to the existence of the defect. This implies that the defects in those parts were not discovered in the previous test run.
As an illustration, let us consider the example 1 above. When we correct the interface mismatch, the flow happens in all the parts of the system which correspond to correct interface. It pays to make a review of
those parts based on the experience till that time.

As another example, consider the example 3 above. When the flag is corrected, the control flows through the parts corresponding to the correct value of the flag. It makes sense to review these parts to uncover the defects in them.

4. How to minimize defects

Defect management system is closely linked to the quality system and quality culture of the organization. The basic principles, guided by the Total Quality Management (TQM) philosophy, for improving every quality system can be summarized as follows:

a) Identify the current state of the aspect under consideration,

b) Measure the values of the relevant parameters,

c) Identify a corrective measure,

d) Apply the corrective measure on a small part as prototype,

e) If successful in (d) above apply the corrective measure on the full system,

f) Measure the results (new values of the relevant parameters), and

g) If the step (f) indicates improvement, then retain the corrective actions as a part of the process in the quality system.

We apply similar steps for the aspects of defect management also. We use the attributes of defects captured in earlier section for this purpose. As an illustration let us look into the defects associated with the attribute value ‘requirements phase’ for the attribute ‘The phase to which the defect belongs’.

We examine all defects of this type. Based on the observation we come up with a checklist. Use this checklist for subsequent requirements capture. Check if that this type of defects reduces. If so, then make the checklist a part of the requirement capture process.

Every quality improvement mechanism can be used to minimize the defects. The main quality improvement mechanisms are: Inspection, Reviews, Testing, Training, Auditing and use of testing best practices. Two more mechanisms are used in addition to these:

(a) The prototyping process is used to minimize defects in the early phases of software development. In particular the throwaway prototypes designed to assess the gravity of a risk are useful in uncovering defects in early phase of software development.

(b) Have a glossary of domain terminology and their meaning.

All these mechanisms are widely used. However, different mechanisms are effective under different contexts. Normally, inspections are used for artifacts in the early phases of software development and for those artifacts (software modules/components and related documents) which are crucial for the software system. For most of the artifacts from the phases earlier to coding phase, reviews are widely used. Code review, which is a special type of review, is used for crucial code documents. Best practices are used in generating all artifacts. Training and auditing are used in all phases as these forms the quality assurance mechanisms. All the mechanisms are simultaneously used for those artifacts which are repeatedly used large number of times and need high reliability. A few examples are the artifacts associated with a software product, embedded system, and mission critical systems. Let us look a little more into the details of these mechanisms.

4.1. Inspections

Inspection is useful for discovering the defects in the artifacts associated with the early parts of software development phases like feasibility, architecture, requirement, analysis and design. Formal inspection consists of a team in which one person acting as reader, the author of the document, and the others experts in the associated areas. The reader reads each line of the document and asks for the feedback from the experts. The author takes notes and takes decisions based on the input. It is not mandatory for the author to incorporate all suggestions.

An interesting variation of this is suggested by Tom Gilb [7]. This is very simple and produces very good results for the ‘Defects’ in requirement documentations. The method is as follows: Call all stakeholders of the requirement document. For each line in the document ask (1) Who is the owner of the line (2) How much he can pay for the feature in it (3) what is the priority associated with it. Answers to these simple questions clean about 40% of the requirements document. Another simple and powerful step suggested by Tom Gilb[7] is to examine the architecture and requirement documents. Identify all the requirements related to ‘non functional’ parameters like performance, reliability, user friendliness, maintainability etc. Try to break up these qualitative...
aspects into a set of quantitative parameters and assign values for those parameters with the help of the customer.

4.2. Glossary of Domain Terminology
One of the biggest sources of defect is communication gap between different groups (from requirement to coding) involved in software development [8]. For example, the developers should understand the program specification in the domain context. The glossary of domain terminology acts as one of the binding material between different groups. When everyone understands the domain terminology correctly then there will be a great improvement in the communication between different groups. This common understanding produces a conceptual continuity of thought from one group to the next group. If the design group does not understand domain concepts then they are likely to misunderstand the analyst’s artifacts. Similar problems arise when developers misinterpret program specification. It is very essential to conduct the training to all the groups involved in software development about the meanings and relationship between different words (concepts) listed in glossary of domain terminology. The designers may need more in depth knowledge compared to developers. The required level of understanding of domain concepts and terms for a group depends on the particular context.

4.3. Prototyping
We need to discover as many defects as possible for the initial phases in the development life cycle. Cost of correction increases exponentially at later phases of the project. Hence, though the methods suggested above are expensive they are very effective and have high return on investment. Prototyping process is a powerful method to address the defects (or risks) identified with the initial phases of software development.

As an illustration, we discuss the prototyping process to address risks due to defects in capturing nonfunctional requirements. After getting clarity on the nonfunctional parameters of the system, one should build prototypes addressing risky nonfunctional parameters and any trade off between expectations from different nonfunctional requirements. Normally for a system two to three nonfunctional requirements are dominant and often one needs to do a trade off between them. Building a right prototype for a nonfunctional requirement helps to eliminate the risk (of not meeting that requirement) and hence defects associated with each of the nonfunctional parameter.

As an example, we need to build an online system to declare the result of a competitive examination taken by 50,000 candidates. The day (and time) on which results are declared is known early. Hence, on that day and at that time, the server receives requests from about 40,000 users. If we test this scenario after the system is developed, the server may crash. It will be very expensive to correct this mistake as a lot of effort has gone in the development work. The better way is to build a prototype (dummy system) with the same number of data files/tables, simulate the network, database and hardware environment and fire 50,000 requests from different places. If things work satisfactorily, then we can do the other parts. As another example, we are designing an ATM with expected response time of 5 seconds. The prototype for this purpose is built on the minimum requirements needed to see the response time during the initial phases of software development. Through this prototyping process we can mitigate the risks due to nonfunctional requirements.

4.4. Reviews
Review process is widely used to discover defects associated with major artifacts in all phases. Review is less expensive than inspection and is conducted by a team of people. In reviews, a reference document is given to the review team. The document to be reviewed is compared with the reference document. A perspective for the review is given. The review team members are experienced in that perspective. For example, a design document is to be reviewed regarding adherence to user interface standards (UIS). The UIS is the reference document. UIS experts will be in the review team along with the author and a reader. The reader looks at key aspects of the document and the experts look at the corresponding parts in the reviewed document. Usually there will be a time limit (one to two hours). The committee will go to the depth that is possible within the time limit. The reviewers log all the defects and the author looks into them after the review is over.
A specific type of review is ‘code review’, where the source code is reviewed before it is tested. The reference document is the program specification. One looks at each aspect in the program specification and see if a corresponding piece of code exists. Based on this, the defects are listed.

The structure and purpose of both inspection and review are same. The main difference between inspection and review is in the time spent on the process and the depth to which it is done. Inspection is a thorough in-depth process and hence is time consuming. Inspections are done for a small crucial
subset of artifacts. Reviews are done for the majority of the artifacts. The source code artifacts go through testing process and a subset of these artifacts also go through code reviews.

4.5. Training and Audit
Training is one of the ways of reducing defects. We noted earlier “To err is human”. It is also true that defects are introduced by people. Hence the easiest way to minimize defects is to train people with appropriate technical and soft skills, process knowledge, etc. Normally in the beginning of a project, the project manager knows the expected number of people and their skill level. He needs to bid for the required human resource with the human resource manager or might directly approach the pool of available employees. Most of the times, the project manager does not get people meeting the complete requirements. He has to opt for employees who do not have the required skills, but capable of learning them. The project manager also discusses with the employee regarding what training is appropriate to make the person productive for the project. One needs an appropriate system to assess the effectiveness of training programs with respect to the requirements given by the project manager.

In a similar way, audits help in reducing defects. A software system is produced by people following a set of processes. As time progresses, normally people start deviating from the process and certain steps in a process get altered or outdated and hence the distance between prescribed process and followed process starts increasing. The audit process helps to reduce this difference at regular intervals. At the end of the auditing either people start following the process or the processes get altered / deleted to reflect the current situation.

In the context of software systems, widely used auditing systems are International Standards Organization (ISO), Software Engineering Institute – Capability Maturity Model (SEI-CMM), and Six Sigma. The guiding philosophy of all auditing systems is TQM philosophy (summarized in the beginning of section 4). The auditors use the following guiding principles:

(a) Say what you do: The method to perform a task is described in a process or procedure. Are the processes, procedures followed by the company properly documented? Do these get mapped to the set of standards outlined by the auditing system?
(b) Do what you say: Do the employees in the company, while performing a particular task, follow the written procedure? The auditors visit the company and interview management people and employees to verify this.
(c) Prove it: Every procedure should have a section which describes the records (quality record) that should be produced if the procedure is followed. The auditors look for adequate number of the quality records for the audited process.
(d) Who is responsible for a given task? If there is a task for which no one is responsible, it means that the task will not be performed satisfactorily. If many people are responsible, then there must be one person answerable for the completion, quality, schedule, etc., of the task.

Each of the auditing systems specifies certain standards to be followed by the management and employees of the company while performing their tasks. In case of ISO auditing, the ISO organization recognizes a set of companies (like KPMG, Veritas etc.) competent to perform ISO audit. Any organization interested in ISO certification should approach one of the ISO recognized auditing firm. The auditors from the auditing firm do the auditing with respect to each of the clauses of ISO standard. If there is a non compliance in following a procedure then that goes as an entry in the non conformance report (NCR) produced by the auditor. Depending on the number and type of non conformance, the company is certified or not certified. ISO auditing also involves an audit called ‘internal auditing’. The internal auditing is initiated by the company management (usually by the quality department). This is done six months after the ‘external audit’ (by ISO auditors). The results (in particular NCR report) of internal audit are used in external audit process.

In case of SEI-CMM system, there are auditors qualified as CMM auditors. SEI has published the CMM guidelines and norms to classify the level to which the company practice belongs. In principle, the quality department of the company itself can audit the company’s procedures and practices with respect to SEI-CMM guidelines and classify itself to be in a certain level based on CMM norms. Normally, to get credibility, the CMM assessment is done by an external auditor.

Testing is more of a quality control mechanism. The testing process is depicted in the following diagram.
We prepare test suite based on the test cases and run the system with the test data. For each of the test data we have expected output and compare this with the output produced by running the system. If there is any difference, an error is reported and the relevant developers correct the program. Again, the test suite is run (called regression test). This process is repeated till there are zero bugs. This type of testing is called regression testing.

A golden rule in preparing test data is that every test data should be traceable to a requirement. Typically, each requirement has many scenarios. For each scenario we have many test cases. For each test case we will create a set of test data. When we follow this model, it is possible to have a system for the traceability of every test data to the corresponding requirement. This aspect can be easily achieved by using tools to capture requirements and conducting tests.

The testing tools also give much useful information related to testing process. The typical information is:
(a) The requirement scenarios not tested,
(b) The efficiency of test data. If test data corresponding to a test case is not producing any error report (i.e. not detecting any bug), then that part of the system may be error-free and one can remove such test data from subsequent tests.

In iterative paradigm (example: Rational Unified Process) in contrast to waterfall paradigm the build happens throughout the development process. In each build, add one concern and at the end, test the system for the correctness with respect to all the previous iterations. This is achieved by passing all the previous test data. In such scenarios automating the testing process using tools becomes necessary.

Testing and Code review: Both the processes attempt to reduce the bugs in the source codes. Experience shows that best results are achieved when code reviews are done in the beginning and then after a certain effort on code review (the effort needed depends on the past data associated with code review & testing process) one can start testing process. Generally, reviews ensure the matching of program specification/ standards with the code. The testing process ensures a minimum quality in the program. The other types of testing [3] are white box testing, black box testing, load testing, stress testing, unit testing, integration testing, software system testing, and customer acceptance testing (CAT).

In white box testing, the tester knows the logic of the program and goes through the source code to see if any problems are present. He creates the test data based on the different paths that are possible in the logic of the program. Black box testing treats the program code as black box. The test data is prepared based on the program interface i.e., for a set of inputs write the expected outputs and check with the actual output.

Unit testing is the testing done by the developer and is a white box test. Different programs are combined to form a module. The integration testing ensures that different programs in a module are passing the right inputs to each other and they are producing the correct output.

Load testing is to test the system with maximum load. In particular in a multi-user environment test the system with maximum number of users accessing the system simultaneously. When a report need to be generated run it with maximum database records. If we carry the load testing beyond the maximum allowed size we are doing stress testing. Here mostly we encounter exceptional scenarios. We study the way the system degrades under such conditions.

CAT is the test conducted by the customer before accepting and deploying the software. The software system testing is done by the software development organization before shipping the software to the customer for CAT. Software system testing creates a pseudo customer environment and the transactions are simulation of customer testing. A realistic criterion to measure the quality of defect management system of a software development organization is that the developed software should produce zero defects for the normal ‘customer’ transactions [3] in CAT and in the field. Here we mean ‘customer’ as the end user of the system. Let us look at the key features of software system test.

4.7 Software System Test
An experienced software system test team models the typical customer’s transaction scenarios and accordingly aims to create essential minimum test transactions needed to test the system. The efficiency of the system testing process is measured by:
(a) How close the testing model of the system is to the customer transaction space. As it is impossible to test a system completely, we test the scenarios and typical exceptions normally encountered by the customer. An
efficient test team can identify such scenarios. The next best is to test it to a level acceptable to the customer. A domain expert from the system test team, filling an important role, helps in identifying the relevant tests.

(b) The number of test data used to test the system. The lower is this number better is the testing process provided the customer satisfaction level regarding the quality of software is maintained. This is necessary because of the following reasons:

i. The size of test data space is very large

ii. Tests need to be conducted frequently using the same test suite.

iii. One need to do a trade off between Cost of testing, Cost of error/failure (mission & life criticality, frequency of exposure – resulting in the probability of error occurrence)

The objective is to remove the potential subset of test data which have not contributed to detect a bug in the previous runs or if the change is small remove the subset of test data which do not test the changed part of the system. This is achieved by improving the maturity of the defect management process and inducting domain experts in the system testing team.

All system testing models are geared to address the above two issues.

4.8. Regression Testing

As we mentioned earlier, regression testing is running the full test suite on a program and correct the source code, then run the same test suite again on the corrected program until no bugs are reported. We call these as regression tests denoted by T0, T1, T2 etc. When we plot graph of number of bugs for each tests T0, T1 etc we expect the number of bugs to be lesser in each regression test. The expected signature is shown in Fig 2(a). However many times we get a signature of the type shown in Fig 2(b). The regression test T1 has more bugs than the first test T0 [3]. This happens due to ‘defects hiding under defects’ in particular when there is a substantial chunk of initialization errors. This signature is common when the developer is an inexperienced one. The usual way of mitigating this type of defects is to provide a checklist of initialization errors to the inexperienced developers.

In the Fig 2 above the y-axis represents the number of bugs found in the regression test.

4.9. Institutionalizing Testing Best Practices

An experienced person would have internalized his solutions into a set of ’units’ or rules of thumbs. These rules of thumbs by the experienced people in different aspects of defect management can be systematically documented. When the experienced people write these rules for the respective context and the beginners use it even without understanding fully, the beginners contribute like experienced people and hence the defects in that part of the product is minimized. The best practices can be written in the following format:

(1) Each best practice has a name
(2) The context is described clearly
5. Test Team Organization

Normally, unit test is done by the developer. The integration test, module test, system test and user acceptance test are done by separate test teams. Unit test is white box test and the other tests are black box tests. When agile/pair programming paradigm is followed, then there will be one tester attached to one developer. In most of the cases, black box type of testing is done by separate testing team. In this case, to monitor the testing process, a simple schedule/monitoring/reporting mechanism is followed. One can have a simple table with attributes like (project id, tester id, module id, start date, expected completion date, actual completion date). The test leader discusses with test team members and comes to an agreement with the dates in the table. As and when the work is completed, the tester enters the ‘actual completion date’ entry. The test leader looking at this column knows about the status of the testing process. We are assuming that there is trust between test leader and the members. This aspect minimizes the effort involved in control and monitoring the work. In spite of trust, it is necessary to have an entry ‘actual completion date’ because oral communication regarding completion of work can be ambiguous even when there is trust among members. If there is no trust, then instead of a simple table accessible to all the team members on need to build a complex system. This system is accessible through passwords and each member can access his/her own record. The system keeps evidence about the entered time, information etc. In short, the system has huge overhead. In general, the systems and procedures of the company will be complex when there is no trust between team members and team leader. The organization culture and management value system plays a vital role in establishing this trust in the employees of the company. The company should make conscious effort communicating its value system to its employee (company’s ‘tea-club’ conversations are good indicators of employee perception).

6. Defects Related to Customer Satisfaction

It is very essential to view the impact of defects on customer satisfaction. The customer is concerned about quality, cost and schedule. It is necessary to define these parameters together with the customer/stakeholder(s). The following are metrics related to these parameters

Schedule: number of deliveries ahead of time, on time and delayed (by how many days).
Quality: The number of bugs in the customer acceptance test is a measure of the quality of the software delivered.
Cost: Planned and actual (percentage over-run).

Deviations in these parameters are defects related to customer satisfaction. Once in 6 months a customer satisfaction feedback [scale: Excellent, Good, Satisfactory, Unsatisfactory] should be gathered from customers. One should set targets for that period (like % of customers giving feedback less than ‘Good’ should be less than some number). In a similar way, monitor the number of customers retained, gained and lost in that year. These metrics help to tackle the defects related to customer satisfaction.

7. Classification of Defects

7.1. Major / Minor
This classification is widely used in the ISO auditing. A defect is said to be major when it happens across the system. The defect is minor when it happens at random instances. ISO system has 20 general requirements and other standards specific to the industry to which the organization belongs. When the process followed in the company violates the ISO standard specifications a non-conformance report (NCR) is created. The NCR is defect list with respect to ISO standards.

7.2. Orthogonal Defect Classification (ODC)
This is a very powerful and useful method for defect classification. The value proposition of Orthogonal Defect Classification [11] is that it learns from the defects of the current ongoing project and uses this learning to improve the current project itself. The following is outline for ODC method.
The classification scheme used is different phases (requirement, analysis, design, integration, coding, builds). A defect belongs to a phase. This phase is identified by observing ‘which document is changed to fix the defect’. If requirement document is changed. then it belongs to requirement phase. As an illustration let the bug is ‘wrong income tax value’ in the context of a payroll system. This belongs to coding phase if the change done to fix the bug is only in the source code. On the other hand, if the problem was due to wrong capturing of requirement, the requirement document will be altered (of course, the source code is also altered, but the cause is due to defect in requirement). In this case the defect belongs to requirement phase. This way of classification is simple (you need to know only which documents are altered to fix the defect), unambiguous and every defect belongs to only one phase (the classification is orthogonal).

In a similar way, record the phase in which the defect is discovered. These two parameters are sufficient to do the ODC classification. The following inference regarding the status of the project can be drawn using this classification scheme.

Let us take a scenario where about 20% of a project is completed. We have 100 defects discovered in coding phase. Let the distribution of defects according to the phase to which it belongs
[requirement, analysis, design, coding] be [40, 20, 20, 20]. These scenarios do occur in real life examples. If the project were executed properly, we expect maximum errors belong to coding phase and minimal number to the requirement. The observed values show maximum number of defects belongs to requirement phase. This clearly shows that the requirement phase was not done properly and the project has already gone to coding stage. It makes sense to disband coding and go back and fix the requirement process so that the remaining 80% of the project is benefited.

Trigger Analysis: A trigger is an environment context in which a particular type of defect surfaces. We can associate a particular type of defect with each type of trigger. When we know the potential triggers for a system we can test the system for the associated scenarios. For example we have developed a system in java environment and it needs to be deployed in .net environment. In java environment we may not get any defects but the .net environment is a trigger for the associated defects. The triggers are functions of the environment. It is necessary to identify the triggers in the organizational environment and test the new system with respect to these triggers. The ODC analysis is independent of specific domain/project. It is not as time consuming as root cause analysis and not as general as statistical techniques. The lessons from one project can be used by other projects.

8. Summary
In this article we have described different aspects of people, culture and process in a Defect Management System. We have discussed the meaning of defects, attitude towards defects, different approaches to minimize defects, some lessons that can be learnt by studying defects in a scientific manner, different attributes (and their significance) of defects that are useful, and different methods to classify defects in a meaningful way.

9. Acknowledgement
I thank Professor H N Mahabala for mentoring me in this area. A part of this work was done during my stay at Malmo University (Sweden). I thank Linnaeus-Palne exchange program and the host Simon Niedenthal and K3 department at Malmo University for making this possible. I thank P G Bhat of Parity Computing and the anonymous referees for their critical and useful feedback.

10. References
[9] Build is a standard process: for details see references [1], [3], [6]
Abstract

Software testing involves the process of detecting software discrepancies so that they can be corrected before they are installed into a live environment supporting operational business unit. Considering the importance of conscious KM practice for software testing project management and better support for the complex and multi-staged task of software testing, it seems logically natural to adopt a KM approach for software testing projects. Because of a high initial and ongoing commitment involved in full scale KM frameworks and methodologies, an alternative and incremental approach to KM for software testing is called for. We suggest to use wikis as a promising technology. Wikis have been touted for low-risk, cost-effective tool for knowledge sharing and collaboration. Based on literature review, we try to identify typical activities in software testing for wiki use, potential benefits of using wikis, and success factors of adoption and usage of wikis. As a future study, we plan to conduct a set of qualitative case studies of wikis in real software testing organizations to explore the role of these success factors of wikis and confirm the potential benefits in a natural setting.

1. Introduction

Software testing involves the process of detecting software discrepancies so that they can be corrected before they are installed into a live environment supporting operational business unit. Normally, there are four different levels of testing in software testing: unit testing, integration testing, system testing, and user acceptance testing. When testers perform each of the four levels of testing, they use various techniques and different types of test cases, and generate a large volume of documents such as testing results, bug lists, and fixes and changes made.

2. Wikis for KM in Software Testing

2.1. Software testing and KM

Software testing involves the process of detecting software discrepancies so that they can be corrected before they are installed into a live environment supporting operational business unit. Software testing normally consists of four different levels of testing: unit testing, integration testing, system testing, and user acceptance testing. When testers perform each of the four levels of testing, they...
use various techniques and different types of test cases, and generate a large volume of documents such as testing results, bug lists, and fixes and changes made.

The need for and benefit of KM in organizations has been widely mentioned in the literature [14,15,16]. Especially in project management, conscious and explicit knowledge management practice for project management was emphasized as one of important factors for project success [25]. Zhang et al [25] argued that the lack of conscious practice in knowledge management for project management contributes to the high project failure rate. Their argument about the relationship between successful projects and KM was reiterated in an empirical study on the success and failure of process improvement projects by Lapre and Wassenhove [8]. As a result of not having conscious practice and effort in knowledge management for project management, we can often observe the typical problems in project management; the mistakes made in one project are repeated in other projects, or the knowledge or lessons learned from a project is lost over time, or the knowledge from multiple projects is unorganized, scattered or isolated across an organization. So, considering the importance of conscious KM practice for software testing project management and better support for the complex and multi-staged task of software testing, it seems logically natural to adopt a KM approach for software testing projects.

There are several KM frameworks and methodologies that can be used for software testing in the literature [5, 16, 17]. These frameworks try to promote four types of conversions between explicit and tacit knowledge [14], and provide detailed steps, mechanisms, critical success factors, and so on. However, all of them involve a high initial and ongoing commitment in terms of training, technology, meeting, documentation, and organizational culture change, which makes it very costly for organizations to adopt these frameworks for their KM endeavor.


Considering the aforementioned problems involved in the full scale KM frameworks and methodologies, an alternative and incremental approach to KM for software testing is called for by using a low-risk and cost-effective technology in order to better support software testing groups.

As a promising candidate technology that will support KM for software testing, we suggest to use wikis as evidenced in a case study by Wagner [22]. Based on the existing studies in the literature [7, 23, 24], Wagner [22] described knowledge acquisition bottlenecks in organizations as follows:

- **Narrow bandwidth.** The channels that exist to convert organizational knowledge from its source (either experts, documents, or transactions) are relatively narrow.
- **Acquisition latency.** The slow speed of acquisition frequently is accompanied by a delay between the time when knowledge (or the underlying data) is created and when the acquired knowledge becomes available to be shared.
- **Knowledge inaccuracy.** Experts make mistakes and so do data mining technologies (finding spurious relationships) Furthermore, maintenance can introduce inaccuracies or inconsistencies into previously correct knowledge bases.
- **Maintenance trap.** As the knowledge in the knowledge base grows, so does the requirement for maintenance. Furthermore, previous updates that were made with insufficient care and foresight (“hacks”) will accumulate and render future maintenance increasingly more difficult.

To break these bottlenecks in knowledge acquisition, Wagner [22] proposed to use Wiki as enabling software and to empirically analyze the Wikipedia to produce evidence for the feasibility and effectiveness of his proposed approach.

Therefore, as an alternative and incremental approach to KM for software testing, it is more desirable to use a low-risk and cost-effective technology - Wikis. A Wiki was originally defined by Leuf and Cunningham [9] as a set of linked web pages, created through the incremental development by a group of collaborating users. However, this original definition has been changed to be a technology or software as follows:

- A Wiki is a Web-based collaboration technology designed to allow anyone to update any information posted to a wiki-based Web site [20].
- A wiki is software that allows registered users or anyone to collaboratively create, edit, link, and organize the content of a website, usually for reference material. Wikis are often used to create collaborative websites and to power community websites. These wiki websites are often also referred to as wikis; for example, Wikipedia is one of the best known wikis (www.wikipedia.com).
- A wiki is actually two things:
  1) A program that makes it exceptionally easy for anybody to edit Web page, and
  2) A philosophy regarding how users should go about that editing [10]

So, a wiki can be a combination of a set of linked pages, software creating and managing those pages (or web site), and philosophy (or wiki way).

2.3. Potential use of Wikis in software testing
Wikis have been widely hailed as a low-cost effective tool for knowledge sharing and collaboration. There has been considerable research on the best practices in the incremental development of wikis within departments and/or project groups for document management, personnel training, and project coordination. A survey of corporate wiki users [21] revealed that wikis were used in a range of business activities, from research and development, to customer relationship management, to e-learning. In the result of the same survey, the specific tasks supported by wiki functionality included collaborative writing, project management, helpdesk/FAQ provision, and information sharing. They identified ten categories of most common corporate activities supported by Wiki. Out of those ten categories, the following seven categories of activities and associated uses of Wikis are deemed more relevant to software testing. Based on the survey result conducted by Majchrzak et al [11], the following table was prepared by removing some of stated uses that are not relevant to software testing. We added one more row to the table to suggest associated uses of wikis specifically for software testing.

Even if there is not much research done about using wikis specifically for software testing, there are a variety of studies and successful stories about using wikis for software development and project management [10, 18, 21]. Considering the similarity between these two tasks, we believe that using wikis for software testing will be also feasible and effective as in software development.

### 3. Benefits of Using Wikis for KM and Collaboration

From literature review and our own observation, we identified three types of benefits from adopting and using wikis: benefits in terms of productivity improvement, benefits from the viewpoints of wiki users and participants, and expected benefits from the perspective of KM and organizational learning.

#### 3.1. Benefits in terms of productivity improvement

Based on interviews from cooperate Wiki users and user managers, Wagner and Marjchrzak [21] claim that Wiki can create significant productivity improvements even though wiki returns are seldom formally quantified. They presented five successful cases in their work [21] that realized productive gain by adopting and using Wikis.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Stated Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-learning</td>
<td>requirement descriptions, testing, assignments to training</td>
</tr>
<tr>
<td>Project management</td>
<td>creation of deliverables, meeting agendas, status reports, standards and practices</td>
</tr>
<tr>
<td>Posting of general info and knowledge management</td>
<td>how-to’s, personal blogs, corporate information, collaborative pages of resources related to a topic as a complement to formal intranet pages, best practices, innovative methods and processes utilized, “great ideas” saved for later, corporate policies and procedures, guidelines.</td>
</tr>
<tr>
<td>Ad-hoc collaboration</td>
<td>“hashing out” ideas, remote collaboration, business brainstorming</td>
</tr>
<tr>
<td>Resource management</td>
<td>enabling users to make claims for usage of shared machines</td>
</tr>
<tr>
<td>Tech support</td>
<td>best practices, customer support information-sharing, local help information with how-to’s and best known methods, systems requests for new hardware, email setup, software downloads</td>
</tr>
<tr>
<td>Software development</td>
<td>technical documentation, client approval, issues tracking internal workflow, quality and process</td>
</tr>
</tbody>
</table>
| Software testing (specifically) | - IEEE Std. 829-1998 for software test documentation for testing planning:  
- Test Plan Identifier; Introduction; Test Items; Risks and Contingencies; Features to Be Tested; Features Not to Be Tested; Approach; Item Pass/Fail Criteria; Suspension Criteria and Resumption Requirements; Test Deliverables (test plan, test design specifications, test case specifications, test procedure specifications, test item transmittal reports, test logs, test incident reports, test summary reports); Testing Tasks; Environmental Needs; Responsibilities; Staffing and Training Needs; Schedule; Approvals  
- Results of unit testing (list of most complex or error-prone modules)  
- Defects/ bug lists and changes/fixes in system testing  
- A matrix of requirements vs. test cases in user acceptance testing  
- Post-analysis report at the completion of software testing. |

Those cases are summarized in Table 2 on the next page. Since all of the activities (and corresponding realized benefits) mentioned in Table 2 are typical in project management, we again believe that Wikis can...
be an effective tool to adopt and use for better project management in software testing.

Table 2: Examples of productivity improvements by using wikis

<table>
<thead>
<tr>
<th>Company</th>
<th>Benefits from using wikis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windriver</td>
<td>Reduced time documenting projects: &quot;... managers at Windriver spend 20% less time documenting their projects using TWiki compared to the previous setup using HTML editors and FTP.&quot;</td>
</tr>
<tr>
<td>Dresdner Kleinwort Wasserstein</td>
<td>Improved communication and less teleconference duration: &quot;... the teleconference used to be one and a half hours long, with much time wasted on bringing people up to speed on the week's events. Now team members update themselves on the wiki, and that part of the teleconference takes five to ten minutes. The rest of the teleconference is used for idea generation, being innovative, talking about problems and looking at solutions, which is what the meeting should be about.&quot;</td>
</tr>
</tbody>
</table>
| Motorola                | Enhanced multi-team, multi-country collaboration and reduced project document management: "... originally used wikis to enable collaboration between distributed teams, separated by geography, language, and culture. The key benefit of that application was the ability to allow individual teams to define their own project management process, while at the same time standardizing on a reporting system which rolled up summary pages from the individual project sites." "...after introducing a wiki application to replace a MS-Word based Quality Management System used by 350 engineers, the organization received the following benefits:
  * Change requests turn around reduced from one year to one month.
  * Number of review comments and change requests has increased by a factor 5-10, on some topics by a factor 100.
  * Time required for preparing presentations dropped from 2 hours to 15 minutes." |
| Artisan Creative        | Reduced e-mail volume, easier knowledge sharing, easier monitoring of managerial performance, new hire training and education: "Originally the wiki was introduced to reduce email volume and to create a shared knowledge resource," "...the company also used it to monitor managerial performance through "agenda templates". "...where there is little time to formally introduce new employees to the company culture and project history, new hires are asked to review wiki content and thus learn about the individuals in the organization, projects, and decisions made." |

IBM

- Effective and efficient project document management, fine-tuned access right control:
  "Using wikis as a software engineering documentation mechanism solves a couple of age old problems with documentation:
  1) documentation cannot be located ; 2) documentation becomes quickly inaccurate
  Related to #1, with wikis, you simply need to remember the URL for any one of the pages, and navigation becomes pretty simple, either through embedded page links or via a search mechanism. Regarding #2, anyone with access can modify the content. So if you see something wrong, you change it. This is much more efficient than changing a traditional document, notifying the author, and getting the author to republish the corrected document in the correct place.
  Another consideration for using wikis to document software projects is the need to restrict access to proprietary information. On the one project, which is really just IBM confidential, access is controlled by hosting the content on the IBM Intranet, which is behind a firewall and requires a VPN account to access. The second project is confidential even within IBM, so we use a simple access control list and user id / password challenge mechanism to limit access."

3.2. Benefits from the viewpoints of wiki users and participants

A survey of 168 corporate wiki users was conducted by Majchrzak et al. They asked participants for organizational, work-related, and personal benefits that can be obtained from using corporate wikis. According to their survey result, there are three main types of benefits (stated by survey participants) from corporate wikis: enhanced reputation, work made easier, and helping the organization to improve its processes [11]. Majchrzak et al. reported that
  1) as for organizational benefits, wikis can help an organization by improving work processes, collaboration and knowledge reuse while wikis appear to rarely help an organization identify new business opportunities;
  2) as for work-related benefits, wiki use made their work easier.
  3) as for personal benefits, not many participants expect to obtain any personal benefits from wikis, which are quite different from the findings in open source research.

3.3. Benefits from the perspectives of KM and organizational learning

Constructing a knowledge base by using wikis can be an effective and economical KM
solution for organizations that want to capture and convey explicit knowledge of their testing groups. Considering the potentially high organizational time, commitment, cost, and payoff risk involved in adopting and implementing a full-scale, knowledge management framework or methodology, an alternative and incremental (organizational learning) approach to KM in software testing can be introduced by first focusing on handling explicit knowledge (as opposed to tacit knowledge) for software testing - identifying, capturing, storing, distributing explicit knowledge for software testing. This explicit knowledge is usually embedded in software testing documents. As opposed to tacit knowledge (typically residing in the heads of the testers) which requires extensive resources to divulge, explicit knowledge in testing documents should be less costly to capture and organize. Lack of conscious practice and effort for any systematic knowledge management activity will result in loss of important organizational knowledge. In essence, best practices embodied in explicit knowledge are the “low hanging fruit” of knowledge management that has not yet to be picked. Once this low hanging fruit has been harvested into a knowledge base in the wiki system, organizations can learn how to better reuse its knowledge and more confidently invest in more expensive, comprehensive and structured knowledge based systems with the assurance they will receive guaranteed payoffs.

The constructed knowledge base in the wiki system will help promote exchange of explicit knowledge and increase their productivity in the software testing. In addition, the captured explicit knowledge in the knowledge base can be used by the members of the testing groups to convert the explicit knowledge into their own tacit knowledge. This internal conversion process, internalization, is one of the important conversion processes identified by Nonaka and Taueuchi [13].

Another benefit can be realized from the perspective of organizational learning and culture. After successful adoption, implementation, and use of the constructed knowledge base implemented by using wikis, this learning experience can be beneficial to launch on a more complex and challenging KM project. The experience and lessons learned from the proposed study can be used as a stepping stone to launching on more challenging KM projects. In addition, this proposed alternative KM approach can help cultivate and promote the organizational culture in the testing groups amenable to other KM endeavors.

4. Possible Success Factors for Adoption and Usage of Wiki

Based on intensive literature review, possible success factors for wiki are identified and summarized in Table 4. Several factors are repeated in many places while other factors are unique in some work. The list of success factors by Wagner and Majchrzak is most comprehensive, so we prepare a summary table (Table 4) mainly based on the work of Wagner and Majchrzak [21] and some other factors are added to the table from other sources [1, 3, 4, 10, 20]. Most widely mentioned factors for wiki success are cooperate culture, fit of task and technology, Wiki champion and shaper, and management and monitoring of the wiki system. The other success factors seem to be straightforward as in any new information technology adoption and implementation (e.g., ease of use, WISWIG interface, user education/training). One of the most interesting success factors identified in a case study by Wagner and Majchrzak [20] is goal alignments among participant. This success factor was identified based on a case study of wiki editorial experiment in a large, mainstream organization that tries to engage readers to prepare an editorial for a controversial issue. Even if many of readers participated in this editorial effort, the wiki editorial was shut down in three days because of repeated site defacing. According to Wagner and Majchrzak [20], the reason this wiki editorial failed is that “there has been little attempt to align goals among participants; moreover, the desired goals in the user community not only were diverse but possibly incompatible.”

Table 4: Factors for Wiki Success

| Cooperate culture | • Does the wiki participation model fit with corporate culture? (less hierarchical/more collaborative, merit based) (also in [1,3])
| Capability of Wiki Champion | • Is there a wiki champion who understands wiki technology and the wiki way, and has the organizational knowledge and standing to engage others, and to shape the collective knowledge contributions?
| Guidelines for protecting intellectual | • Does the organization (wiki team) have guidelines or a shared understanding to protect wiki based knowledge? |

| Cooperate culture | • Does the wiki participation model fit with corporate culture? (less hierarchical/more collaborative, merit based) (also in [1,3])
| Capability of Wiki Champion | • Is there a wiki champion who understands wiki technology and the wiki way, and has the organizational knowledge and standing to engage others, and to shape the collective knowledge contributions?
| Guidelines for protecting intellectual | • Does the organization (wiki team) have guidelines or a shared understanding to protect wiki based knowledge? |
Wiki software

- Has wiki software been chosen that incorporates access rights management and task structuring?
  - intuitive, easy to use, WYSIWYG (from [3, 4, 10])
  - egalitarian, freeform (from [4, 10])

Management and Monitoring

- Does each wiki have someone who appears to be shaping the contributions?
- Does the organization (wiki team) have clearly defined rules for posting contents? (from [1])
- Does the organization (wiki team) have plans in place for managing and monitoring user behaviors? (from [1])

Fit of Task and Technology (also in [3])

- Is the nature of the wiki supported task such that participants with personal, efficiency driven interests can benefit from wiki participation?
  - information intensive tasks: all require information and knowledge exchange between team members, enabling others to learn from the combined insights of the team.
  - tasks with a considerable level of knowledge turnover; suggesting that tasks should involve ones where knowledge freshness and exchange are critical.

Critical Mass of participants (also in [3])

- Is the wiki team of sufficient size to ensure both reading and writing?

Processes for Sustainability

- Do you have plans in place to phase in the wiki, steer participants to the wiki, create active participation, create wiki culture, and maintain steady-state?

User Training, Education (also in [3],[4])

- Will training sessions for participants be arranged and flexibly available?

Goal Alignment (exclusively from [20])

- Are there alignable, multiple purposes/goals among wiki site participants?

5. Summary and Future Research

Considering this complex nature of software testing, adopting and practicing an appropriate and effective project management is indispensible. We started by mentioning importance of KM for project management in software testing. As a promising technology to support this KM effort and practice (as well as collaboration), we proposed using wikis for KM in software testing. Wikis are touted as a low-risk and cost-effective tool for knowledge sharing and collaboration. We went on to identity typical activities in software testing that might be appropriate for wiki use. Nine categories of activities and associated wiki uses are summarized in Table1. Then, we tried to identify potential benefits of using wikis for KM and collaboration based on literature review and our observation. There can be three types of benefits resulting from using corporate wikis: benefits in terms of productivity improvement, benefits from the viewpoints of wiki users and participants; benefits from the perspectives of KM and organizational learning. To realize these types of benefits of adopting and using wikis, many of success factors need to be carefully taken into account and meticulously followed through during adoption and diffusion. Table 4 summarizes those success factors. Most widely mentioned factors for wiki success are cooperate culture, fit of task and technology, Wiki champion and shaper, and management and monitoring of the wiki system. However, the factor, goal of alignment, seems to be an interesting one to note. This factor was identified from a case study of a failure case of wiki adoption for editorial effort by one of major media organization. Another similar failure case of using wiki for mass creativity is the case of “A Million Penguins” (http://en.wikipedia.org/wiki/A_Million_Penguins). The objective of this project was to write a novel based on mass creativity using a wiki. The project was supported by Penguin Publisher and De Montfort University. There were 1500 relatively anonymous writers from around world, which was quite a successful and significant participation, but this project suffered from chaotic zaniness and failed: too many changes in characters and too many updates/deletions to manuscript too soon. This case illustrates that a large number of
participation to a wiki system is not necessarily a good sign of higher sustainability and nor enhanced collaboration.

As a future study, we plan to conduct a set of qualitative case studies of wikis in real software testing organizations. We will explore the role of these success factors of wikis and confirm the potential benefits in a natural setting to ensure that the possible success factors and benefits would be generalizable to actual software testing environment. Alternatively, we also want to know there are any other success factors of wikis and potential benefits of using wikis unique in the software testing environment. In addition, there has not been formal ROI research conducted when wikis are adopted and used in the software testing environment.

Another line of future study is to conduct a comparison study between wikis and a web-based search tool such as Google Search Appliance for KM in software testing. Google Search Appliance has many merits in terms of cost, comparability, seamless deployment, ease of installation and use, scalability, and implementation risk. Google Search Appliance (GSA) delivers fast, relevant and secure access to information. This tool can provides universal search across variety of internal and external sources — including file shares, intranets, databases, applications, hosted services and content management systems. In short, this kind of search tool would also seem to be appropriate as a low-risk, cost-effective and easy-to-use tool that is well suited for implementing a low-cost incremental approach to knowledge base development and use for the testing groups. As pointed out by Wagner and Majchrzak [21], alignment of wikis with corporate IT architecture is of great concern in the long run once there are many wikis within a firm. We may use GSA to connect these wikis in a firm to better support KM effort and collaboration by combining wikis with GSA. We will also investigate this possibility of using hybrid approach to KM in the software testing environment.

6. References


Perspectives of Offshore Testing Vendors

Robin Poston, Judith C. Simon, and Bill Kettinger
University of Memphis
rposton@memphis.edu

Abstract

Global offshoring is not new. Steady improvements in information and communication technologies and technical skill levels globally have led to a boom in offshore outsourcing for software development and testing. While considerable research exists examining specific factors related to selecting and working with offshore vendors, little research has taken a best practices perspective on how to evaluate and sustain a successful long-term relationships with an offshore vendor. Now that many firms have been outsourcing globally for many years, it is time to examine those best practices that result in successful outsourcing partnerships over the long term. This paper addresses this gap in the literature by examining three major testing firms in India that have differing levels of success in sustaining a long term relationship with their Fortune 500 client. Key findings which incorporate direct quotes from the vendors are offered to shed light on the best practices in building successful long-term relationships between offshore vendors and global enterprises.

1. Introduction

More and more companies are moving their system development and testing work offshore. Key drivers of this trend include the need to: reduce costs and remain competitive; improve the quality of development and testing efforts; obtain better predictability in work loads; change the entrenched organizational culture while improving processes and quality; and take advantage of the value of offshoring in regard to cost, quality and time-to-market [9]. Yet to achieve these benefits, companies and their vendors will need to adopt successful outsourcing partnerships over the long term for resource management, coordination and communications, knowledge retention, initiating and generating new innovations, sharing ideas, relationship building, and value add activities.

Considerable research already exists but is limited to specific factors of selecting and working with offshore vendors. For example, recent research has indicated that senior technology managers are focusing on ways to add more value through outsourcing, such as increased flexibility and efficiency [26]. They are also realizing that many sourcing options exist, many of which are equally viable, thus raising issues of implementation and execution [22].

The outsourcing relationship between the client and vendor has been examined based on an economic view (coordination and governance of economics agents), social view (relationship between clients and service providers regarding issues like power, trust, or culture), and the strategic management orientation (how organizations deploy strategy to achieve their goals) [28]. From these analyses we find firms are constrained by their past choices, resources are not perfectly mobile, and expertise is not easy to reproduce or imitate [28].

Meanwhile, published reports continue to indicate a trend toward offshore outsourcing, including estimates that U.S. companies account for approximately 70% of the global offshore market and about 80% of India’s offshore clientele [1, 2, 14]. One study reports that the most common challenges in offshore outsourcing include [25]:

1. Instability because of geo-politics,
2. Mismatch in cultures, values and norms,
3. Protection of intellectual capital,
4. Imperfect information about offshore vendors,
5. Unrealistic expectations on cost savings,
6. Differences in time zones,
7. Location of client and vendor team members,
8. Knowledge transfer difficulties both directions,
9. Layoff and loss of human capital,
10. Disruption of work practices for employees, and
11. Lack of due-diligence resulting from offshore bandwagon mindset.

Given the trend for U.S. companies to send more work to Indian offshore vendors and the challenges of working with them, it becomes critical that research take a best practices perspective on how to evaluate and sustain successful long-term relationships between clients and their offshore vendors. This paper, directed
toward that goal, reports on interviews with three major testing firms in India that have differing levels of success in sustaining their relationship with their Fortune 500 client. While this paper is evidence of our initial data collection efforts, it serves as a starting point for future data collection and analysis.

2. Data Collection Methodology

“If theory talks only to theory, the collective research exercise runs the danger of becoming entirely self-referential and out of touch with reality” [29]. We propose that the current literature on offshore outsourcing relationships does not adequately address the evolving “real world” situation companies are facing today. Most of the literature for outsourcing and other contributing components do not reveal insight into the best practices of “how” to manage a long-term client-vendor relationship in offshore outsourcing situations.

In the current project the case study interview methodology was employed. Case study interviews are a widely accepted methodology used to identify and create relevant theoretical constructs, propositions, and theories [8]. In the absence of an appropriate existing theory, such as in this current project, case study research is appropriately used to frame research in terms of the importance of a phenomenon in the absence of a plausible theory [7].

The data collection for this current study took place over a two-month period at a Fortune 500 company in the U.S. and three of its Indian vendors with whom a lengthy relationship has existed. The interview data reported here was gathered from vendor employees who performed different levels of software testing activities. Interview data was then transcribed and analyzed by the independent researchers. Additional data will be gathered in future phases of this study in order to compare and contrast the best practices from the vendor’s as well as from the client’s perspective, and to understand what influences the differences in levels of success across vendors.

3. Interview Findings

The findings from our interviews with three testing vendors in India are organized across four main themes: Resources and Communications, Evolution of Work and Contracts, Knowledge Sharing, and Relationships and Value Add. For each theme, we include a discussion of recent academic literature on the topic, followed by a report of findings from our interviews.

3.1. Resources and communications

Research findings suggest one key to good communications in offshoring work is channeling of communication through key intermediaries who had good relationships with main players [17]. Knowledge and experience asymmetries as well as requirements and task characteristics prompt onsite and offshore team members to engage in acts to make sense of their tasks and their environment, which increases the likelihood of congruent and actionable understandings among the parties. These acts not only assist distributed workers in coping with problems of understanding and transferring pre-existing understandings, but they also enable them to co-create novel understandings [31].

Table 1 summarizes our findings from our interviews for this topic. Our interviews illustrate that each vendor takes resource management seriously. Vendor 1 has sophisticated resource estimation tools that it uses and re-calibrates to ensure adequate staff is assigned to a job. Vendors 2 and 3 identified staff retention as an important goal of their businesses. Vendor 2 emphasizes the importance of winning additional work from a client in order to have more work during off-peak testing times for their employees dedicated to the client’s account. Vendor 2 suggests this is more easily accomplished when a client reduces the number of vendors it works with, so that more work can be given to one vendor.

Our interviews also highlight the considerable attention paid to coordination and communication issues. All of the vendors send their own employee representative to work at the client’s location. They expect that employee to work with a single point of contact (SPOC) employed by the client. They also expect to send their employees to the client’s site and would like to see the client send their employees to India. These vendors understand the importance of supporting continuous communication via phone calls and reports. Also, the vendors understand they need to remain flexible, stay connected, keep their onsite representative involved with the client, and the need to report negative news if it is necessary.

3.2. Evolution of work and contracts

Some research regarding the evolution of work describes a two-stage offshoring model where both companies act as bridges in offshoring arrangements. Location may be an advantage, but location alone will not be enough to gain a place in future two-stage offshoring arrangements. Instead, depth of expertise and experience are expected to be even more important for future competitiveness. The expectation is that
two-stage offshoring will eventually become “multi-stage offshoring,” involving a sequence of progressively lower-cost destinations [23].

On the topic of contracts, service level agreements (SLAs) have been found to help in managing outsourcing relationships and in achieving success, but little empirical validation has been done [10]. In fixed-price (FP) contracts, more risk is borne by the vendor; while in time & materials (T&M) contracts, more risk is borne by the client. In the task uncertainty literature, one study examined the effect of the information known during contracting on project profits. Results indicated that the vendor makes higher profits from T&M contracts, controlling for project specific variables such as project type and effort [12].

Another study examined the four major dimensions of contract provisions: monitoring, dispute resolution, property rights protection, and contingency provisions. The study found that considerable diversity exists across contracts in the manner in which contractual provisions are used and described, due to differences in goals, contracting environments, and other transaction-specific characteristics. Pricing terms alone do not fully address the risks associated with outsourcing interdependent processes, and additional safeguards are warranted. Under FP terms, contracting parties may feel more secure about their respective obligations without using as much monitoring as in T&M contracts. However, parties in FP contracts are more likely to protect their investments via additional provisions for dispute resolution and property rights allocation [5].

Another research examined quality and technical processes as well as three communication/coordination mechanisms: project status meetings, number of incremental releases, and number of liaisons. The research also studied other variables: contract type, rework stage, requirements volatility, prior experience, size, and complexity. The study found contract types were mostly: (1) FP and (2) T&M, where T&M is less restrictive, so the vendor is more willing to undertake additional changes to the system being developed. Results indicated that FP contracts result in less rework; the assumption is that the vendor resists requests leading to further rework since they bear the burden of cost overruns. The number of client site liaisons tends to increase rework level because more people at the client site must reach consensus on open issues. Reaching consensus may require additional rework in order to come to an agreement. Technical processes (e.g., product planning, product engineering, requirements management, configuration management) reduce effort, and quality processes tend to increase effort. Both communication and coordination mechanisms (project status meetings and incremental releases) increase the effort required. Incremental releases cause clients to want more features added (not rework of an existing system). Project status meetings may clear up some issues but may also generate new ideas, thus increasing effort [13].

Software metrics are defined as a set of on-going organizational processes required to define, design, and implement an information system for collecting, analyzing, and disseminating measures of software processes, products, and services. Recommendations are that upper management should recognize the importance of metrics programs, should formally adopt the programs, and enable an adaptive process. Management should also set up an internal environment that motivates software managers to invest in it. Management must provide resources and demonstrate support. This research distinguished between two stages of metric adoption: Adaptation, that stage in the implementation process in which an innovation is developed, installed, and maintained; and Acceptance, where the innovation is accepted and used by the organization which is the desired outcome of the investments in new work processes and procedures surrounding the metrics program [11].

Key areas in offshore contracts include: definition of work schedules; distribution of onsite/offshore resources; documentation standards; skill requirements and ways to address attrition; stipulations for handling subcontracted work; vendor representation and reporting; payment terms; penalties, bonuses/rewards; dispute resolution procedures; jurisdiction for resolving disputes; security and confidentiality of client data; intellectual property rights [25]. Research has developed and tested a governance model that increases the chances of contractual and relational success. Three key requirements of outsourced processes (interdependence on other processes, complexity, and strategic importance to the organization) should determine three key governance capabilities (the outsourcing contract, relationship management, technical capabilities) [21].

Additional research has found that there are three outsourcing arrangements used as organizations move through various stages of maturity. When moving from the initial stage (business silos) to the second stage (standardized technologies), a strategic partnership, involving negotiated accountability, can help in the transition. When moving to the third stage (rationalized processes), a co-sourcing alliance, i.e., joint project management, has been found to be helpful. And when moving to the final stage (business modularity), the transaction exchange form of outsourcing is beneficial, which is described as having an arms-length relationship [27].

Table 2 summarizes our findings from our interviews for this topic. Our interviews indicate that
all the vendors understand that they are moving to doing more and more work as well as more managerial-level work. Vendor 1 states that clients previously focused on leveraging vendor methods, but today clients want more from their vendors; they want value add activities (see section 3.4. below). Vendors 1 and 2 mention the gradual migration of work offshore after their own employees have spent time at the client’s site.

Our interviews also highlight the considerable attention paid to contract and SLA issues. Vendor 2 notes the relationship is more than just adhering to the contract. Vendors must provide a service regardless of the contract terms. Also, Vendor 2 illustrates the idea that in new relationships, T&M contracts are used, with FP contracts used in more mature relationships. This is consistent with findings in the research literature discussed above. Finally, Vendor 2 states that short-term contracts do not incent vendors to provide better services and adopt innovative solutions. Longer term contracts do allow a vendor to invest in the relationship and initiate new innovations for the work processes performed.

3.3 Knowledge sharing

Studies have found that vendors gain economic benefits from the efficiencies they derive from leveraging a variety and multitude of projects controlled by the vendor across different clients. Vendors can offer benefits that cannot be readily replicated internally by client firms. Vendors’ management practices were found to be dependent on the vendor’s access to many projects. Vendors need scale and scope to reduce the costs of developing their own competencies, and these competencies are used to reduce the cost of delivery on each project [19].

Research has found that the main barriers to offshoring include English language skills, lack of understanding of Western business culture, and problems related to intellectual property protection [15]. Also, transformational technologies are needed for enabling offshoring at the task level rather than functional level. Individuals engaged in task-based offshoring find themselves having to transfer occupational knowledge across time and space. This process has required time they thought they were saving by offshoring tasks and detracted from their perceptions of the effectiveness of the offshoring arrangement. Research indicates that problems may arise in the offshore transfer of technical knowledge within an occupation. Problems occur in transferring knowledge across time and space, and workers involved may need to develop new work practices for knowledge transfer to other individuals employing identical technologies [18].

Offshoring also has an effect on the knowledge supply chain. Short-lived offshore projects may generate substantial cost savings to a domestic firm, but long-lived offshore projects may cause a disruption in the firm’s knowledge supply chain, resulting in substantial cost increases in the later stages of the project. Offshored projects that become unexpectedly delayed during development may be particularly problematic for firms. In this case, the domestic firm may take a wait-and-see approach, possibly delaying the “backshoring” decision (i.e., the decision to return the project to the client site) until it becomes an economically infeasible option. On the other hand, a manager could grossly underestimate the benefits of offshoring by not accounting for potential transfers of knowledge from the vendor firm [4].

Table 3 summarizes our findings from our interviews for this topic. Our interviews show that Vendor 1 has dedicated non-billable buffer resources to the client in order to build their employees’ knowledge. Vendor 2 believes in making sure both parties send employees to the other location in order to transfer knowledge. Vendors 2 and 3 both indicate that they have created knowledge repository portals for their employees to access and learn about the client’s business context. Vendor 3 explains that it is important to make sure they understand their client’s business needs in order to provide better services.

Our interviews show that considerable attention is paid to initiating and generating new innovations. The vendors appear to believe that, in the services industry, incremental improvement is more probable than breakthrough innovations. Vendor 1 works with consultants and universities, has dedicated non-billable staff working on innovations, and shares best practices among its employees. Vendor 2 also hires consultants, awards employees for innovation ideas, holds focus groups with clients, and shares best practices among its employees. But Vendor 2 also realizes they must have visibility to how their client’s business works (i.e., see the “entire eco-system”). It takes several business cycles of working with a client before innovations can be generated, and innovation is a necessity in order to keep costs down. Vendor 3 also shares best practices among its employees, knows innovation is a necessity, has performed internal projects to improve processes, and realizes innovation is more likely in a long term relationship than a short term one because of the learning curve.

Our interviews also highlight the attention paid to sharing ideas with the client. All of the vendors appear to use proof of concept activities to show clients how new ideas will work. Vendor 1 states that the client has to be willing to take risks, Vendor 2 states it is the onsite folks who are most aware of opportunities, and both Vendors 2 and 3 state that a champion or high-
3.4. Relationships and value add

One study describes the evolution of relationship building between a client and offshore vendor as involving these two phases: courtship and cohabitation [17]. Research has distinguished the institutionalized and emergent social boundaries from time and space and related them to the allocation of various types of capital. Also it has showed that spatial and temporal distance accentuated boundaries and status differences. Furthermore, it shows how the use of economic, intellectual, social and symbolic capital helped participants renegotiate differences in power relations. In making sourcing decisions, factors to consider about providers and location include (a) provider’s ability to attract and retain qualified people in a given location with enough expertise to engage in creative dialogue, (b) provider’s existing business knowledge, and (c) internal managers’ willingness to engage in collaborating with that provider [29].

Another study found that offshore outsourcing has a positive effect on the country’s economy by lowering production costs for technology vendors, resulting in a positive effect on end product costs for consumers. The customer value equation is affected by the overall economic gains. The value or benefit that outsourcing provides was found to be cost savings, improved competition, and access to a larger resource pool [3].

Another study examines the trends in offshoring and the effects on U.S. jobs and the educational curricula. It identifies the types of work going offshore and destination countries and mentions the twofold logic of offshoring: (1) organizations should focus on their competencies while contracting other necessary activities to specialists in those activities, and (2) organizational strategy should include economic arbitrage in order to exploit price differences and profit from them [6].

Research has examined the effect of the degree of trust on collaborative relationships. Starting conditions influence the degrees of formal coordination and control and the levels of performance achieved in early stages of cooperation, and also affect how managers interpret the behavior of their partners. Trust, distrust, and formalization tend to develop along self-reinforcing paths. Neither trust nor formalization should become ends in themselves. Governance should be accompanied by performance assessments. Managers should refrain from inter-organizational cooperation when starting conditions are unfavorable, as well as break off relationships that are characterized by negative self-reinforcement in early stages of cooperation [30].

Research has also addressed the predominant types of partnerships for international outsourcing of services and factors that are important for strength and longevity of partnerships. Three types of partnerships were identified: tactical, strategic, and transformational. Businesses have to decide whether to consider vendors as cost reducers vs. value enhancers, opportunistic agents vs. helpful stewards, suppliers of services. vs. active collaborators, all of which depends on the nature of the client-vendor relationship. Two moderating factors that might affect the client-vendor relationship were discussed – trustworthiness and cultural distance [16].

Another possibility is referred to as captive offshoring, where firms set up their own centers in other countries and maintain full control [16]. Also, two keys to vendor competitiveness were identified: (1) “bestshore” policies of choosing the best site for providing services, and (2) a global service delivery model [24].

Table 4 summarizes our findings from our interviews for this topic. Our interviews indicate that Vendors 1 and 2 recognize that long-term relationships are needed. Vendor 2 states the need to know the client better as “closer collaboration needs closer rapport” and also states that the onsite team is the key to a good relationship because they have the visibility to the client operations. Vendors 1 and 3 mention that they have started doing testing work before the official testing stages begin in order to preempt any problems.

Our interviews also highlight the attention paid to the need for value add activities. All of the vendors understand they must find ways to add value to their clients’ operations. Vendor 3 offers their global resources and expertise as ways to add value, but recognizes that they will need to take a bigger role in the management of the work to make this happen.

4. Discussion and Conclusions

The current project gathered and analyzed interview data from three major testing vendors located in India and reported the findings related to four useful themes: Resources and Communications, Evolution of Work and Contracts, Knowledge Sharing, and Relationships and Value Add.

This paper represents an intermediate phase of a multi-phase research project. Prior work involved a qualitative interview phase with internal offshoring management of a Fortune 500 company to gain insights about offshoring followed by a survey of a broader set of managers at other Fortune 500 companies. The results of the current phase of the study have provided valuable insights about various issues regarding managing offshore outsourcing of
software testing from the vendors’ perspective. These results also serve as input for the next phase of this research. The next phase of this project will involve finding gaps in our knowledge about how to manage vendors from a comprehensive literature review. Then, additional qualitative interview data will be gathered in order to compare and contrast the best practices from the vendor’s as well as from the client’s perspective, and to understand what influences the differences in levels of success across vendors. Once all data is completed, the goal is to develop a model of the best practices perspective in building successful long-term relationships between offshore vendors and global enterprises.

5. References


[18] Leonardi, P. M., and Bailey, D. E. “Transformational Technologies and the Creation of


Appreciation is expressed to the following for their support of this research project: Systems Testing Excellence Program at the University of Memphis and the Center for International Business, Education and Research at the University of Memphis.

Table 1: Notes from Vendor Interviews on Resources and Communications

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Resource Management</th>
<th>Coordination and Communication</th>
</tr>
</thead>
</table>
| 1      | • We do 30 days, 60 days, and 90 days ahead test plan to predict resource needs.  
• Get advanced knowledge of changes like changing testing tools, so we can be ready and have the same tool up and running. We roll over and invest in whatever the client is using at no charge to the client.  
• We have a testing estimation tool and collect baseline number to ensure we are on track.  
| • Onsite representative is paired off with SPOC at client, onsite folks spend time clarifying.  
• Client project managers are not aware of all the contract terms, they cannot see people we overcome this by having review meetings every 6 months first at their site, then a when they came to India they could see how we were doing things.  
• It must be a partnership; we must remain flexible and not talk about the contract.  
• Requirements come in at less than 100% accuracy which carries over to the development groups. In critical test stages, we have daily calls for clarification, submit documents, applications to understand the context, and build artifacts when needed.  
| 2      | • We have a 70% to 30% ratio of offshore to onsite staff at our clients.  
| • We must have a point of contact for each project we do with a client.  
• We have calls every morning Indian time.  |
• Staff retention is important. Our staff is not just doing test work, but should be able to write code, etc. We need extra work so we can schedule jobs and keep our folks busy. Also, we rotate people across the different projects we have for one client which also helps with knowledge transfer.
• We need additional work to smooth out the workload as testing tends to spike in its workload.
• Clients need to consolidate their list of vendors (have fewer) in order to give enough pie to each vendor in order for the vendor to have the incentive to invest in the relationship.
• During testing, we work 24 hours-a-day and each shift passes on the reporting of items do live calls, weekly calls, huddle calls with client management, and join client’s test product status reports with defects labeled as high/medium/low.
• We have good remote connectivity.
• The client needs to come to visit us in India more often. Sometimes information gets lost by the time it gets to us in India. It gets re-translated along the way. Another client did this.
• Clients need to consolidate their list of vendors (have fewer) in order to give enough pie to each vendor in order for the vendor to have the incentive to invest in the relationship.
• Big challenge.
• We have been increasing our investment in hardware, but we need to decrease it in people.
• To shrink the testing time, we brought in a consultant who has been helping us to use more automation.
• We need to continue understanding how to plan capacity better.
• We need to retain our staff and do this by developing people with domain understanding and maturity.
• Staff turnover is expected, but we ensure that no one person leaving can impede the project.
• Biggest challenge.
• We have been increasing our investment in hardware, but we need to decrease it in people.
• To shrink the testing time, we brought in a consultant who has been helping us to use more automation.
• We need to continue understanding how to plan capacity better.
• We need to retain our staff and do this by developing people with domain understanding and maturity.
• Staff turnover is expected, but we ensure that no one person leaving can impede the project.
• During testing, we work 24 hours-a-day and each shift passes on the reporting of items do live calls, weekly calls, huddle calls with client management, and join client’s test product status reports with defects labeled as high/medium/low.
• We have good remote connectivity.
• The client needs to come to visit us in India more often. Sometimes information gets lost by the time it gets to us in India. It gets re-translated along the way. Another client did this.
• We have input into the requirements documents.
• We share our metrics showing how we are doing in order to increase visibility. We explain that it is because of their system issues that impeded our work. It brings awareness to them they would not know. Everyday we send a report at the end of the day India time. We have a dashboard to increase visibility of this information.
• We need to continue understanding how to plan capacity better.
• We need to retain our staff and do this by developing people with domain understanding and maturity.
• Staff turnover is expected, but we ensure that no one person leaving can impede the project.
• Regular monthly calls with the client to discuss issues. We have weekly video conferences.
• We have a person involved in the testing process all along the way.
• We have a SPOC to contact.
• We did an inadequate job of communicating in the beginning. We needed to have ongoing improvements and now we do.
• Also, the client has come to India to visit us. Seeing each other is important. We used to have them come at least monthly. People are willing to come from Europe to visit us but not from U.S. because it is considerably farther.
• Onsite person helps make sure there is no miscommunications. We have to have the right people at the customer facing contact. We don’t want the client to get shocked. We try to ensure the client would need. A senior onsite person will know the political scenario and we can communicate it back to us in India.
• During testing, we have daily reports with defect status and any bad news if needed. We tell them medium/low level. We have to tell them the bad news too.

NOTE: SPOC = Single Point of Contact
Table 2: Notes from Vendor Interviews on Evolution of the Work and Contracts

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Evolution Of Work</th>
<th>Contracts and SLAs</th>
</tr>
</thead>
</table>
| 1      | • At first clients just wanted our methodology because they had none, but now they ask for our value add.  
• At first we migrate work gradually offshore, say move 60% for 18 months, then 70% with the goal of 85% over a 2 year people and we reduce the price as we go. | • Clients use them to raise the bar. |
| 2      | • We plan/scope the work, then ramp up our resources, then stabilize the work processes.  
• We start with our folks onsite. It takes time to complete the knowledge transfer before we could scale up to being able to take the work offshore.  
The client lets us shadow them then we move the offshore. | • We think outside the contract, it is not just about the SLA/RFP.  
• We start with T&M because we didn’t know enough about the work and what it was going to take to get it done successfully. Now we do so we can plan accordingly and we went to a FP contract. FP is better for the client, we share the risks, it allows for repeat business.  
• At our own cost, we work extra hours, have moved to more automation, created a leadership level dashboard which was necessary for us to meet our SLA targets.  
• The contract is for 2 years so we must innovate. If the contract was for 6 months, then we won’t bother to innovate. To invest in new ideas, we have to have a stake in it. FP contracts with SLAs cause vendors to innovate. |
| 3      | • Moving toward getting more involved in creating the test plans, to actually being in charge of its creation. It takes time for the client to get confidence in our team. | • It may mean staying late, and the team does not leave until their work is complete.  
• The client also works extra time.  
• I feel that I work for the client.  
• Because our contract is SLA based, we have adopted process measuring tools and made tools built for our clients. |

NOTE: T&M = Time and Materials; FP = Fixed Price; SLA = Service Level Agreements; RFP = Request For Proposals
Table 3: Notes from Vendor Interviews on Knowledge Sharing

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Knowledge Retention</th>
<th>Initiating and Generating New Innovations</th>
<th>Sharing Ideas with the Client</th>
</tr>
</thead>
</table>
| 1      | • Dedicated resources to a client with 10% buffer resources not billable but able to rotate in if someone leaves. | • Work with consultants and universities.  
• Have a delivery team who is billable and serving clients and a service team who is not billable but dedicated to finding new processes.  
• We have a testing school and Intranet to share best practices.  
• In the service industry, you cannot have break-through innovations, but will have incremental improvements. | • Client must be able to show value add before earning more work on new products.  
• Need long term knowledge trust between us.  
• We need to have visibility of how the business works.  
We need to know the entire eco-system so we can foster innovations.  
We have to have visibility to know the current ways the client is doing things in order to suggest improvements.  
• We have an innovation award to our staff.  
• Before we can innovate, we have to go through 2-3 cycles to figure it out first.  
• Best practices are shared among our employees across clients.  
• We hold focus groups with the client for new ideas.  
• We have to generate ideas because it is necessary to make the job less costly and faster.  
We know companies outsource to save money.  
Since we cannot charge more, we have to find ways to reduce our costs, while keeping quality high. | |
| 2      | • We need adequate knowledge transfer and to do this you need to be close by (distance-wise).  
• We have created a knowledge management portal for our employees to understand the client. | • Ideas come from 90% evolution (incremental ideas of improvement) and 10% new breakthrough processes.  
• When a process needs improvement, we hire consultants (say, an investment banker for a financial services client) and they bring skills beyond testing that reflect the business context to help.  
• We need to have visibility of how the business works.  
We need to know the entire eco-system so we can foster innovations.  
We have to have visibility to know the current ways the client is doing things in order to suggest improvements.  
• We give an innovation award to our staff.  
• Before we can innovate, we have to go through 2-3 cycles to figure it out first.  
• Best practices are shared among our employees across clients.  
• We hold focus groups with the client for new ideas.  
• We have to generate ideas because it is necessary to make the job less costly and faster.  
We know companies outsource to save money.  
Since we cannot charge more, we have to find ways to reduce our costs, while keeping quality high. | • Onsite folks know what the client is capable of doing, so the onsite folks do it at the client site but need opportunities to apply ideas, so we help them.  
• Proof of concept is fun and sometimes by the client, we do it at the client site but need to know exactly what the problem is we are trying to solve.  
If they buy-in, we offshore.  
It fails about 50% of the time.  
Sometimes the client writing an RFP opening it up for bid.  
There is our desire to bring new ideas to the future, but about 30% of the time we get the new business.  
• Must have a champion client to drive an idea to adoption.  
• We do a proof of concept using the latest tools.  
• We have knowledge sharing where we share ideas across employees.  
• Lower level managers must be interested in their area, so take the ideas to a higher contact at the client. | |
| 3      | • Important for us to have knowledge in order to continue to work and grow the work.  
We need to have knowledge delivered back to us in India and to manage the client’s expectations.  
• We need to know what our client’s business needs are so we can prioritize the work.  
• Onsite people participate in client meetings in order to do this.  
• We have created a knowledge repository all about the client to bring staff up-to-speed.  
• All new testers get 2 months of training, first on tools, then domain and client applications, then it just takes experience. | • Share monthly best practices, use automation to do 24 x 7 work.  
• The client’s products are growing, so we must do more automation, we need to make profits.  
We must increase automation, decrease people and decrease costs.  
• We have an internal project now to analyze the day-in-the-life of a tester to find the value add steps.  
We analyze the interactions and formally document them to reduce inefficiencies in the process.  
• Innovations in short term relationships are not easy, because you need a benchmark to start with.  
You need to know what each other are capable of.  
In long term relationships the knowledge of each other allows for innovation.  
Over time the testing process becomes routine, then you can innovate. | • We do a proof of concept using the latest tools.  
• We have knowledge sharing where we share ideas across employees.  
• Lower level managers must be interested in their area, so take the ideas to a higher contact at the client. | |

NOTE: RFP = Request For Proposals
### Table 4: Notes from Vendor Interviews on Relationships and Value Add

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Relationship Building</th>
<th>Value Add Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>• Need a successful project for long-term relationships.</td>
<td>• Bring efficiency of service and being a technology partner to launch new products and services.</td>
</tr>
<tr>
<td></td>
<td>• We run test automation macros before testing begins so there will be no show stoppers.</td>
<td>• We try to decrease costs and increase quality for our clients.</td>
</tr>
<tr>
<td>2</td>
<td>• Closer collaboration needs closer rapport. We need the client to come to India.</td>
<td>• Consistency of process, speed-to-market, driving costs down, variable capacity, and timely metrics and reporting.</td>
</tr>
<tr>
<td></td>
<td>• For some clients, we have their people work out of our offices so they really understand how we get work done.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The onsite team makes or breaks the relationship. They have the increased visibility into the client operations. They create the friendships and insights needed to find new ideas. This relationship grows slowly over time.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>• Because of the compressed testing schedule, some testing now takes place before official testing begins. This way we can give the client results beforehand and the managers get information ahead of time.</td>
<td>• We have global resources to pull from to deliver and meet our value proposition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• We need to do the management of the services in order to do this well.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• We bring a different expertise to augment the client’s staff.</td>
</tr>
</tbody>
</table>

### Offshore Vendor Management: Results from a Recent Survey of Managers

Judith C. Simon and Robin Poston  
*University of Memphis*  
jsimon@memphis.edu

**Abstract**

This paper represents the second phase of an offshore vendor management research project. Phase one was a qualitative phase involving interviews with internal offshoring management personnel of a Fortune 500 company to gain insights about offshoring, with results presented at the 2007 Workshop on Advances and Innovations in Systems Testing. Those results served as a basis for the content of a survey instrument developed for Phase two. In this second phase, we have gathered data through surveys of additional organizations, both large and small. The results provide levels of agreement on several factors regarding offshore vendor management.

1. Introduction

In Phase one of our research on issues related to offshore vendor management, we gathered and analyzed interview data from one Fortune 500 company and reported on ten important areas of consideration when working with offshore outsourcing vendors, including:

1. Contracting with vendors
2. Offshoring strategy and cost-benefit
3. Expectations of vendor performance
4. Management of vendors
5. Offshoring issues with vendors
6. Technology infrastructure issues
7. Management within the vendor company
8. Vendor level of services provided
9. Testing work issues
10. What testing work to send offshore

Related research was reviewed as an additional basis for content. This prior research has examined...
various facets of managers’ opinions of outsourcing activities. Surveys of outsourcing clients have provided studies of the advantages and disadvantages of global outsourcing [7] and the length of time managers expect these situations to persist [1]. Other studies have examined the predisposition of vendors to be helpful and trustworthy in their work [4]; [3] and the level of knowledge sharing that is actually taking place [2]; [4]; [3].

Research has also examined managers’ feelings about their relationships with vendors [8], their level of cultural similarity with vendors [4]; [3], who has the clear authority for certain tasks [2]; [3], and whether the vendors maintain dedicated project staffing [2]. In addition, surveys asked about the quality of communications [3], level of coordination [3], ability of vendors to perform project scoping [2], and their ability to take charge of the work processes [2]. Also, studies have examined the outcomes of outsourcing regarding service quality output [6] and opinions about vendor performance [5]; [8].

The second phase of this research attempts to corroborate the previous findings from Phase one and from previous research for the Mid-South region of the U.S., by collecting relevant data from managers at multiple companies via an online survey. Most of the previous studies examined offshore outsourcing rather generally, and none addressed software testing explicitly. This paper also addresses this gap in the literature.

2. Data Collection Methodology

This offshore vendor management survey was conducted to capture multiple clients’ current perceptions of vendor issues and overall performance in a variety of areas. Participation in this study was limited to persons in the Mid-South area who have experience with offshore vendors. The number of usable responses received was 65. Most of the participants (about 2/3) were from organizations with at least 25,000 full-time employees. About 1/3 were from transportation/warehousing organizations, with the remainder scattered among industries that include manufacturing, finance, retail, professional services, health care, and hospitality.

Ten categories of topics were used for this study, with participants asked to identify their level of agreement with each item. A scale ranging from 1 to 7 was used, with 1 representing “Strongly Disagree” and 7 representing “Strongly Agree.” The statements vary as to expected direction of the result; in other words, there is no expectation that participants should always be on the “Strongly Agree” end of the scale.

3. Survey Findings

For purposes of this paper’s report of findings, each item is listed below exactly as stated in the survey, with the average score shown for each item. With the 7-point scale used for this survey, it is possible to look at the average score for each item and easily determine whether it was above or below the midpoint of 4, as well as the strength of the direction (how far above or below the midpoint).

The first category dealt with clients’ perceptions of the pros and cons of offshore outsourcing. The topics were sometimes stated positively (e.g., as an advantage), and sometimes were stated as a problem. As indicated in the Category 1 table, only one statement, “Violations of intellectual property rights are a major problem,” had an average below the 4.0 midpoint, and its 3.81 score is not far below that midpoint. This statement was in a negative form, so respondents indicated slight overall disagreement with the statement that this topic of intellectual property rights was a major problem. The greatest level of agreement occurred with the item regarding “Verbal communication difficulties …,” with an average of 5.18, indicating some agreement that this communication topic is a major problem.

<table>
<thead>
<tr>
<th>Category 1. Offshore Outsourcing - Pros and Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scale:</strong> 1 = Strongly Disagree; 7 = Strongly Agree</td>
</tr>
<tr>
<td><strong>Topic</strong></td>
</tr>
<tr>
<td>Lower salaries/wages of information systems professionals in other countries allow significant cost reductions.</td>
</tr>
<tr>
<td>A much larger number of highly skilled professionals is available.</td>
</tr>
<tr>
<td>Overall development time for systems is reduced.</td>
</tr>
<tr>
<td>Greater access to a global workforce is provided.</td>
</tr>
<tr>
<td>Verbal communication difficulties with offshore vendors are a major problem.</td>
</tr>
<tr>
<td>Data communication difficulties with offshore vendors are a major problem.</td>
</tr>
<tr>
<td>Cultural differences with offshore vendors are a major problem.</td>
</tr>
<tr>
<td>Violations of intellectual property rights are a major problem.</td>
</tr>
<tr>
<td>Time zone differences as to working hours of offshore vendors are a major problem.</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
</tr>
</tbody>
</table>
perform more tasks beyond the Service Level Agreement” than would be true for onshore vendors. The greatest amount of agreement was with the statement that offshore vendors “should take more initiative to suggest solutions to technology issues.”

**Category 2. Offshore vs. Onshore Vendor Responsibilities**

**Scale: 1 = Strongly Disagree; 7 = Strongly Agree**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compared with onshore (U.S.-based) vendors, offshore vendors ...</td>
<td></td>
</tr>
<tr>
<td>... should be more willing to work extra hours.</td>
<td>3.27</td>
</tr>
<tr>
<td>... should volunteer to perform more tasks beyond the Service Level Agreement (SLA).</td>
<td>2.84</td>
</tr>
<tr>
<td>... should invest more in improving existing skills to serve us better.</td>
<td>4.09</td>
</tr>
<tr>
<td>... should be better at promoting a positive attitude.</td>
<td>3.92</td>
</tr>
<tr>
<td>... should take more initiative to suggest solutions to technology issues.</td>
<td>4.63</td>
</tr>
<tr>
<td>... should adjust the work times of their day as needed to participate in meetings with onshore staff.</td>
<td>4.51</td>
</tr>
</tbody>
</table>

Participants were asked to indicate which service involved the most direct contact with their primary offshore vendor. The next category, Software Testing Issues, was provided only for those participants who indicated that Application Testing was the service provided by their primary offshore vendor. There was complete agreement on one item. All participants strongly disagreed with the statement that “test leadership is appropriate to send to an offshore vendor,” resulting in a 1.00 score. Strong disagreement (average = 1.20) also occurred with the statement that “all software testing work can be sent to an offshore vendor to perform.” The statement that “test execution is appropriate to send to an offshore vendor” received a relatively high score of 6.50 on the “Strongly Agree” end of the scale. Two other statements had fairly strong agreement (6.20 average), as shown in the Category 3 table.

**Category 3. Software Testing Issues**

*(Responses by those who indicated Application Testing as their primary vendor service)*

**Scale: 1 = Strongly Disagree; 7 = Strongly Agree**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>We have difficulties with offshoring our software testing automation.</td>
<td>2.80</td>
</tr>
<tr>
<td>We have continual increases in software testing work and send it to our offshore vendor(s).</td>
<td>6.20</td>
</tr>
</tbody>
</table>

**Category 4. Project Management Issues**

*(Responses by those who did not indicate Application Testing as their primary vendor service)*

**Scale: 1 = Strongly Disagree; 7 = Strongly Agree**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>We have continual increases in project work and send it to our offshore vendor(s).</td>
<td>4.40</td>
</tr>
<tr>
<td>Projects sent offshore involve complex work.</td>
<td>4.33</td>
</tr>
<tr>
<td>Offshore vendor staff should be involved in all phases of the project.</td>
<td>4.56</td>
</tr>
<tr>
<td>The offshore vendor must deal with continuing changes in project work as project deadlines approach.</td>
<td>5.07</td>
</tr>
</tbody>
</table>

The Project Management Issues category was provided to all participants who were not provided with the Software Testing Issues statements. The statements in this group are almost identical to some of the statements rated by the software testing group. The primary difference in statements was that those doing Software Testing Issues had additional statements specifically related to that topic. The statement with the lowest average score (greatest amount of disagreement) was “project leadership is appropriate to send to an offshore vendor,” a statement very similar to the one with the greatest amount of disagreement for Software Testing Issues above. The next lowest average score for “all project work can be sent to an offshore vendor to perform,” was very similar to the second lowest average score in the Software Testing Issues results above. For Project Management Issues, the statement with the highest average score was “the offshore vendor must deal with continuing changes in project work as project deadlines approach.” The similar statement in the Software Testing Issues results above was the second highest score. These results indicate considerable consistency as to agreement/disagreement regardless of whether the focus is on software testing specifically or on project management issues.
All project work can be sent to an offshore vendor to perform.  
Project execution is appropriate to send to an offshore vendor.  
Project leadership is appropriate to send to an offshore vendor.

The lowest average in the next category, Client-Vendor Relationship, involved the statement that the offshore vendor “and our organization have similar corporate cultures.” However, the 3.26 average does not represent particularly strong disagreement with the statement. The highest average (5.13) was for the statement that the vendor “invests time in building a good relationship with us.”

Category 5. Client-Vendor Relationship  
Scale: 1 = Strongly Disagree; 7 = Strongly Agree

<table>
<thead>
<tr>
<th>Topic</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>This offshore vendor …</td>
<td></td>
</tr>
<tr>
<td>… makes decisions beneficial to us under any circumstances.</td>
<td>3.60</td>
</tr>
<tr>
<td>… is willing to provide assistance without exception.</td>
<td>4.24</td>
</tr>
<tr>
<td>… is always sincere.</td>
<td>4.43</td>
</tr>
<tr>
<td>… invests time in building a good relationship with us.</td>
<td>5.13</td>
</tr>
<tr>
<td>… has a common or joint sense of mission and purpose with us.</td>
<td>4.65</td>
</tr>
<tr>
<td>… knows our needs well.</td>
<td>4.42</td>
</tr>
<tr>
<td>… cooperates with us to help do the job well.</td>
<td>5.02</td>
</tr>
<tr>
<td>… provides valuable feedback.</td>
<td>4.43</td>
</tr>
<tr>
<td>… provides recommendations for continuous improvement on an ongoing basis.</td>
<td>4.00</td>
</tr>
<tr>
<td>… works as a team with us.</td>
<td>4.94</td>
</tr>
<tr>
<td>… and our staff have no difficulty in understanding each other’s business rules and forms.</td>
<td>3.89</td>
</tr>
<tr>
<td>… and our organization have similar corporate cultures.</td>
<td>3.26</td>
</tr>
</tbody>
</table>

In the category of Vendor Responsibilities, there were no average scores below the midpoint of 4.0, indicating no average scores on the Disagree side of the scale. The highest level of agreement (5.79) was for the statement that the vendor is responsible for “meeting our standards.”

Category 6. Vendor Responsibilities  
Scale: 1 = Strongly Disagree; 7 = Strongly Agree

<table>
<thead>
<tr>
<th>Topic</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>This offshore vendor is responsible for …</td>
<td></td>
</tr>
<tr>
<td>… including flexibility in contracts to accommodate changes in scope.</td>
<td>4.63</td>
</tr>
<tr>
<td>… specifying clearly the work that each party (vendor and client) is to perform.</td>
<td>4.71</td>
</tr>
<tr>
<td>… honestly about problems if delays or other problems occur.</td>
<td>5.48</td>
</tr>
<tr>
<td>… completing the work with little disruption to our operations.</td>
<td>5.56</td>
</tr>
<tr>
<td>… solving problems with little need for involvement from us.</td>
<td>4.52</td>
</tr>
<tr>
<td>… keeping staff turnover low during the project.</td>
<td>5.37</td>
</tr>
</tbody>
</table>

Regarding Communications, three of the average scores were below the midpoint, indicating a tendency toward the Disagree side of the scale. The lowest average (3.46) was for the statement suggesting that oral communications “… are more effective than written communications.” A similar average (3.50) on the Disagree side occurred with the statement that “communications are not affected by differences in time zones.” On the Agree side of the scale, the highest average (5.63) was for the statement indicating that they have frequent contact with the vendor.

Category 7. Communications  
Scale: 1 = Strongly Disagree; 7 = Strongly Agree

<table>
<thead>
<tr>
<th>Topic</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communications between us and our primary offshore vendor are accurate.</td>
<td>4.65</td>
</tr>
<tr>
<td>Communications are complete.</td>
<td>4.41</td>
</tr>
<tr>
<td>Communications are timely.</td>
<td>4.63</td>
</tr>
<tr>
<td>Communications are credible.</td>
<td>4.67</td>
</tr>
<tr>
<td>Oral communications with this offshore vendor are more effective than written communications.</td>
<td>3.46</td>
</tr>
<tr>
<td>Communications are not affected by differences in time zones.</td>
<td>3.50</td>
</tr>
<tr>
<td>Technology problems have not been encountered regarding data communications.</td>
<td>3.71</td>
</tr>
<tr>
<td>We have frequent contact with the vendor.</td>
<td>5.63</td>
</tr>
<tr>
<td>Most problems are solved through mutual discussion.</td>
<td>5.16</td>
</tr>
<tr>
<td>We have coordination mechanisms to solve problems with the vendor.</td>
<td>5.38</td>
</tr>
</tbody>
</table>

In the Information Sharing category, none of the average scores fell below the midpoint, and none were particularly far above the midpoint. As shown below, the highest average (4.86) was for the statement, “Business knowledge of core business processes is shared between us and this offshore vendor, if necessary.” The topic with the lowest average (4.12) is a somewhat negative statement regarding “limited ability to see how this vendor gets the work done,” the result indicates some agreement with this statement.

Category 8. Information Sharing  
Scale: 1 = Strongly Disagree; 7 = Strongly Agree

<table>
<thead>
<tr>
<th>Topic</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business knowledge of core business processes is shared between us and this offshore vendor, if necessary.</td>
<td>4.86</td>
</tr>
</tbody>
</table>
We exchange information with this offshore vendor that helps in business planning. 4.55
Environmental information that affects our two businesses is shared between us and this offshore vendor. 4.67
We have limited ability to see how this vendor gets the work done. 4.12

All of the statements in the category of Vendor Performance on Specific Activities were written in a positive form, i.e., a listing of items that would generally be expected or desired. The averages suggest that performance on these activities is at least minimally acceptable, since no average score was below the midpoint. There were no particularly high scores, e.g., above a 6.0; the highest average (5.46) occurred with the statement that the vendor “works to build a good relationship with us.”

Category 9. Vendor Performance on Specific Activities

<table>
<thead>
<tr>
<th>Topic</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>... is accurate in estimating the contract scope.</td>
<td>4.50</td>
</tr>
<tr>
<td>... is able to handle scope changes adequately.</td>
<td>4.69</td>
</tr>
<tr>
<td>... clearly defines the vendor and client roles.</td>
<td>4.90</td>
</tr>
<tr>
<td>... clearly defines the vendor and client responsibilities.</td>
<td>4.94</td>
</tr>
<tr>
<td>... works independently with minimal reliance on us.</td>
<td>4.39</td>
</tr>
<tr>
<td>... assigns adequate staff dedicated to the project.</td>
<td>4.88</td>
</tr>
<tr>
<td>... replaces any departing staff member with someone at least as well qualified.</td>
<td>4.67</td>
</tr>
<tr>
<td>... delivers appropriate and complete documentation.</td>
<td>4.79</td>
</tr>
<tr>
<td>... works to build a good relationship with us.</td>
<td>5.46</td>
</tr>
<tr>
<td>... has staff that provides accurate work.</td>
<td>5.22</td>
</tr>
<tr>
<td>... has staff that provides complete work.</td>
<td>5.29</td>
</tr>
<tr>
<td>... has staff whose work is of a high quality.</td>
<td>5.33</td>
</tr>
</tbody>
</table>

The final category was designed to measure the satisfaction level specifically. The level of satisfaction overall appears to be above the midrange but is not overwhelmingly strong. The highest average score (5.45) was satisfaction with the vendor’s attitude. The only average score below 4.0 was for the statement written in a negative form, “there is always some problem with this vendor.” An average score on the Disagree side of this negative statement is indicative of a positive feeling about the vendor performance, so it is consistent with the other scores on satisfaction level. It does not represent strong disagreement, as the average of 3.42 is not significantly below the midpoint of 4.0.

Category 10. Satisfaction Level of Vendor Performance

<table>
<thead>
<tr>
<th>Topic</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>... attitude.</td>
<td>5.45</td>
</tr>
<tr>
<td>... communication.</td>
<td>4.94</td>
</tr>
<tr>
<td>... reliability of work provided.</td>
<td>5.17</td>
</tr>
<tr>
<td>... relevance of work provided.</td>
<td>5.31</td>
</tr>
<tr>
<td>... accuracy of work provided.</td>
<td>5.15</td>
</tr>
<tr>
<td>... completeness of work provided.</td>
<td>5.13</td>
</tr>
<tr>
<td>... this vendor understands our business objectives perfectly.</td>
<td>4.45</td>
</tr>
<tr>
<td>... there is always some problem with this vendor.</td>
<td>3.42</td>
</tr>
<tr>
<td>... I am delighted with my overall relationship with this vendor.</td>
<td>4.69</td>
</tr>
</tbody>
</table>

4. Discussion and Conclusions

Responses from Mid-South area managers as to their experiences with offshore vendors appear to be similar and confirmatory on some issues when compared with results of some previous published research. For example, offshore outsourcing is seen by Mid-South managers as a way to take advantage of lower salaries globally and the availability of a larger pool of highly skilled professionals. Their responses also confirmed that communication difficulties are a problem area in this region. Also, agreement existed that offshore vendors should take more initiative to suggest solutions.

Several specific topics were addressed in this study, including recommendations for software testing and for project management. Regarding software testing, participating clients who worked with this type of vendor indicated an increase in the amount of software testing being sent offshore. They did not indicate significant difficulties with offshoring of software testing automation. There was also a favorable response toward sending test execution offshore, but they were less favorable about sending either test design or test case creation offshore. They had unanimous agreement that test leadership should remain at the client site rather than going offshore.

Mid-South managers who are more involved with other project management activities rather than specifically with software testing had some similar
responses. They are continuing to increase the quantity of project work sent offshore. However, they are not favorable to the idea of project leadership going offshore. A slightly negative result also occurred regarding the statement that project execution is appropriate to send offshore.

In overall comparisons of Mid-South managers’ responses, some consistency was seen across different categories of topics. For example, “verbal communication difficulties” had the greatest level of agreement in the initial topic regarding pros and cons of offshore outsourcing. This result was consistent with the results regarding the category of Communications, which had several topics with averages below the 4.0 midpoint. In the overall satisfaction level category, “communication” had an average score of 4.94, which is an OK satisfaction level, but it was the lowest score of the six items rated (other items included attitude, reliability, relevance, accuracy and completeness).

Overall, the responses were considered rather positive regarding clients’ perceptions of the performance of offshore vendors. In the statements related to the client-vendor relationship, most of the average scores were at the midrange or higher. Similarly, all of the average scores were above the midrange regarding vendor performance on specific activities. And the same result occurred with the overall satisfaction level, where all the average scores were above the midrange except for a negative statement; the disagreement with that negative statement indicates a positive perception of that topic, thus 100% favorable responses in that category of satisfaction level.

Relationships with offshore vendors are expected to increase in frequency, and the increasing experience may cause changes in the offshore vendor issues and in the client level of satisfaction. Therefore, this study should serve as baseline data and be replicated to track changes and to identify success factors for areas where satisfaction levels increase. In addition, this current survey should be administered to a wider range of participants to provide additional data for comparison across geographic regions.

5. References


Appreciation is expressed to the following for their support of this research project: Systems Testing Excellence Program at the University of Memphis, Center for International Business, Education and Research at the University of Memphis, Center for Innovative Technology Management at the University of Memphis, and PMI Memphis.