The Proceedings of the Fourth 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
FedEx Institute of Technology

**Systems Testing Excellence Program**
Fourth 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 3-4, 2010
Memphis, Tennessee
Session 1
A Model and Simulation of Duplicate Identification Methods’ Cost Effectiveness

Vasile Rus and Sajjan Shiva
Department of Computer Science
The University of Memphis
Memphis, TN, USA
{vrus,sshiva}@memphis.edu

Abstract—We present in this paper a theoretical model and simulation for estimating the cost effectiveness of automated methods that address the task of duplicate bug report identification. The paper argues that precision is an important performance measure for assessing the cost effectiveness of duplicate identification methods and thus their impact on software testing and development processes. This is in response to the incorrect emphasis on recall only, as an assessment measure, in recent work on duplicate identification. We also show that the relation between the precision of a duplicate identification method and the cost necessary to handle false positives, called misses, has a major impact on the cost effectiveness of methods. The simulation shows the necessary precision and false positive handling cost levels for each recall value such that duplicate identification methods become effective. It should be noted that our cost effectiveness model focuses on the worst case scenario.

Software testing; Duplicate defect identification; cost effectiveness.

I. INTRODUCTION (HEADING 1)

This paper models and simulates the cost effectiveness of automated methods to the task of duplicate defect report identification. Duplicate defect report identification is a major problem in defect management. Not all defect reports in a defect repository are unique because often the same problem is discovered by different testers and reported independently. Reports that describe that same underlying problem are called duplicate reports or simply duplicates. The duplicate reports are a natural consequence of the software process itself in which testing comes after design and implementation (Alexandersson & Nyholm, 2004). About 30% of Mozilla reports and about 20% of all reports in Eclipse are duplicates (Anvik et al, 2005). It should be noted that the identification of duplicate reports is one of the most wanted features of next generation bug tracking systems according to a recent survey conducted by Just and colleagues (Just et al., 2009) among several hundred developers and users of three open source projects: Apache, Eclipse, and Mozilla.

The existence of duplicate reports in the defect database could lead to various problems if not handled properly as has been recently documented in the literature (Hiew, 2006; Runeson et al, 2007; Wang et al, 2008). If there is no information recorded in the database that two reports are duplicates, the result can be duplication of work, i.e., two developers analyzing and trying to fix the same defect. On the other hand, duplicate reports can also help in debugging (Bettenburg et al, 2008).

While there is not yet a consensus with respect to what to do with a duplicate (eliminate it completely versus keep it and simply link it to the main defect report), there is clear agreement that they must be identified. There seem to be a preference towards keeping the duplicates as duplicate reports may contain additional, important information, relative to the main defect report (Bettenburg et al., 2008).

We propose a novel model to estimate the cost effectiveness of duplicate identification methods. There are two reasons that motivated our work. First, previous efforts focused primarily on the recall performance measure of their proposed methods. Recall indicates the percentage of true duplicates that are retrieved. We show that the precision of the methods is also important to decide whether a method is cost effective when compared to the traditional approach of manually triaging defect reports. Second, there is no known study that relates performance metrics, such as recall and precision, to the cost effectiveness of the duplicate identification methods. We address these two problems by offering a cost effectiveness model that relates the performance measures, recall and precision, to the cost effectiveness of a method. Our cost effectiveness model uses another parameter, the cost of finding a missed duplicate (false positives) versus manual triaging, and show that the relation between precision and the cost of handling false positives is crucial in assessing cost effectiveness.

The rest of the paper is organized as in the followings. The next section presents related work to the general task of duplicate report identification. Following that, we describe in detail our cost effectiveness model with an emphasis on the three parameters: recall, precision, and cost of handling false positives. The section on Simulation Results presents preliminary results we obtained by simulating different methods by varying the recall, precision, and cost of handling false positives. The Conclusions sections ends the paper.

I. RELATED WORK

The importance of the analysis of defect reports for improving software quality was recently observed by several groups (Anvik et al, 2006; Hiew, 2006; Bettenburg et al, 2007; Runeson et al, 2007; Wang et al, 2008). In particular, Bettenburg et al. (2007) considered automatic quality assessment, but their proposed technique uses only shallow NLP processing and provides suboptimal results. Duplicate
identification gained much more attention (Hiew, 2006; Runeson et al., 2007; Wang et al., 2008), but the proposed techniques again do not use the full potential of latest advances in NLP (Jurafsky & Martin, 2008) and IR (Manning et al., 2008). Specifically, the techniques rely on the traditional vector space model (Salton et al., 1975) to compute semantic similarity of defect reports, which can only detect the similarity if the reports use same words. This is a major limitation as people/testers often use different words, e.g., synonyms, to express the same ideas. Furthermore, the current evaluation methodology for duplicate identification techniques is weak. Some other tasks that our methods could be applied to, defect triaging (Anvik et al., 2006) and time-to-fix estimation (Weiss et al., 2007), have been also approached using text similarity techniques based on the vector space representation. Weiss et al. (2007) retrieve the $k$ most similar defects from the database and then compute an average of the time it took to fix these $k$ defects, measured in person-hours. The results are impressive: the predicted time estimates are off by only one hour, significantly beating some basic methods.

II. THE APPROACH

A. Cost Effectiveness Model

We propose a model that embeds performance measures of the underlying method as well as the cost of handling false positives. The two performance measures are recall and precision. We will define them using a confusion matrix shown in Table 1. $A$ represents the number of true positives, $B$ the number of false positives, $C$ the number of false negatives, and $D$ the number of true negatives. $A+B+C+D = N$ which represents the total number of defect reports in our bug database. We can define recall and precision using $A$, $B$, $C$, and $D$. Recall is defined as the percentage of true duplicates that are correctly predicted, i.e. $A/(A+C)$.

Precision is the number of correctly identified duplicates, i.e. $A/(A+B)$. We also assume that for a given database the number the distribution of duplicate-nonduplicate reports is given by a variable DupPrcnt, the percentage of duplicates out of the whole set of bug reports $N$. We deem DupPrcnt to be a characteristic of a bug database corresponding to a project. Previous efforts worked with bug databases of specific projects and mention the percentage of duplicates in their datasets. On average, there seems to be 25 duplicates out of every 100 defects in open source projects (we took the average of duplicate percentage found in Mozilla and Eclipse by Anvik et al., 2006). This average matches the duplicate percentage in Jalbert and Weimer (2008) who worked with 29,000 defect reports from Mozilla, of which 25.9% were duplicates. Other distributions of duplicates-nonduplicates are possible such the one in Runeson and colleagues (2007) who worked with data from an industrial setting, i.e. non-open-source, Sony-Ericsson projects, in which about 10% of the instances were duplicates. Wang et al., 2008) used a dataset of 220 reports containing 44 duplicate pairs, a somewhat artificial distribution due to the way they created the dataset.

We also define two costs inspired from Jalbert and Weimer (2008). Triage represents the cost of triaging a defect. For each defect labeled as non-duplicate by a duplicate identification method the cost to triaging the defect is Triage. Miss is the cost of handling a false positive, i.e. a bug report that has been mistakenly labeled as duplicate as thus not considered for triaging at all. The Miss cost is larger than the Triage cost as a false positive is viewed as a duplicate by the triager and thus wrongly resolved as DUP. The Miss cost can go up to 10 times or even more the cost of Triage (ad-hoc estimate in Jalbert and Weimer, 2008). It should be noted that this definition of Miss is slightly different from the one proposed by Jalbert and Weimer. This change in definition is needed because we want our cost model to focus on the worst case scenario. Jalbert and Weimer considered a miss only when all the duplicates of a defect and the original defect were wrongly labeled as duplicates. Our definition leads to a worst-case scenario estimate because it considers a miss for any of the duplicates of a defect. The overall cost of triaging $N$ defects using a duplicate identification method would be $Cost = x \cdot Triage + y \cdot Miss$. This would be compared with the traditional approach in which every defect would be triaged for an overall cost of Traditional-Cost = $N \cdot Triage$.

As mentioned earlier, we would like to relate the cost to the performance characteristics of a method, i.e. precision and recall. We show now next how to express the above Cost in terms of precision and recall.
We can rewrite the Cost as the sum of handling the True and False Negatives (each costing Miss, where we assume Miss = Factor * Triage with Factor varying from 1 to 10 in our simulations but larger values are also possible). We will express the Cost in terms of recall and precision because our goal is to study how cost effective different methods are based on their performance. First, we express the number of False Negatives A in terms of Recall and DupPrcnt. From the Recall definition we get that $A = \text{Recall} \times \text{DupPrcnt} \times N$. Similarly, $B = \frac{\text{Recall} \times \text{DupPrcnt} \times N \times (1-\text{Precision})}{\text{Precision}}$ and $C = (1-\text{Recall}) \times \text{DupPrcnt} \times N$. Thus, the Cost is given by the equation below.

$$
\text{Cost} = (N - \text{Recall} \times \text{DupPrcnt} \times N - \frac{1-\text{precision}}{\text{precision}} \times \text{Recall} \times \text{DupPrcnt} \times N \times \text{Triage} + \frac{1-\text{precision}}{\text{precision}} \times \text{Recall} \times \text{DupPrcnt} \times N \times \text{Miss})
$$

If we consider, Miss = Factor * Triage then the Cost becomes:

$$
\text{Cost} = (N - \text{Recall} \times \text{DupPrcnt} \times N - \frac{1-\text{precision}}{\text{precision}} \times \text{Recall} \times \text{DupPrcnt} \times N \times \text{Triage} + \frac{1-\text{precision}}{\text{precision}} \times \text{Recall} \times \text{DupPrcnt} \times N \times \text{Factor} \times \text{Triage})
$$

From the above equation, we notice that the cost is expressed in terms of only recall, precision, DupPrcnt, Factor, N, and Triage. The traditional cost of triaging all N defects is:

\[ \text{Traditional Cost} = N \times \text{Triage} \]

We can now formulate the Cost-Effectiveness score of a method as:

\[ \text{Cost - Effectiveness} = \frac{\text{Cost}}{\text{Cost - Effectiveness}} \]

We can rewrite this using Equations 1 and 2 above as:

\[ \text{Cost - Effectiveness} = 1 - \text{Recall} \times \text{DupPrcnt} - \frac{1-\text{precision}}{\text{precision}} \times \text{Recall} \times \text{DupPrcnt} + \frac{1-\text{precision}}{\text{precision}} \times \text{Recall} \times \text{DupPrcnt} \times \text{Factor} \]

The Cost-Effectiveness tells us how much lower (or higher) the cost associated with triaging a set of bug reports using an automated method is compared to manually triaging the bug reports. Assuming DupPrcnt being a constant for a particular bug database associated with a project (=25% in our simulation described later), the Cost-Effectiveness only depends on three factors: recall, precision, and Factor parameters that tells us how much more expensive it is handling a False Positive versus manual triaging.

\[ \text{B. Theoretical Bounds on The Effectiveness of Automated Methods for Duplicate Identification} \]

We now present an analysis that shows the theoretical bounds of a method’s effectiveness. True Positives have zero cost associated with them and this is the net cost benefit that automated methods provide us. However, this benefit may be offset by the cost of handling the False Positives given the fact the Miss can be significantly larger than Triage.

From the last Cost-Effectiveness equation above we observe that the second and the third term describe the cost savings while the last term describes the penalty of using an imperfect (precision<1) automated method. For a method to provide any cost savings the sum of the second and third terms must exceed the penalties incurred by the last term as shown in the inequality below.

\[ \text{Recall} \times \text{DupPrcnt} + \frac{1-\text{precision}}{\text{precision}} \times \text{Recall} \times \text{DupPrcnt} \frac{1-\text{precision}}{\text{precision}} \times \text{Recall} \times \text{DupPrcnt} \times \text{Factor} \>

This can be rewritten as:

\[ \frac{1}{1-\text{precision}} > \text{Factor} \]

In other words, a method becomes cost effective only if Factor is less than 1/(1-precision). Interestingly, the above equation does not really depend on recall but rather on precision, largely ignored by others in their studies of automated methods for duplicate identification. The dependence only on precision of the bounds of a method cost-effectiveness indicates the theoretical (and practical for that matter) importance of this performance measure. From the equation, we infer that the higher the precision the larger the Factor can be and thus a method can be effective even if the costs of handling false positives is much larger than the traditional Triage costs. When precision asymptotically reaches 1, Factor can have an arbitrarily large value.

\[ \text{III. Simulation and Results} \]

Given the above model, we focused on the three major parameters of our cost model: recall, precision, and Factor. We simulated an entire space of methods with various combinations of recall (values ranging from 0.1 to 0.9 in increments of 0.1) and precision values (similar to recall) as well as projects that have various values for the Factor.
parameter (values ranging from 1, meaning the cost of handling a false positive is equal to the traditional cost of manually triaging a defect report, which is Triage, to 10 meaning it costs 10 time more to handle a false positive). For each recall level, we observed the precision and factor values that maximize effectiveness. The values for the most cost-effective methods are shown in Table ???.

<table>
<thead>
<tr>
<th>Recall</th>
<th>Precision</th>
<th>Factor</th>
<th>Cost-Effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Any</td>
<td>1</td>
<td>0.97</td>
</tr>
<tr>
<td>0.2</td>
<td>Any</td>
<td>1</td>
<td>0.94</td>
</tr>
<tr>
<td>0.3</td>
<td>Any</td>
<td>1</td>
<td>0.91</td>
</tr>
<tr>
<td>0.4</td>
<td>Any</td>
<td>1</td>
<td>0.88</td>
</tr>
<tr>
<td>0.5</td>
<td>Any</td>
<td>1</td>
<td>0.85</td>
</tr>
<tr>
<td>0.6</td>
<td>Any</td>
<td>1</td>
<td>0.82</td>
</tr>
<tr>
<td>0.7</td>
<td>Any</td>
<td>1</td>
<td>0.79</td>
</tr>
<tr>
<td>0.8</td>
<td>Any</td>
<td>1</td>
<td>0.76</td>
</tr>
<tr>
<td>0.9</td>
<td>Any</td>
<td>1</td>
<td>0.73</td>
</tr>
<tr>
<td>0.99</td>
<td>0.99</td>
<td>1</td>
<td>0.70</td>
</tr>
</tbody>
</table>

Table ???. Results for Factor=1 which means the cost of Miss is the same with the cost of Triage.

<table>
<thead>
<tr>
<th>Recall</th>
<th>Precision</th>
<th>Factor</th>
<th>Cost-Effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.9</td>
<td>2</td>
<td>0.97</td>
</tr>
<tr>
<td>0.2</td>
<td>0.9</td>
<td>2</td>
<td>0.95</td>
</tr>
<tr>
<td>0.3</td>
<td>0.9</td>
<td>2</td>
<td>0.92</td>
</tr>
<tr>
<td>0.4</td>
<td>0.9</td>
<td>2</td>
<td>0.89</td>
</tr>
<tr>
<td>0.5</td>
<td>0.9</td>
<td>2</td>
<td>0.87</td>
</tr>
<tr>
<td>0.6</td>
<td>0.9</td>
<td>2</td>
<td>0.84</td>
</tr>
<tr>
<td>0.7</td>
<td>0.9</td>
<td>2</td>
<td>0.81</td>
</tr>
<tr>
<td>0.8</td>
<td>0.9</td>
<td>2</td>
<td>0.79</td>
</tr>
<tr>
<td>0.9</td>
<td>0.9</td>
<td>2</td>
<td>0.76</td>
</tr>
<tr>
<td>0.99</td>
<td>0.99</td>
<td>2</td>
<td>0.70</td>
</tr>
</tbody>
</table>

Table ???. Cost-effectiveness for a Factor=2, which is the first Factor at which a method becomes cost effective (Cost-Effectiveness=1) for current state-of-the-art methods that yields approximately recall=0.60 and precision=0.60.

IV. CONCLUSIONS

We described a cost-effectiveness model for automated methods to duplicate identification. The model relies only on the performance characteristics of the methods and on the factor relating the cost of handling false positives relative to manually triaging a defect report. We simulated our model by generating an entire range of performance and factor values. The simulation revealed that current state-of-the-art methods to duplicate identification (recall=0.6, precision=0.6) can only handle development environments where it costs at most twice (Factor=2) to handle false positives related to manual triaging. When Factor=10, we learned that the precision of automated methods must be in the upper 90s for any level of recall to become effective. Our proposed model and simulation emphasize the importance of precision as a performance measure for duplicate identification methods. These finding is extremely important as the tendency in the research community has been on identifying as many true duplicates as possible (high recall) while not paying much attention to precision.

<table>
<thead>
<tr>
<th>Recall</th>
<th>Precision</th>
<th>Factor</th>
<th>Cost-Effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.1</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>0.2</td>
<td>0.2</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>0.3</td>
<td>0.3</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>0.4</td>
<td>0.4</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>0.6</td>
<td>0.6</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>0.7</td>
<td>0.7</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>0.8</td>
<td>0.8</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>0.9</td>
<td>0.9</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>0.99</td>
<td>0.99</td>
<td>10</td>
<td>1.00</td>
</tr>
</tbody>
</table>

ACKNOWLEDGMENT

The work of Rus and Shiva was sponsored by The University of Memphis under a Systems Testing Excellence Program (STEP) project. The opinions expressed in this paper are solely the authors’.

REFERENCES


Extending the Development of Test Cases in Conditions of Large Numbers of Variables and Values

Mark L. Gillenson, Ph.D., University of Memphis
Martin Dinstuhl, University of Memphis
Scott Fischer, FedEx Corporation
Thomas O. Meservy, Ph.D., University of Memphis

Abstract
For software that requires a large number of input variables and/or a large number of values per variable, generating an exhaustive set of test cases is clearly infeasible. A well-known heuristic technique for generating a set of test cases that is both efficient and effective is pairwise testing. However, even pairwise testing has its limits as the number of input variables and/or the number of input variable values is very large. This research describes two efforts to aid in this problem. One is lessons learned in developing an effective test bed for experimenting with different test case generation techniques. The other is the development of a new concept for extending pairwise testing through exploratory testing guided by genetic algorithms.

1. Introduction
Developing targeted test cases is one of the most critical tasks in software testing. In all but the most trivial applications, the vast number of possible combinations of input variable values makes exhaustive testing with all possible combinations infeasible. Thus, we seek techniques that will derive a limited set of test cases that give us confidence that they are capable of finding most defects. We speak of the set of test cases as being both efficient – limited in number – and effective – able to find most defects. A standard technique for generating test cases under these circumstances is pairwise testing. But even pairwise testing has its limitations in the presence of a large number of input variables and/or a large number of variable values.

In this research, we approached the problem of extending pairwise testing in the presence of large numbers of variables and variable values from two different directions. In one, we developed a university/corporate test bed approach that can be used to test new concepts in generating test cases. The intention was to use this test bed arrangement to test the ideas proposed in Gillenson, etal, 2009. In the other, we propose a methodology for extending pairwise testing by using the test cases it produces as a base for further exploratory testing using genetic algorithm techniques.

2. A University/Corporate Test Bed
In Gillenson, etal, 2009, we discussed several approaches to extending the efficacy of pairwise testing (Copeland, 2004) and of base choice coverage (Ammann and Offutt, 1994, Ammann and Offutt, 2008, Sherwood, 1994). The basic concept was generating test cases by using pairwise analysis or base choice testing on a subset of the variables while freezing the remaining variables at a single value each. A suggested variation was to take one of the "excluded" variables and combinatorially including a small subset of its values in the mix. Yet another variation was to perform pairwise analysis or base choice coverage on more than one subset of the variables. The last variation was to perform pairwise analysis or base choice coverage on overlapping subsets of the variables.
In order to test these ideas we developed a university/corporate test bed arrangement between The University of Memphis and the FedEx Corporation. We sought a FedEx application or portion of an application that was self-contained, had several but not too many input variables, and had some input variables that had a limited number of discrete values. Furthermore, we required the software to be at a stage in which it was known to have a reasonable number of defects, i.e. it was at a pre-test or early-test stage. In consultation between the two organizations, the application chosen was the Global Ship Manager (GSM.) GSM satisfied the conditions regarding the variables and we had access to an early version of it that reportedly was known to contain code defects.

GSM's purpose is to determine whether or not a package's characteristics qualify it for the assignment of a tracking number based on the company's business rules. Thus, the software has only two possible values: "pass" indicating that a tracking number can be issued, or "fail" indicating the negative. We identified 13 input variables, which for our purposes was a workable number, not too large or too small. Some of the variables, such as Total Weight, are continuous; some such as Origin City and Destination City are discrete but have many possible values. All of these are good candidates for being frozen. We focused on three variables for pairwise analysis using the Microsoft Pairwise Independent Combinatorial Testing (PICT) tool: Service Type (x values), Packaging Type (x values), and Special Handling (x values.) Running these three variables with all of their values through the PICT tool and freezing all of the remaining variables at a single value each produced xx test cases.

The arrangement between The University of Memphis (U of M) and FedEx Corporation (FedEx) was that U of M would select the variables and values, run them through the PICT tool, and therefore produce the set of test cases. Based on the business rules, FedEx would determine the expected output for each test case and run the test cases through the application software. FedEx would then provide the pass/fail results of running the test cases to U of M which would then do further analysis, modify the variable and value selections as described earlier, and repeat the process. The ultimate goal was to begin to understand what kinds of variations in the variables and values were the most useful. (Incidentally, "pass" in this sense means that the application correctly issued a tracking number or correctly declined to issue a tracking number; "fail" means the application incorrectly issued a tracking number when it should not have or did not issue a tracking number when it should have.) The version of the application being used was deliberately an early version that was known to have defects and was being held on a test machine at FedEx. The application software was not subject to being modified in any way. The concept was that the initial set of test cases and their results would form a baseline for further experimentation.

Unfortunately, as often happens in research, we ran into an unexpected problem: the software was in better shape than we expected it to be. The baseline set of test cases produced no defects. We tried several variations, including changing the straightforward frozen value choices for origin country and destination country from France and Italy, respectively, to the hopefully more problematical Ireland and Hong Kong, respectively, but to no avail. We could not make the software fail and we eventually ran out of time.

We were successful in determining that the kind of university/corporate partnership that we established, with the division of labor as described herein, worked well and can be a model for future research. We further learned that simply knowing that an application has defects in it is not sufficient for its use in this kind of
experimentation. Defects can come in many shapes and sizes, ranging from code that violates business rules to interface problems with other applications and so on. Even if there are defects that violate business rules, it is detrimental to this kind of exercise to have to take "shots in the dark" to find the defective portions of the software. What we recommend for future research is an environment in which a programmer who is familiar with the code can deliberately introduce defects into specified areas of the code involving specific variables. Variables and variable values can then be chosen that will target the known defective code. In this way, we can experimentally see which approaches to variable and value selection using pairwise analysis and its derivatives work best.


Data mining (Berson and Smith, 1997, Han and Kamber, 2001, Tan, etal, 2006) is a collection of techniques that seeks to find patterns in data. Derived from the field of visual pattern recognition, data mining today is most often used in such endeavors as finding patterns in marketing data for further targeted marketing efforts. In particular, two data mining techniques, neural networks and genetic algorithms, have found their way into software testing.

Neural networks, or "artificial neural networks" are algorithms that can be trained to recognize patterns in data. Fundamentally, they employ three levels of variables: inputs, outputs, and an internal level of "black box" variables with weights that can be adjusted with training sets of data. Experiments with using neural networks in software testing have been going on since the 1990's (see for example Anderson, etal, 1995, and Vanmali, etal, 2002.) The purpose is to have the neural network "learn" which portions of the software tend to be defect-prone.

Genetic algorithms (Berry and Linoff, 2004, LaRose, 2006) are based on the biological principle of making changes to genes to evolve the species. They have been used in pattern recognition and data mining to evolve from a known result to a new one to determine if the new result is superior to the known result. Genetic algorithms have been used in software testing in several ways. In one, generally known as mutation analysis or mutation testing (see, for example, Offutt, 1995) defects are deliberately introduced into the code of a working program. If a test case can be produced that finds the introduced defect, then that test case becomes useful in testing the finished code. If that test case passes, then the defect is not present.

Genetic algorithms have been put to a fairly wide variety of uses in experiments in software testing (Berndt, etal, 2002, Miller, etal, 2006, Pargas, etal, 1999, Srivastava, etal, 2009, Sofokleous and Andreou, 2008.) As our interest in genetic algorithms centers around pairwise testing, we particularly note the work of James McCaffrey, who recently developed a system that uses a genetic algorithm to generate sets of test cases that satisfy the pairwise testing paradigm (McCaffrey, 2009.)

For our part, we have conceived of what we believe to be a new way to use genetic algorithms in software testing by extending the test cases produced by pairwise analysis in an exploratory fashion. Exploratory testing (Jorgensen, 2008) is a broad term that generally implies that testers run test cases through software and then, based on the results of those runs and their knowledge of the software and of the business requirements, devise additional test cases to further "explore" interesting (perhaps defect prone) parts of the software.
Pairwise analysis is a heuristic that experience has shown generally produces an efficient and effective set of test cases. The number of test cases produced is much smaller than the maximum number of combinations of all of the possible values of all of the variables. Experience has shown that if defects exist, the pairwise analysis test cases are likely to find many, but not necessarily all, of them. That being the case, a logical extension would be to slightly modify the pairwise analysis test cases to see if these modified test cases can pick up remaining defects that the original set missed. In a parallel sense, this follows the logic of testing on and around the boundaries of equivalence classes. A slight variation of a good test case may also be a good test case.

Following this logic, our idea utilizes the genetic algorithm concept of mutation in an exploratory fashion to produce additional test cases. The mutation might, for instance, be a pairwise analysis test case modified by the change of a single variable to a different, nearby value. An experienced tester might want to mutate a pairwise analysis test case that passed to see if a slight variation will make the software fail, or might want to mutate a pairwise test case that failed to see if a slight variation will continue to make the software fail. In either case, the tester is seeking to probe the software and discover the origins of the failures within the code. Such additional exploratory testing could be automated by using actual genetic algorithms, which select the “fittest” test cases, i.e. for our purposes those that discover defects, for further mutation.

Another genetic algorithm technique is known as "crossover." In this technique, two genomes are broken at the same position in their sequencing and the corresponding halves are switched to create two new genomes. Our idea here is to apply this technique to pairs of test cases produced by pairwise analysis. If such test cases consist of seven variable values then, for instance, we might split each of two test cases after the fourth value and interchange the remaining three values between the two test cases and then run them through the software again.

4. Conclusion

We believe that we have identified two useful results from this research. In one, we have developed a workable university/corporate framework for software testing experimentation and found some practical "dos and don'ts" in doing so. In the other, we have conceived of a new method of test case generation, combining the concepts of pairwise analysis, genetic algorithms, and exploratory testing.

References


Role of the Testing Group in Selecting an Enterprise Architecture Solution: A Case Study

Abstract—Software testing groups are playing an increasingly prominent role in both the software development lifecycle (SDLC) and in the long-term planning of technology architectures that support organizational information systems. The advent of integrated enterprise architectures (EA) provides new opportunities for testing groups to play a proactive role in building consistent and testable guidelines for improving enterprise-wide software quality. Given that testing groups historically have not been invited to participate in EA decisions, there is little academic literature or industry best practices on approaches that testing groups might use to guide their participation. This article draws lessons from the experience of a Fortune 100 corporation whose testing group used theoretical notions of “testability” to guide its involvement in an EA acquisition process. It describes how it operationalized testability criteria, incorporating controllability, observability, and simplicity, into various stages of the process and illustrates the benefits and challenges of taking such an approach.

V. Keywords- Enterprise Architectures; Testing; Development Lifecycles; Case study

VI. Introduction

Today’s large scale organizations are globally dispersed and highly complex entities which interact frequently and often in real-time with other enterprises that are also dispersed in many different locations. As organizations continue to grow and become more collaborative and networked, integrated enterprise architectures (EA) are becoming more popular and necessary (Camarinha-Motos and Afsarmanesh, 2008). A well designed EA will apply systematic, holistic and rational methods to the design of an organization’s data and technology infrastructures in order for information technology (IT) to more effectively and efficiently pursue its purposeful business activities. EA is a master plan of the organization covering business planning with goals, visions, strategies, and governance principles; business operations with business terms, organization structures, processes, and data; automation with information systems and databases; and enabling technologies with computers, operating systems, and networks. The challenge however is how to secure an EA that provides standardization benefits while resolving localization challenges created by complex global business operations involving diverse products, markets, and technologies.

The role of the testing group in software development has focused on how to test complex integrated systems where testing activities were
relegated to one of the later stages in structured development approaches (i.e., waterfall model). As a result, many software products are conceived, initiated, and designed with little thought as to how they will be tested or even be testable in the later stages of development. Consigning testing to the later stages of lifecycle creates a multitude of problems including: too much work in too little time, how to fix problem code and design flaws, and software modules that are more difficult to test than needed (Gelperin and Hetzel, 1988). Companies are therefore moving testing activities to the beginning stages of the software development lifecycle (SDLC). The hope is that by applying testing earlier in the lifecycle, software teams will be able to catch bugs when the fix is cheaper (Zhu et al., 1997). However, this movement to early testing is challenged by incompatible architectures, tools, and platforms that make it difficult. The advent of prototyping and newer more agile approaches to software development (i.e., SCRUM, Adaptive Software Development, Crystal Methods, and eXtreme Programming) has also not made the situation better because of similar reasons. The recent popularity of integrated EA that strive to provide common technological platforms for both IT development and operational purposes, therefore holds promise for having testers engage earlier in the SDLC. A key requirement for this is that testing groups must engage even beyond the start of the SDLC. They must be actively engaged from the very start in the conceptualization, design, and acquisition of the EA itself.

Activities within the SDLC, like testing, are directly influenced by the manner in which the EA is set up. How software will be developed and tested will be based on how the integrated EA is designed. Thus, the testing group must have input into the strategic discussions about the overall EA that will serve as the basis of software development and testing down the road. This article explores the following research question: How can a software testing group in a large global organization play a proactive role in the definition, design, and vendor selection process for acquiring an EA that will serve as the focal point for future IT activity?

VII. LITERATURE REVIEW

A. Enterprise Architecture

An EA description gives a holistic, systematic description of an enterprise. It encompasses business functions, business process, people, organization, business data, software applications, and computer systems with their relationships to enterprise goals. Enterprise model methodologies and tools allow the user to represent, visualize, understand, communicate, redesign, and improve the operations of the enterprise (Chapurlat and Braesch, 2008). EAs use enterprise models to focus on timeliness, cost, quality, and speed-to-market. An enterprise model represents the environment, systems, and entity in the physical, social, and logical world of a company (Camarinha-Motos and Afsarmanesh, 2008). The model organizes the activities of constituents, roles for participants, and rules of governance over data, processes, and technology. An enterprise model is a tool for decision-makers, and it is especially useful for those designing and maintaining software systems that support enterprise operations (McGinnis, 2007).

B. Testability

In order to achieve testing goals, the testability of components of software code is critical. The IEEE Standard Glossary defines testability as, “the level at which a component facilitates the formation of test conditions and at the determination through whether those criteria have been fulfilled” (Freedman, 1991). Other definitions of testability include: the probability of uncovering a bug if it is present in the software (introsoftwaretesting.com); greater controllability, observability, and simplicity lead to easier testing (Payne et al., 1997); and the likelihood of the code
failing if something in the code is incorrect (Voas and Miller, 1995).

Testability consists of several notions such as controllability and observability (Binder, 1994). Controllability reflects the testers’ ability to control inputs to the software code being tested and its internal environment (i.e., how the code within it works). Observability reflects the testers’ inspection of the output of the software code being tested (i.e., how well the code performs its job). If the testers cannot control the input or observe the output, they cannot be sure how a given input was processed by the code. Lacking controllability means identical tests of the same code may produce different results. Lacking observability means the testers may think the output of the code is correct because they cannot see that it is not correct (Payne et al., 1997). But when input is controlled and output is observable, testers can determine whether the code is working properly (Bach, 2003; Binder, 1994).

Test automation also impacts testability. According to Fodeh (2003), “Automated testability is the degree to which the application under test facilitates the implementation, execution and maintenance of automated testing.” Testers should consider the:

- Visibility of the output, errors, system interactions, etc. (i.e., applying a glass-box approach);
- Control of the ability to enter input, trigger events, and invoke methods associated with the capability to exercise command over system parts;
- Persistence which is the frequency of change of the software being tested;
- Consistency in the level of coherence in the look, operation, and performance of the software being tested;
- Reliability which pertains to the probability of the system functioning properly; and
- Documentation where good specification of system functionality and interface is required for adequate automated testing (Fodeh, 2003).

Software testability is not a characteristic of source code artifacts alone. From the domain of distributed real-time systems, companies are beginning to understand that the architecture (i.e., high-level design) of a software system can be a main factor of testability. Based on this premise, we extend this view and refer to testability as a characteristic of a software artifact (a system, component, or document) independent of the current abstraction level (Jungmayr, 1999).

The same views can be applied to a company’s strategic software product development infrastructure that handles all aspects of the SDLC. In this environment the software product development infrastructure serves as the single source for the enterprise’s product data and rules while enabling current product aware systems to leverage a common set of product information. Adoption of an enterprise product development infrastructure across the enterprise enables integrated configuration, testing, and deployment methodologies for reducing time-to-market for new software products and their enhancements.

C. Engineering Testability into the SDLC

Design for testability has been proposed as an approach to increase the quality of software development outcomes (Pettichord, 2002). “Testability takes cooperation, appreciation, and a team commitment to reliability” (Pettichord, 2002, p. 25) and focuses on the more human aspects. Other researchers consider testability as the extent to which
a software artifact enables testing in a given test context, with the goal being to reduce the overall testing effort (Jungmayr, 1999). If code lacks testability, similar to other design problems, it is costly to fix when detected late in the SDLC. As a result, testability should be addressed during earlier reviews of development work. This research describes different aspects of testability, offers heuristics to evaluate testability, and illustrates how reviews based on checklists help encourage testability throughout the SDLC (Jungmayr, 1999). Our approach illustrates the managerial processes need to gain buy-in for these endeavors.

Prior research has also addressed testability regarding how to address it in object-oriented development. Testability has been illustrated to be lower in object-oriented software than procedural implementations (Payne et al., 1997). To address these concerns, software design-for-testability is becoming more important. One paper has offered methods to measure testability based on the fault revealing ability of a class-components based on data flow analysis and considering definition and use locations (Kansomkeat et al., 2005). However, this technique is based on the program’s implementation, and it fails to highlight testability issues prior to code development. Our approach compliments this method by urging development teams to consider ways to infuse testability into all the stages of the SDLC, including infusing it into the overall EA.

VIII. RESEARCH METHODOLOGY AND CASE STUDY

We use a qualitative approach in gathering investigative case study data on how testability is being engineered into an EA implementation process. This approach focuses on gathering case study data in order to provide descriptive and explanatory insights into the testability activities engineered throughout the EA acquisition process. This approach allows us to induce a theoretical account of the activities found in empirical observations and analysis of managerial data. This approach is also known to lead to accurate and useful results by facilitating deep understanding and interpretation of the contextual complexities of the environment in the research analysis and outcomes. Finally, this approach encourages an understanding of the holistic systematic view of the issues and circumstances of our situation, that of engineering testability into the entire EA acquisition process from different development perspectives (Yin, 1994).

We selected a large Fortune 100 organization, herein called LogiCo, known to have successful software development and testing activities. The organization has approximately 5,500 IT employees, with 3 high-level managers, 14 middle-managers, and 149 full-time employees in the testing organization. The software development team (including the testing organization) has 4 major local and global roll-outs of hundreds of new and updated software versions a year. We were able to capture the composition of the development teams’ relationships, the history and background of software development efforts, and the nature of practices used to foster successful outcomes from their endeavor to engineer testability into an integrated EA. LogiCo is adopting an EA solution using a product lifecycle management (PLM) methodology. The following topics delineate the important facets of testability when attempting to engineer testability into the EA.

A. Engineering Testability into the EA Definition and Vendor Selection Process

LogiCo offers integrated logistic services, which are highly automated, to the global marketplace. As a result the services, which are viewed as software “products” as they are referred to herein, are based on software code that encapsulates business data and rules. LogiCo is looking at increasing its software
product development speed-to-market through a centralized software product management technology platform that can be integrated into the overall LogiCo IT infrastructure. This will enable holistic new product development processes and leverage a standard modular design for all global logistics products and related features of their services which are supported by the IT infrastructure.

Prior to pursuing an integrated EA, LogiCo began with a diffused and decentralized IT architecture platform, which reflected the high growth that the company experienced over the prior two decades, which did not permit central architecture planning, decision making, or IT maintenance and support (e.g., there was no central data dictionary that existed for the myriad operating companies that LogiCo owned). This led to a proliferation of programming languages, architecture, and interfaces, etc. As a way to overcome this complexity, an initial corporate initiative was established to explore technological consolidation and integration. An integration and standardization program or corporate initiative to consolidate technology tools and enterprise solutions was pursued. These efforts impacted the testing group who argued for a common architecture and tool set for both development and testing. After three years, these efforts morphed into a higher level EA initiative. An EA team was established with key representatives from all pertinent business and IT sub-units with top management oversight and support. The EA definition and selection process used is illustrated in Figure 1 in the Appendix.

Initial meetings of the EA team explored EA through sandboxing and adopted a PLM approach given that services could strategically be viewed as structured software products. An EA “Product” was defined as a base logistic service (generally delineated by geography, transit, and delivery time) supported by software modules with associated options and product support attributes that define a sellable logistics service. A Technical Quality Advisor (herein named Testing Lead) was selected by the Vice President of Software Quality Assurance to participate in the EA definition and selection process. As the Testing Lead considered PLM and related approaches, he pondered questions regarding what does testing have to do with EA and how best could testing engage on the project. He engaged organizational experts, did research, called academics, and consulted with his testing leadership team for guidance. He decided that to facilitate quality assurance and reliability any new EA must ensure that all software products developed must meet basic testability criteria. It was made imperative that theoretical notions of testability be derived for practical application and injected into EA requirements white-paper documents, request-for-proposals, scoring criteria, and proof-of-concept stages of the EA acquisition process.

In a decision that was approved by strategic management, the EA team decided LogiCo needed to locate a solution in the marketplace that will fulfill the Enterprise Product Manager (EPM) or configurator role. The capabilities needed spanned Product Data Management and Product Lifecycle Management (PLM) including product configuration with a focus on the intersection between product data and lifecycle management to support their EA vision. Currently, software product information was decentralized throughout multiple functional systems. The company needed the future EPM to be a single source of valid and verified software product information for all of the company’s logistics products across the enterprise.

Figure 2 (see Appendix) illustrates the key components of LogiCo’s EA strategy that seeks to increase speed-to-market through three complementary dimensions adopted across the enterprise:
Modular Product Design—Redesigning the company’s products in a way that is modular and works with the LogiCo’s IT infrastructure,

Technology Platform—Building an EA software solution that allows business users to effectively and efficiently configure and manage company products, and

IT Enablement Services—Retooling IT to accommodate the outputs of the EA solution.

Modular product design of LogiCo’s product framework was necessary so that business professionals could design products using building blocks (or tested coded modules) as opposed to defining a product in a silo approach. This allows for a more uniform option-driven approach to product development rather than a “start from scratch” approach for every new product or enhancement which is supported in the EA. Once established, product modularization offers flexibility to manage features within a product and across a product portfolio (i.e., effective up-feature or de-feature). In addition, modularization offers flexibility in defining base price and/or fee structures (i.e., what features are included in the base rate and what features are offered for a fee) of a new or existing product. Figure 3 in the appendix illustrates how LogiCo is moving from an IT platform that supported software products that needed considerable integration testing efforts (i.e., built using a silo approach) to one that is more flexible based on a modular design.

The EPM approach for their EA uses for the logistics software products decomposes them into modules that are made up of attributes, values, and business rules. This hierarchy is used multiple times throughout the EA business requirements document and describes the various “items” managed within the EA product data structure. Rules can apply to any level in the product architecture and vary in purpose. For example, portfolios may be defined by geography, which could set a series of rules (e.g., Mexico doesn’t allow dangerous goods). While another rule could apply at the attribute level and may state a link to another attribute for example (e.g. alcohol requires an adult signature). Figure 4 also in the appendix shows how the modularity in the EA design structure will facilitate testability of products and components.

The new EA product structure and module design will be configured and maintained within the EA software platform. An illustration of how these components will interact is shown in Figure 5 in the appendix. The EA solution includes several key requirements, including data and rule management comprising:

- **EPM Catalog**—data management for the complete list of modules, attributes, and values;
- **EPM Library**—data management for the complete list of products grouped into portfolios and available for sell in production systems; and
- **Product Rule Management**—rules management applied throughout the product and module design and configuration process and maintained in the EPM Catalog, EPM Library and Configuration workflow.

In addition, the EA solution will include components that involve workflow, visualization, analysis and reporting, comprising:

- **Product Lifecycle Management (PLM)**—visibility to product definition information across portfolios to enable effective product lifecycle management feature analysis with the ability to enact efficient changes to product definitions at the appropriate life-cycle phase;
- **Product Configuration**—product definition design, project management, and change governance workflow; and
- **Release Management**—publication management of product and module introduction, variation and change to LogiCo production systems, including quality assurance procedures, through
varying techniques (e.g., versions or effective
dates).

The goals of testability for the EA solution will be
bound by certain key principles. At LogiCo, the overall
goal of testing goals in the EA definition was to inject
quality early in the concept and definition phases.
This was done by articulating the following set of key
principles to the EA team and winning acceptance for
their inclusion in the EA acquisition process:

- Testability is comprised of controllability,
  observability, and simplicity heuristics;
- EA will be used to inject testability principles for
  building “quality in” earlier in the SDLC by its
  inclusion in the request for proposals from EA
  vendors and used as a criterion in their selection;
- Incorporating testability principles in
  requirements, architecture, interfaces, design,
  and coding activities will lead to higher quality
  software products created early in the SDLC; and
- Decoupling is a very important aspect of
  testability that must be factored into the overall
  architecture design.

A key achievement was that LogiCo’s enterprise
architects accepted these arguments and have
adopted testability principles in the EA proof of
concept and white-paper writing activities.

In order to accomplish the objectives of
making the EA initiative a reality, the Testing Lead was
assigned to work with the business, IT core, and
extended EA teams. These teams were asked to
understand the current state of PLM in the industry
and participate in benchmarking sessions with leading
providers and their customers that adopted this
technology platform. At the beginning of this process,
the sessions focused on vendor presentations of EA
products as well as stakeholder interviews with key
LogiCo business and IT representatives to discuss
product feature sets and understand technology
platforms. The key role of the Testing Lead in these
sessions was to listen, learn, and understand the
industry PLM capabilities for product development,
while defining and espousing the role of “testability”
for the new EA. At the end of a three-month period,
the team narrowed down the list to ten providers.
The team then conducted a stringent “request for
proposal (RFP) process” where testability assessments
were designed, conducted, evaluated, and
recommended for the top two vendors who were
selected to execute a proof-of-concept (POC) with
LogiCo.

B. Engineering Testability into Vendor Selection

This section discusses how testability was assessed
in vendor selection processes. The EA team started
by looking at industry reports examining how well
different vendors supported a PLM environment,
configuration capabilities, and an integrated suite of
software services. When examining the industry
reports, none included measures of testing and
testability. The Testing Lead defined and crafted
testability requirements in the RFP documentation
that was sent out to selected vendors who were
asked to respond to the RFP and provide product
demo evaluations. Figure 6 in the appendix shows the
key accomplishments as part of this evaluation
process. Testability was built into the EA definition
through the RFP requirements stating the need for
integrated validation testing of products, modules,
attributes, and rules within the EA itself and the need
for enabling testing of a software product once it was
configured. Testability was built into vendor selection
though explicitly including scoring of testability
requirements which were weighted 6.25% of the
overall 30% assigned to the IT criteria as well as by
selecting the top two vendors that met or exceeded
the testability requirements.

Figure 7 in the appendix illustrates the key
testability heuristics that were designed into the RFP
and vendor EA product demonstration activities. The
main attributes of testability considered important in
selecting an EA were: controllability, observability,
and simplicity. The next section considers each of
these attributes and explains what makes them important to consider in the EA and PLM environment and how they were incorporated into the RFP process.

As illustrated in the left columns of Figure 8, key testability principles for controllability, observability, and simplicity were included in the RFP requirements. A key RFP requirement involved ensuring there is a mechanism provided for validating the configured EA (EPM to LogiCo) software module via a dedicated product test environment. This also ensures that there is intrinsic validation capabilities engineered within the EA architecture, which allows the product developer tool to validate the EA modules and their attributes defined by their business rules. The right columns of Figure 8 depict the actual feature sets that were observed for the corresponding RFP requirement that were evaluated in the EA vendor demos.

Testability can also be directly included in IT decoupling processes using the testability attributes. This in essence facilitates on-going integration and systems testing even in situations where particular modules do not meet specified criteria. An integrated EA facilitates this if controllability, observability, and simplicity are built into the architecture. Figure 9 in the appendix depicts how these were approached in relation to the EA acquisition processes at LogiCo.

Ideal IT decoupling plans can be implemented as follows:

- Without code being backed out,
- Are testable at all test levels (integration/systems/functional testing) and testing cycles,
- Allow flexibility for business and operational constraints,
- Are clear and concise – reads like a disaster recovery plan, and
- Have flows depicting the ‘decouple’ points.

IT decoupling strategies are being conceived for the EA. In the context of the EA, let’s say a new software product is defined as composed of A, B, and C. Some number of days prior to launch, it comes to light that a portion of B doesn’t work and a decouple effort must be initiated. Can the product now be described as A, B1, and C (where B1 is some working version of the code)? Impact assessments can be initiated to ensure B1 doesn’t negatively impact the product launch. Likewise, B1 may mean that the product launch is delayed and must be “switched off” for launch. The capability to “switch off” could be a mainstream capability for product and sub-product definitions. The goal is to have the flexibility to have certain products, certain countries, or certain features switched on or off.

There should be capabilities to decouple EA products by product type, feature, attribute, module, and rule sets. There should be decoupling capabilities provided in the product testing environment. One way to decouple is to rollback to the current version of that specific module, attribute, and rule and publish that back real-time into the EA while re-configuring the stated product in the EA and republishing the output of the decoupled re-configured product back into the product testing environment. Once this is done, regression and new functionality test cases must be executed and validated on the re-configured/decoupled product/feature/attribute/rule as the case applies to ensure it is not broken. Testing strategies will need to be further defined to explore various decoupling testing scenarios. Given the requirements, two EA vendors have been selected for POCs.
IX. IMPLICATIONS AND CONCLUSIONS

The EA acquisition process is still an ongoing activity at LogiCo. Figure 10 (in the appendix) illustrates the testability elements tied to the integrated EA definition and vendor selection for a solution. The testing group is positioning the testability notions in the EA architecture whitepaper as a key driver for the planning phase deliverables.

A. Building Quality in Earlier through the Testing Groups roles as a Business Technology Leader, Test Architect and Test Advisor

Building quality in early is the goal for injecting testability into the earliest stages in the SDLC. The Testing Lead not just focused on testability but took the time to understand the business landscape of the EA and the associated technologies and processes. While the testing group is not always included in the definition/development/launch phases of an SDLC, this case study illustrates the importance of including representatives from the testing group in an EA decision from its inception. Software quality promoted by the testing group needs to become the driving force for enterprise business and IT deliverables that are developed, managed and used in high performance software and product development organizations.

On the EA team were Enterprise IT Architects, Enterprise Data Architects, Business Leads, IT Leads, Project Managers and extended team members from IT and Product Development from Marketing. The EA team members supported the testability principles and practices that were researched and proposed by the Testing Lead. Based on the experiences of this case example, the testability materials put forward to the EA team were not met with concerns but were embraced and ensuing discussions were related to how testability could best be applied to promote software quality earlier into the EA processes. There was strong cooperation by all team members once the testability concept was understood. In fact, the EA team encouraged and assisted the Testing Lead to create testability best practices. As a result, EA project team meetings could be leveraged as a training ground to help others understand the principles and help evangelize the inclusion of testability into the appropriate EA vendor deliverables (e.g., RFP, BRS, Scoring Models, Vendor demos, POC sessions, Whitepaper).

One unique outcome of the Testing Lead’s involvement with the EA team was his influence on this corporate enterprise strategic initiative for the selection, design and adoption of the EA suites, platforms, and architectures. Through this process, the Testing Lead was able to coin the term “Enterprise Test Architect (ETA)”, much in line as a counterpart to an Enterprise IT Architect. The ETA role was created as a senior position in the organization and treated on par with equivalent management positions in terms of rewards, recognition, visibility and influence. However, one basic factor that distinguishes a TA from a Manager is the absence of direct-responsibility for managing people. The future expectation is for the ETA to influence, mentor, coach and provide direction to members of the testing group. Key responsibilities expected from this new ETA role involve:

1. **Strategy** -- providing technical leadership and strategic direction to the testing group
2. **Quality** -- is the foremost technical authority and is responsible for the overall quality of deliverables across all parameters, both functional and non-functional including performance, security, usability, etc. Testability can be foreseen to be one of the major functional responsibilities
3. **The Big Picture** -- maintains a “big and complete” picture of the product, its dependencies, organizational goals, technology arena, etc. and helps guide and direct the functioning of the testing group appropriately
4. **Influencer** -- influences the business’ future direction, strategy and planning as it relates to software products

5. **Collaborator** -- collaborates effectively and on an on-going basis with all constituents involved in software product development and release activity including development, testing, technical publications, marketing, program management and other entities to ensure execution and deliverables

6. **Testability** -- is involved as a business technologist in understanding business requirements and works with the development architect to translate requirements into solution architecture designs and incorporates testability as part of overall designs. Reviews requirements and seeks clarity as required, participates in product design reviews and works with the development architect and development team to make any design improvements and refinement as needed

In summary, this paper presented a testability approach that was used by a Fortune 100 corporation to ensure that the EA it was adopting would support testing efforts aimed at assuring high standards of software quality. Through this case study, we illustrate that design for testability was a central theme and the company expects it will result in a better EA adoption as long as the selected EA framework can follow the four defined software product development principles:

1. **Comprehensive**: covering all applicable EA products
2. **Rigorously accurate**: all aspects of software product definitions are fully accurate
3. **Standardized**: all definitions are in standard form, nomenclature and style
4. **Fully described**: Each software product is individually fully described

In order for the EA to be testable at the architecture and design layers, based on the results of this case study, we recommend architecting the following heuristics in order to achieve an EA that is effective, efficient and adoptable:

1. “In-built validation” of the software modules and their attributes that design in mechanisms to promote functional validation before promoting them to ‘end to end’ or ‘black box testing’ techniques
2. “Self-error checking” or exception handling mechanisms to promote “crisp” error messages that enable testers/users/business analysts to debug business rules and enable them to correct and validate
3. “Sand-box” type environment to enable “Self-Testing” of the actual product configuration/development and validation aspects via iterations before it can be released to a software runtime environment

Given all these findings, this paper has reported on the results of a case study to demonstrate how testability criteria were infused at multiple stages of the EA acquisition process. A key implication of the approach espoused in this paper is that testers have a pivotal role to play in the design phases of holistic EAs that serve as the integrated platforms for enterprise-wide software development and IT operations. While the traditional loose coupling and tight cohesion principles of modular design remain important, testers have a responsibility to ensure that the new integrated platforms also facilitate controllability, observability, and simplicity in software development as a means of ensuring testability of applications that are developed. Importantly, this paper demonstrates how testing leaders can play a valuable role in shaping the EA of their organizations.

REFERENCES
APPENDIX

Figure 1: LogiCo’s EA Acquisition Process

## Enterprise Architecture

<table>
<thead>
<tr>
<th>Modular Product and Design</th>
<th>Technology Platform</th>
<th>IT Enablement Services</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EA is about...</strong></td>
<td>...enabling and supporting</td>
<td>...and market competitiveness</td>
</tr>
<tr>
<td>Standardization</td>
<td>Robust product management</td>
<td>Faster feature of service improvements</td>
</tr>
<tr>
<td>Simplification</td>
<td>Complexity reduction</td>
<td>Ability to bundle services</td>
</tr>
<tr>
<td>Modularization</td>
<td>Product development effectiveness</td>
<td>International expansion of domestic services</td>
</tr>
<tr>
<td>Parameterization</td>
<td>Effective integration of acquisition</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Components of LogiCo’s EA Strategy

### 4 key principles for modules:
- Products assembled from modules.
- Modules are a logical grouping of attributes, values, and supporting business rules.
- Modules are designed to optimize reusability.
- Different products can be created by reusing existing modules or modifying select modules.

Figure 3: Description of LogiCo’s EA Modular Design
Figure 4: Modularity in EA Product Architecture

Figure 5: Description of LogiCo’s EA Components
Figure 6: RFP Process and Testability Weighting in Vendor Selection

- Built testable RFP requirements that evaluated features for:
  - Inbuilt validation of products/modules/attributes/rules
  - Enable testing of a configured product within the EA

- Developed scoring methodology and evaluated RFP responses to testing requirements:
  - 6.25% weighting to SQA out of a total of 30% IT score
  - Picked two vendor finalists that met/exceeded testability requirements
<table>
<thead>
<tr>
<th>Controllability (C)</th>
<th>Observability (O)</th>
<th>Simplicity (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The EA consists of a <strong>rules development interface</strong> that can be designed to create, export, import, query, examine and test rule sets and their associated attributes and values</td>
<td>The EA consists of <strong>automated business logic triggers</strong> to assess and check the viability of a rule, flag incompatible rules and/or modules</td>
<td>Enable simple creation of modules and its rules/attributes thereby promoting <strong>functionality simplicity</strong></td>
</tr>
<tr>
<td>Dedicated product test environment to configure a product and <strong>enable iterations</strong> of the product development process and product validation.</td>
<td>Validate module for its rules and attributes by <strong>inducing error logging and correction mechanisms</strong> that enable the user to debug error messages</td>
<td>Evaluate products for <strong>structural simplicity</strong> to easily assemble new products via access to a EA catalog and library</td>
</tr>
<tr>
<td>Define, configure, test a <strong>controlled</strong> suite of Enterprise Product use case scenarios and product data definitions</td>
<td>Observe EA XML product data outputs and validate actual vs. expected for consumption to back-end product aware systems (FLAGS, EPIC)</td>
<td></td>
</tr>
<tr>
<td>Out of the box API’s for standard integration to business rule management systems (ILOG, JRules)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing tool sets (API’s for QTP Load Runner Scripts and Automated Regression Testing)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7: Testability Criteria within RFPs and Demos**
Controllability (Architecture)

The better we can control it, the more the testing can be automated and optimized.

RFP Requirement

Testability Principle 1:
Software modules, objects or functional layers can be tested independently.
- RFP Reqmt.: Modular architecture that architects “in-built modules, attributes and products”
- RFP Reqmt.: Capability to test configured products in a “sandbox” type environment

Testability Principle 2:
A scriptable interface or test harness is available.
- RFP Reqmt.: Application should have the ability to provide common standard API’s which can be linked with 3rd party products

Evaluated EA

- Validate module for its rules and attributes by inducing error logging and correction mechanisms that enable the user to debug error messages
- Dedicated product test environment to configure a product and enable iterations of the product development process and product validation
- Out of the box API’s for standard integration to business rule management systems (ILOG, JRules), testing tool sets (API’s for QTP Load Runner Scripts and Automated Regression Testing) and XML type data for consumption to back-end product aware IT systems such as LSSI (FLAGS), EPIC, etc.

Observability (Architecture)

What you see is what can be tested

RFP Requirement

Testability Principle 1: Internal errors are automatically detected and reported through self-testing mechanisms.
- RFP Reqmt.: Design a “table driven approach” of valid built in constraints on product combinations/permutations to automatically detect, report and provide correction of product configuration errors

Testability Principle 2: Incorrect output is easily identified.
- RFP Reqmt.: Ability to perform real-time configuration integrity validation to prevent invalid configuration selections against existing EA business rules

Testability Principle 3: Incorrect output is easily identified.
- RFP Reqmt.: Ability to examine rules/attributes/values at product or module level
- RFP Reqmt.: Ability to query, view, report product/module and specific EA rules and their values for configuration

Evaluated in the EA

- EA consists of a “testing environment” within its framework where a product configuration process can be tested and validated for “self-testing” mechanisms
- Which enable the product management specialists to validate and correct incorrect business rules for a given module or a product itself
- The EA consists of automated business logic triggers to assess and check the viability of a rule, flag incompatible rules and/or modules
- The EA consists of a rules development interface that can be designed to create, export, import, query, examine and test rule sets and their associated attributes and values
**Simplicity (Architecture)**

*The simpler it is, the less there is to test*

<table>
<thead>
<tr>
<th>RFP Requirement</th>
<th>Evaluated in the EA</th>
</tr>
</thead>
</table>
| **Principle 1:** Functional Simplicity (e.g., the feature set is the minimum necessary to meet the requirements)  
- RFP Reqmt: Ability to maintain versions of rules and ability to differentiate between the rule sets and the test levels where they reside |  
- EA promotes functional simplicity by simple creation of modules, rules, attributes using pre-defined rules within the EA or ability to import business rules from external business rules systems |
| **Principle 2:** Structural Simplicity (e.g., Modules are cohesive and loosely coupled)  
- RFP Reqmt: Promote re-usability of new products and modules/attributes via access to an EA catalog |  
- The EA consists of a “single source of truth” library that will consist of housing “productized data and their associated rules, attributes, modules for a transportation product” |

Figure 8: Testability in RFP
Decoupling Criteria for Testability in EA

Effective Data driven - architected so that a configuration change causes the code to be engaged or not

Data dependency - New Logic is engaged if certain data is found

Operationally controlled - Logic is only engaged if the new air-bill is distributed

Parameter Driven - Code is engaged based on customer flag in CHEERS

On/Off Trigger - project is architected in such a way that the code is not engaged

Control a “On-Off” Switch for Product feature Staging via FLAGS - No Code Change

Execute and Validate the “Switch-Off” mechanism at product/module/attribute/level levels based on geography/region to ensure regression functionality is not broken

Design an elegant “Throttle” Decoupling Capability

The EA team is visualizing an Enterprise Wide “data-driven” logic where individual applications do not have to code Decoupling but rather use this common service

Figure 9: Decoupling Criteria and Heuristics for Testability

Controllability (C)

Observe the XML data messages across the Pipe-line and who receives it

Prevent complicated and ambiguous data mapping issues between interfaces

Easily Adopt, Receive and Consume Data and Business Rules

Figure 10: Testability Elements
Session 2
Test-Driven Development in the Corporate Workplace

Ahmed Owian  
FedEx Services  
Collierville, TN, USA  
ahmedowian@yahoo.com

Linda Sherrell  
Department of Computer Science  
University of Memphis  
Memphis, TN, USA  
Linda.Sherrell@memphis.edu

Abstract—Software engineers in industry use many different processes and techniques in an attempt to create quality software. In today’s market, many software developers are using agile practices, especially test-driven development (TDD). What is TDD, and why is it giving traditional software development practices a run for their money? This work answers these questions, while focusing on a popular agile methodology, Extreme Programming (XP). It places a particular emphasis on the exploratory programming nature of XP and its testing practice, TDD. The paper also summarizes prior research on TDD and includes the results from a research survey conducted to compare TDD with traditional testing practices. The survey shows TDD has many benefits; namely, it helps developers better understand their code and increases their confidence concerning whether the product adheres well to requirements.

Keywords: test-driven development; agile; software testing; Extreme Programming

X. INTRODUCTION

A. Agile Software Development

Rapid prototyping and the resurgence of "programming as a craft" led to agile methods [6]. The Agile Manifesto for Agile Software Development (see below) summarizes the values for agile software development [2]. Note the items on the left take precedence over the items on the right, but both are considered valuable.

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

One of the most popular agile methodologies is Extreme Programming (XP) [4]. The next subsection discusses this methodology with particular emphasis on test-driven development.

B. Extreme Programming

Kent Beck, Ron Jeffries, and Ward Cunningham founded Extreme Programming in the late 1990's. As stated by Jeffries [15], the 13 XP practices are the following: pair programming, the planning game, test-driven development, whole team, continuous integration, design improvement, small releases, coding standard, collective code ownership, customer tests, simple design, metaphor, and sustainable pace. The major practices discussed above are not all original to Extreme Programming, yet XP has increased their popularity in the software industry. For example, Larman and Basili [16] note that NASA's early 1960's Project Mercury applied iterative, incremental, and test-first development.

A major XP practice is test-first development, which is also known as test-driven development (TDD). Here the developers use acceptance tests and unit tests to drive their development. They start by breaking down a task into testable units. The developers write an automated test case in code (before writing the production code) to verify what a particular unit's functionality should be. The tests drive development since the production code must conform to the interfaces designed while writing the test. Then the developers write just enough production code to satisfy that test before refactoring and writing another test. Refactoring is improving the design of existing code without changing its functionality or intent. The tests allow the developers to confidently refactor both production code and test code. Since the developers created no production code except to satisfy a test, they know that if all tests pass after refactoring, the code should still work. Any failing test, which was previously passing, helps the developers find a fault as soon as they introduce it into production code. The practice of writing a failing test, implementing the production code to satisfy that test, and refactoring both test code and production code creates a quick, iterative feedback cycle. It should be noted that developers continuously make design decisions when writing tests and when refactoring. This exploratory programming practice produces the useful side effect of an automated regression test suite. As a means of comparison, traditional test-last development is the practice of developers designing up front, writing code until done, and then testing that code to verify that it is correct. Test-last development is the way developers who follow the Waterfall Model or one of its variants usually develop software.

XI. RELATED WORK

Davis and Venkatesh [8] agree with the agile idea of iterative test-first development, but from the users' perspective. They propose that user acceptance tests be
performed even before code is generated. This belief leads to the following questions:

- "What will the user be able to test?"
- "What software artifacts are available before code is implemented?"

According to these researchers, it is widely believed that in order to quantifiably predict user acceptance of a software product, a user must first play with a prototype of the system. They believe that given enough detail of how the system works, end users can actually judge the system's perceived usability and thereby predict "... post-implementation user acceptance in the workplace."

Geras, Smith, and Miller [10] performed experiments to measure how test-first thinking affects both customer and developer tests. They used the following metrics: effort, testing, and failure rate. They believe that "... test-first thinking early in the software process may result in developers understanding the requirements better and therefore fewer post-release defects..."

In [23], the researchers investigated the impact of TDD on program design and code quality. The authors compared three software projects using object-oriented metrics. They found that TDD does not automatically improve code quality. However, it does result in higher code coverage, while not causing the product to be late.

Crispin uses both unit TDD and customer TDD along with other agile practices to produce fewer defects in production code and more satisfied stakeholders. She believes TDD is hard to learn for someone whose mind is already set in traditional practices, and therefore its usage does not immediately generate good results. However, once developers learn how to perform TDD well, better code is the outcome [7]. This is substantiated by [22], as they found students gained a better understanding of their own code and confidence in the software product after practicing TDD, but the acceptance of the practice took some time.

Martin [17] presented the following three laws for TDD practitioners:
1. You may not write production code unless you've first written a failing unit test.
2. You may not write more of a unit test than is sufficient to fail.
3. You may not write more production code than is sufficient to make the failing unit test pass.

Following these laws, a developer can decrease the testing loop (write test case in code; implement code for test case; run test case; refactor code) to a few minutes so that when a test case fails, the developer always knows the location of the fault since he/she just produced it a few minutes ago. This is especially beneficial, as [24] claims debugging can take up to twice as long as coding. Similarly, Jared Richardson, co-author of *Ship It* [19], said debugging, on average, usually takes about half the entire development time [30]. Robert and Micah Martin, the authors of FitNesse, a software-testing tool consisting of about 45,000 lines of code, rarely ever use the debugger, and when they do, it is after a failing test. TDD allows them to clean up code, because they are not afraid to break it.

Other benefits of TDD follow. TDD enforces highly decoupled code to facilitate independent testing of units. TDD also provides unambiguous documentation for the production code through the associated test cases. Therefore, it is important to maintain test code at the same level as production code [3].

Janzen and Saiedian [12] emphasize that the focus of TDD is design and not testing. When a developer writes a unit test, the developer must make design decisions about how the production code will work. Then, using the simplest design, the developer satisfies the test. In order to improve the design and reduce duplication, the developer must refactor. In [16], the authors suggest that when a test fails many times, the developer should break up the test into smaller parts in order to pinpoint where the fault resides. This strategy is different from refactoring, because it actually changes the structure and the purpose of the test cases, whereas refactoring improves both test and production code without changing the functionality. It should be noted that the need for alteration of test cases may be avoided if each test case is written sufficiently small in the first place.

Janzen and Saiedian [12] also claim "Understanding that TDD is more about analysis and design than it is about testing is one of the most challenging conceptual shifts for new adopters of the practice." This view is supported in [22] and [26] where some student researchers integrated TDD into the software development practices of their research groups with mixed results. In [13], the authors determined through a survey that "... mature programmers who try TDD are more likely to choose TDD over a similar test-last approach."

Some researchers [11] conducted an experiment where students created two medium-sized programs using TDD and conventional code development. They found TDD caused the students to be more productive, as TDD decreased the development effort needed to complete the two programs. They also found that regardless of the development style, the code quality depends mostly on the amount of testing applied.

Ambler [1] addresses some misconceptions about TDD including the one where people think that unit testing is the only type of testing to perform. Acceptance testing, where customers with the help of the software development team write the test cases, is also important. Another misconception is that TDD is only applicable for small projects. When projects are large, then the test suites should be divided into more manageable sets of tests.

Janzen and Saiedian [14] studied metrics taken from five teams' Java projects as well as from an industry case study of 2,750 lines of code. Three of the five teams were from industry. The fourth team was composed of undergraduate students, and the fifth one was a team of graduate students. The authors gathered metrics from projects developed using TDD and from those developed using a test-last strategy, and they found TDD has a
positive affect on software design quality. They ascertained that TDD leads to "... simpler classes and sometimes simpler methods" by applying the following metrics: weighted-methods complexity, cyclomatic complexity per method, and nested-block depth per method. They also measured lines of code per method, and methods per class discovering TDD leads to smaller methods as well as smaller classes. In addition, Janzen and Saiedian measured code coverage, and they found in most cases, TDD resulted in more production code covered by automated tests. In order to measure coupling, they took the following metrics: coupling between objects, fan-out per class, average number of method parameters, and information flow. It turned out that code developed using TDD had a higher degree of coupling. The authors suspected this phenomenon was due to the programmers trying to make the code easier to test. Although the authors also measured cohesion, these metrics led to inconclusive results.

Bhat and Nagappan [5] added to the empirical body of knowledge regarding the efficacy of TDD. They showed that professional teams employing TDD at Microsoft produced code of over twice the quality of similar teams not using TDD. They also noted that practicing TDD added at least 15% more development time; however, the tests continued to benefit the teams as executable documentation.

In [9], the authors conducted experiments with professional software developers to test whether TDD yields superior external coded quality and attributes to higher productivity. Their results indicated that TDD practitioners took 16% more time than the control group to implement a simple program, and code quality correlated with time spent. They noted that TDD practitioners tested their code at percentages well above industry standards. They also conducted an opinion survey from which they concluded that "TDD is effective in terms of code quality and improves programmers’ productivity. However, getting into TDD mindset is difficult."

Not all literature reviewed on TDD was positive. In [25], the authors argue that the success of TDD lacks sufficient empirical evidence. They hypothesize that due to the inherent nature of TDD lacking an up-front design, a project developed using TDD may be threatened by a lack of architecture. They also argue the product will be lacking in maintainability and its ability to evolve. The work of [25] is only a proposal for research, and not the result of a study; however it does show that there are some critics of TDD.

A hybrid software development process [6] has some practices adopted from agile methodologies and some from traditional plan-driven approaches. Many software projects today utilize this approach to satisfy different needs of the project. For example, certain work products such as architectural design documents may be required in order for software deliverables to move to production. At the same time, agile methods such as TDD and pair programming may be adopted as best practices due to their many benefits.

XII. COMPARATIVE SURVEY

The authors conducted a survey [18], which can be found in Appendix A to compare TDD and traditional test-last development practices. We placed the survey online in order to reach as many people as possible and to programmatically gather results. The respondents indicated whether they had no experience, 1 year or less, 1-3 years, or greater than 3 years experience with each testing practice. Over 60 respondents completed the survey. However, we filtered out eight responses due to the respondents’ lack of experience with either or both TDD and traditional testing practices.

A summary of the results of some of the questions on the survey appear as bar charts in Figure1. The survey revealed that the majority of respondents liked TDD and found it more useful than traditional test-last development. Some results were remarkable. For example, 90% of all respondents and 100% of the respondents with either greater than 3 years of experience or less than 1 year of experience in TDD indicated that TDD fosters better confidence in the code. Only one respondent answered that traditional coding-then-testing fostered better code confidence. On the question to determine whether code is easier to read and understand using TDD, 67% favored TDD; 8% chose traditional testing practices; and 25% claimed that there is no difference. Interestingly, 92% of all respondents claimed TDD promotes better quality code. The rest of the positive-phrased questions followed this trend.

Some negative-phrased questions also existed in the survey. TDD takes more upfront effort than traditional testing practices, but that effort pays off later with less time needed to fix faults in the code. The survey results indicate 38% of the respondents think that TDD takes more effort, while 27% think that traditional testing practices are more costly; 35% said that there is no difference. The results of the effort question are fairly evenly distributed.

There were no results indicating that TDD was an unfavorable practice. It should be noted that many of the respondents were professionals who attended the Agile Development Practices 2009 Conference. It is possible that the sampling of respondents was a biased group, but since the majority of respondents had greater than 3 years using traditional test-last development, it is also possible that the favorable results simply reiterate prior research [13]. That is, developers who try TDD like its benefits, so they favor it and stick with it.
EXECUTIVE SUMMARY

Prior research has shown that practitioners of TDD benefit from its usage, but adhering to the practice takes discipline and a degree of skill. As TDD is more widely accepted, there will be more available empirical evidence of TDD’s effectiveness. The authors have contributed to this body of research with a survey comparing the effectiveness of traditional testing and test-driven approaches to software development. In conclusion, this research advocates the use of TDD for software engineers to produce maintainable, testable, self-documented and automatically self-verifying software.

REFERENCES


APPENDIX A: SURVEY

This is the survey questionnaire used in this research. The first author posted the survey to spreadsheets.google.com on November 3, 2009, and he compiled the survey responses received through November 20, 2009.

Comparison of Traditional and Test-Driven Development

Traditional (test-last) development is the practice of the developers writing code until done and then the developers testing that code to verify that it is correct. This is the way developers who follow a Waterfall Life Cycle Model usually write and test software.

Test-Driven (test-first) development is the process of writing a test before the code and then writing just enough code to satisfy that test before writing another test. Using this iterative process, the tests drive the design for the code and provide a regression test suite along the way.

The purpose of this research survey is to compare traditional and test-driven development from industry professionals’ experiences. Members of the Department of Computer Science at the University Of Memphis are conducting this survey. Your individual responses to this survey will remain confidential.

Questionnaire:

Choose the best answer for how much experience you have ...

1. using traditional development. [None, less than 1 year, 1-3 years, greater than 3 years]
2. using test-driven development. [None, less than 1 year, 1-3 years, greater than 3 years]
3. developing software in general. [None, less than 1 year, 1-3 years, greater than 3 years]

If respondent indicated "None" for any of the questions above, then the survey takes him to the post-questionnaire. Otherwise, the survey continues.

Choose the process that, according to your experience, ...

4. makes you more productive. (Measure by number of tasks completed.) [Traditional, Test-Driven, No difference]
5. promotes better quality code. [Traditional, Test-Driven, No difference]
produces more customer satisfaction  
[Traditional, Test-Driven, No difference]

results in better coverage of production code.  
[Traditional, Test-Driven, No difference]

results in code that is easier to reuse.  
[Traditional, Test-Driven, No difference]

results in code that is more extendable.  
[Traditional, Test-Driven, No difference]

results in code that is easier to read and understand.  
[Traditional, Test-Driven, No difference]

fosters better confidence in the code.  
[Traditional, Test-Driven, No difference]

helps you to stay on schedule and meet your deadlines.  
[Traditional, Test-Driven, No difference]

gives you a better understanding of the algorithm/logic.  
[Traditional, Test-Driven, No difference]

better facilitates refactoring.  
[Traditional, Test-Driven, No difference]

takes more effort.  
[Traditional, Test-Driven, No difference]

you will be more likely to use in the future.  
[Traditional, Test-Driven, No difference]

you like better for most tasks.  
[Traditional, Test-Driven, No difference]

Produces a code that adheres more to the requirements.  
[Traditional, Test-Driven, No difference]

Given the process you use most frequently, are there any other aspects for which that process does a significantly better job than the alternative? (please specify) [open-ended]

Types of Testing:

Indicate all types of testing that you have used while following ...

20. traditional development.  [Unit, Integration, Acceptance, Other (specify)]
21. test-driven development.  [Unit, Integration, Acceptance, Other (specify)]

Specify what tool(s) you use for ...

22. unit testing.  [open-ended]
23. integration testing.  [open-ended]
24. acceptance testing.  [open-ended]

Post-questionnaire:

This section is optional, but it would be helpful for our research

25. Who is your employer?  [open-ended]
26. What kind of industry are you in?  [open-ended]
27. What is your title?  [open-ended]
28. If you would be interested in communicating with us via e-mail questions or over the telephone in an interview after the survey to obtain quotes and other information for the manuscript, then please provide your email address and/or other contact information.  [open-ended]
Implementing Quality Gates throughout the Enterprise IT Production Process

By

Bart Dahmer and Rick Prendergast

FedEx Software Quality Assurance

Vladimir Ambartsoumian, Jasbir Dhaliwal, Ted E. Lee

Thomas Meservy, and Chen Zhang

Department of Management Information Systems

The University of Memphis

Abstract

The concept of quality gates has been successfully applied as a quality assurance mechanism in several industries. The quality gates approach combines aspects related to project management, decision modeling, and work flow management to increase measurability and promote quality. Software development and testing organizations are now investigating the use of this approach for the purposes of promoting software quality and improving software development processes. This paper summarizes prior literature on quality gates and applies the concept of quality gates to the software development context. It reports a case study of implementing quality gates in enterprise IT production process in the context of a large Fortune 500 company. A conceptual framework is also proposed to represent the various levels and disciplines where quality gates may be implemented. This framework suggests that 1) quality gates can be applied at many different levels throughout the organization such as system, project, and release, 2) the format of quality gates varies by level of implementation, 3) quality gates are useful both as part of an overarching software development methodology as well as for targeted IT projects where quality assurance standards have to be established for procedural success (e.g., transitioning enterprise data centers), and 4) more than the increased measurability promoted by quality gates, greater value may be yielded by a change in developers’ and testers’ mindset of building quality into the software product and development processes.
1. Introduction

Most organizations are dependent on highly complex software packages to achieve their organizational/business objectives. Although the salient question for some of these organizations is which software to buy and how to configure and integrate it with existing systems, many organizations (including many Fortune 500 companies that use technology to sustain their competitive advantage) still must develop custom applications that need to work in concert with numerous other systems that have been developed in-house and/or acquired from software vendors. In such an environment, assuring software quality is a non-trivial task, and thus over the years numerous techniques have been introduced in an attempt to increase software quality. One such technique is the Quality Gates approach, which has been successfully applied in many industries for quality assurance during production processes.

This paper reviews literature on the origin of Quality Gates, the use of Quality Gates in manufacturing processes, and benefits and challenges in using Quality Gates. What follows is a discussion of how to combine the stage-gate project management model with various software development methodologies and how to use Quality Gates in the different phases of the systems development lifecycle. Then, a case study of implementing quality gates in enterprise IT production process is presented and a conceptual framework is proposed. The paper concludes with a discussion of the contributions and implications of our study.

2. Overview of Quality Gates

Numerous quality assurance approaches have been adopted to increase the quality of the ultimate output of a production process. A relatively common technique is the Quality Gates approach. Quality Gates, when used in conjunction with project management, help manage the balance between cost, functionality, and quality.

Literature regarding Quality Gates as a Quality Management concept originated in the European Union. Quality Gates were initially applied to product development processes especially quality control in the automotive industry. Since then, Quality Gates have been more broadly applied to quality assurance and project management (Schneider 2004; Scharer 2006).

The concept of Quality Gates is based on the stage-gate system initially presented in 1986 and later refined by other researchers (e.g., Schmitt 2008). In order to apply this technique, a process must be broken down into several distinct phases. Then, quality checkpoints or gates are placed between phases as an explicit means of checking the quality of artifact that is being produced. Figure 1 (Cooper 1990) illustrates a simple example of a production process that incorporates stage gates.
In general, a Quality Gate marks the formal end to a particular process within a project, a “gate” through which the project proceeds from one phase to another. Each gate results in the certification that all appropriate work required to move products forward to subsequent project activities has been completed and reviewed and products meet specific quality expectations. Based on a set of pre-determined exit criteria established for each phase or milestone being certified, a Quality Gate results in a pass/fail decision for moving forward (Dustin 2009).

Over the years many different conceptualizations of Quality Gates have been proposed. Table 1 captures some of these perspectives. While there is no universally accepted definition across industries as to what a Quality Gate is or how it should be structured, certain regional or industry initiatives have emerged including the Transregional Collaborative Research Centre SFB/TR4 described in academic literature (Schmitt 2008) and the trade press (Charvat 2008).

Table 1 - Quality Gates Definitions

<table>
<thead>
<tr>
<th>Author</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charvat (2003)</td>
<td>• Formal checklists are used throughout the life of a project.</td>
</tr>
<tr>
<td></td>
<td>• Formal sign-off and acceptance occurs at each gate.</td>
</tr>
<tr>
<td></td>
<td>• Assessment of the quality and integrity of the product takes place.</td>
</tr>
<tr>
<td></td>
<td>• Information is assured to be communicated to the correct stakeholders (i.e., deployment hands off to operations, etc.).</td>
</tr>
<tr>
<td>Flohr (2008)</td>
<td>• Significant milestones and decision points with predefined and quality focused criteria</td>
</tr>
<tr>
<td>Schneider (2004)</td>
<td>• “Quality Gate is a checkpoint consisting of a set of predesigned quality criteria that a project must meet in order to proceed from</td>
</tr>
</tbody>
</table>
Quality Gates thus serve as amendments to milestones and deliverables which meet predefined quality benchmarks.”

Historically Quality Gates have been viewed as binary checkpoints throughout a serial process where all of criteria have to pass in order to continue on to the next activity. However, Quality Gates criteria can also include the success of other Quality Gates in such a way that Quality Gates can be interconnected with each other (Schneider 2004). Additionally, Quality Gates need not run only serially, but can (and often) run in parallel. That is, different sub-processes run independently but at some point filter together as products outputted from one phase are used as inputs for the next phase. Additionally, it is important to recognize that Quality Gates are often implemented on a very granular scale that they can be implemented at different levels of process abstraction. For example, a code-level Quality Gate might be established for assuring the quality of a piece of software code that a developer is implementing and at the same time a higher-level project-level Quality Gate may be established for determining whether or not the software project should be released on a particular date.

Numerous benefits of adopting Quality Gates have been identified in the extant literature. These benefits can generally be dichotomously categorized as benefits associated with the product itself or benefits to the production process. For instance, a benefit associated with the product would be the enhanced quality of the product as a result of a series of Quality Gates that were implemented. The production process also sees direct benefits from Quality Gates such as increased communication between teams. Table 2 summarizes some of the benefits of implementing Quality Gates as identified in the literature.
<table>
<thead>
<tr>
<th>Category</th>
<th>Benefit</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Product</strong></td>
<td>Ability to assess quality</td>
<td>Q-gates provide a mechanism for the project team to readily assess the quality of their work products throughout the project life cycle and improve quality at the source. Improve status visibility</td>
<td>Younack (2009)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Measure the current project status more efficiently and effectively</td>
<td>Schneider (2004)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Q-gates ensure that intermediate work products meet the needs of downstream activities through a formal, disciplined process. Problems, their resolution, and opportunities for improvement are identified; and risk assessment or escalation procedures are invoked, if appropriate.</td>
<td>Younack (2009)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The quality of the single process step output is measurable and controllable for every process participant,</td>
<td>Valeri and Rozenfeld (2004)</td>
</tr>
<tr>
<td><strong>Product</strong></td>
<td>Decreased project risk</td>
<td>Minimizing project risk through phase-by-phase checklists</td>
<td>Charvat (2003)</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>Reduced development time</td>
<td>Reducing development cycle time—getting it done right the first time</td>
<td>Charvat (2003)</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>Enhanced project communication</td>
<td>Q-gates enhance project communications, thereby resulting in improved management of the expectations of key stakeholders through their participation in Q-gate certifications. Enabling project managers to continuously communicate the process and build quality directly into the project</td>
<td>Younack (2009)</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>Focus on Quality</td>
<td>Increasing focus by project team on a well-designed product</td>
<td>Charvat (2003)</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>Better planning and control</td>
<td>Better support planning</td>
<td>Schneider (2004)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Quality Gate concept helps to overcome the complexity in the planning and control of production process chains</td>
<td>Valeri and Rozenfeld (2004)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Better control necessary changes or improvements</td>
<td>Schneider (2004)</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>Shared Responsibility</td>
<td>Project stakeholders share responsibility with the project manager for the project’s quality outcome.</td>
<td>Younack (2009)</td>
</tr>
</tbody>
</table>
Despite the salient benefits of using Quality Gates, there are still numerous challenges associated with implementing Quality Gates. For example, in order to be able to implement Quality Gates, production processes must be decomposed into discrete phases. In addition, each Quality Gate is often characterized by its entry and exit criteria. Because in many environments the production process spans teams, departments, or organizations, agreeing on useful entry and exit criteria can be challenging as all parties may not share a common vision of what is requisite for quality at a particular stage or may be biased due to differing perspectives/cultures. More importantly, a major task for organizations that implement Quality Gates is determining where during the production process they should be implemented, how to structure and define Quality Gate criteria and how Quality Gates relate to each other. We suggest that organizations address at least the following questions prior to implementing Quality Gates:

- Where should Quality Gates be implemented?
- What types of Quality Gates should be implemented?
- What should the entry and exit criteria be for each Quality Gate?
- Who is responsible for monitoring each Quality Gate?
- How will the results of the Quality Gate criteria be used in decision making related to the subsequent stages of the production process?

3. Quality Gates in Software Development

A few researchers have outlined combining stage-gate project management models with various software development methodologies. For example, Karlstrom and Runeson (2006) investigate the introduction of an agile software development process within a traditional stage-gate model and identified challenges as well as success factors in the integration process. Wallin et al. (2002) also propose an integration of software development life-cycle models (SDLM) and business decision models such as Cooper’s Stage-Gate Process Model by demonstrating that the latter can be mapped to the major milestones in both the unified process and extreme programming. Finally, Nguyen (1999) proposes a decision model for managing software development projects. The motivation of this study is somewhat similar to the study of Wallin et al (2002) in that both studies try to bridge the gap between SDLMs and business decision-making. The decision model consists of a set of indicators, which link tasks performed during each phase of the software development process (i.e., project definition, requirements, design, and implementation) and the responsible managers. The quality of the tasks is assessed based on their completeness, correctness, consistency, and compliance. The model is then illustrated using a prototypical web-based implementation.

The above studies generally discuss the integration of stage-gate models and software development life-cycle models at a high level with limited details on how Quality Gates are actually implemented (e.g., types of gates, gate criteria).
From a practical standpoint, a number of researchers and practitioners have provided some guidelines for implementing Quality Gates in each phase of the SDLC. For example, Tarrani and Zarate (2001) break the software development process into nine distinct phases: requirements, design, build, product test, quality assurance, roll-out, roll-out post implementation validation, roll-out release, and operation. They propose that a quality gate be implemented after the conclusion of one phase and before the initiation of the next phase. They not only list some possible deliverables and responsible parties at each gate but also identify the criteria for the gates and sample metrics for the criteria. Table 3 provides sample Quality Gate criteria for the “build” phase in systems development lifecycle (SDLC) as proposed by Tarrani and Zarate (2001). Similarly, Farrell-Vinay (2008) identifies several Quality Gates throughout the SDLC including requirements review, architectural design review, code review, system test, and deployment. In addition, Salger et al. (2009) focus on the design phase of information systems development and present detailed Quality Gates for software specification. Chenal and Schwartz (2007) provide some practical guidelines for improving the software development process. One of these guidelines is that Quality Gates and criteria should be defined at the beginning of the unit testing phase, the integration testing phase, the system testing phase, and the user acceptance testing phase.

Table 3 – Sample Quality Gate for the Build Phase of SDLC (Tarrani and Zarate 2001)

<table>
<thead>
<tr>
<th>Responsible</th>
<th>Deliverables</th>
<th>Quality Gate</th>
<th>SQA Metrics (Defects Injected in Phase)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developers</td>
<td>Products and Deliverables: Code(Source code, compiler object code and linked object modules, executable code, objects and classes) DDL (Data Definition Language) scripts Database Creation DML scripts (Data Manipulation Scripts – Queries, etc.) ;( SQL, SQL* Plus, PL/SQL, etc.) Stored procedures, triggers and encoded business rules Server side scripts (csh, pearl, cgi, etc.) Data dictionary Data documentation (keys, constraints, E-R</td>
<td>Quality Gate 3: Conditions to be met/deliverable characteristics: Executable code passes unit and integration tests Schema verification against database design documentation (includes validation of DDL scripts) Data dictionary matches schema Data dictionary, schema, and data documentation validated against one another Stored procedures and triggers return expected results DML and server side scripts validated</td>
<td># unit test failures # integration test failures # build discrepancies # instances of script rework # discrepancies found during scheme verification # discrepancies found in schema documentation (traceability, factual errors, instances of missing information) Documentation follows FSS standards (Yes/No) # source code reconciliation failures between code base and library control logs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technical Writers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Manager (Software and documentation control)</td>
</tr>
</tbody>
</table>
Furthermore, Quality Gates have been adopted by organizations such as Fidelity International, United States Department Veteran Affairs, and Microsoft Visual Studio Team for quality assurance purposes. For example, Fidelity International reported that six Quality Gates (pre-project, business study, functional model/design-build iterative, system integration testing and user acceptance testing, implementation, and post-project) were used with its agile software development method. What is interesting about these Quality Gates is that they consist of multiple criteria with a pre-determined weighting scheme and that the quality score at each gate is calculated based on the weighted sum of individual criterion scores.\(^1\) Although this weighted calculation approach for quality scores may require more effort in the initial identification of all possible quality criteria, it allows project managers and other responsible personnel to adjust the weightings of certain criteria based on the specific nature of each project.

As another example of applying Quality Gates to software development, a team working on Microsoft Visual Studio has also adopted Quality Gates to improve software quality (Saad 2008). Before a feature

\(^1\) Available at http://www.cmminews.com/2006/Pres2006/25th/Practitioner/Quality%20Assurance%20in%20an%20Agile%20Delivery%20Method.pdf

<table>
<thead>
<tr>
<th>Responsible</th>
<th>Deliverables</th>
<th>Quality Gate</th>
<th>SQA Metrics (Defects Injected in Phase)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>diagrams, triggers and stored procedures)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Technical documentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operational documentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>User documentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Findings from peer, preliminary and critical reviews of all documentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Library control logs (manual or automated)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Build analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Release notes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Can also include:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Object request broker configuration</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Access control list development</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>All documentation changes are traceable to source of/reason for change</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>All documentation matches system as built</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Source code changes (including code, DDL/DML and server side scripts) can be traced to source of/reason for change</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Code base is reconciled with library control logs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A build analysis is provided with the product</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Release notes are provided with the product</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td># errors in build analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td># errors in release notes (missing, incomplete or inaccurate information)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
can be checked into and integrated into the main product, it must go through a set of Quality Gates to ensure the quality and stability of the final product. These Quality Gates may include static code analysis, code coverage, localization testing, security implication analysis, API reviews, and so on.

Overall, the existing literature has identified that Quality Gates can be applied throughout the software development life cycle or within the development and testing domains. Yet, most studies so far have primarily focused on quality gates within the scope of a project. As such, we propose in this study that quality gates can be adopted as a mechanism to achieve better quality assurance throughout the enterprise IT production process. In the following section, we present a case study in the context of a large organization in the process of implementing quality gates.

4. Case Study of Implementing Quality Gates to Software Development Life Cycle

FedEx is a Fortune 500 company with a diverse and complex software application portfolio that consists of customer facing applications, middleware applications, and backend systems that need to operate in concert in order to provide the best customer experience. Although the organization utilizes existing technologies and some commercial software products, a significant portion of the applications are developed in-house following a standardized software development process. Due to the complex nature of this portfolio (e.g., number, type, and interdependency of applications) and the composition of the software development teams (e.g., size, skill set, and process maturity), software quality assurance is especially critical to corporate revenues and customer satisfaction. As a part of the software quality assurance initiative, alternative approaches were considered to resolve the inconsistent levels of the quality of the code coming from disparate development groups, which are housed in multiple operational divisions.

In July of 2008, the enterprise testing organization within FedEx was renamed to Software Quality Assurance, with an expanded charter to provide processes and measurements to ensure software quality. Soon after, several quality improvement efforts (e.g., QA Prescripts, enterprise-wide IT renewal, improvements to the Global Development Process) were undertaken to develop key quality standards throughout the software development lifecycle, to improve standardization of development and testing deliverables, and to improve metrics around test coverage of requirements.

Subsequently, a special task team was set up to investigate the potential of Quality Gates as an approach to promoting software quality. The team consisted of an external consultant, vendor team and staff members from various areas in FedEx including Software Quality Assurance, Information Security, Revenue Systems, eCommerce, and Enterprise Architecture.
In collaboration with this team from FedEx and based on an extensive review of scholarly and practical studies, a framework for implementing quality gates for enterprise IT production process was developed.

Like most Quality Gates research in software development, the framework captures the idea that quality gates can be implemented as gates between software development phases (e.g., scope, build, test). The framework characterizes these lowest-level quality gates as Heartbeat-level Quality Gates (Rautiainen 2004). In general these gates combined with objective and metric-based criteria most often result in a pass or fail outcome. Typically these heartbeat-level metrics are gathered by automated tools. Some tools analyze the artifact itself only (e.g., syntactical analysis) whereas other tools examine the artifact in combination with other supplemental code that exercises the artifact (e.g., passing or failing or unit tests, code coverage).

From an intra-system perspective, these heartbeat-level gates are implemented within the production process for different disciplines (e.g., physical infrastructure and services, back-end/data systems, middle ware, end-user applications) that are involved with producing a software system. For example, physical infrastructure and services might go through a scope, build, and test production process to ready the physical environment for downstream disciplines. Between each of these phases, specific low-level quality gates may be implemented to ascertain the quality of the software components in meeting its pre-determined objectives. Subsequently, other disciplines (e.g., back-end/data systems,
middle ware, end-user applications) will follow a similar pattern of implementing discipline-specific quality gates between the same phases.

At the end of each discipline’s production process for the particular component, another quality gate is implemented to ensure that the quality is adequate for downstream disciplines to utilize. Although the production process for each discipline typically requires the completion of upstream discipline activities to be completed before finalizing its product, often some portion of the process may occur simultaneously. When the application production process is complete, which necessitates the artifact produced by other disciplines to have passed through intra-discipline and inter-discipline gates, the products have to pass through another quality gate before beginning systems testing.

Unlike most existing work related to Quality Gates, the framework also suggests that strategic-level Quality Gates should be implemented between layers of the enterprise IT production process. Strategic-level Quality Gates improve the ability to assess the current state of IT systems’ quality through increased transparency and allow management to make more accurate risk-based decisions. While many of the decisions made at higher-levels of the software process (e.g., whether to include specific system changes in a particular project, whether or not a particular project should be released) still incorporate subjective evaluations of information streams, strategic-level Quality Gates attempt to reduce the subjectivity by introducing further structure/criteria to the decision making process. Furthermore, strategic-level Quality Gates often utilize abstract heart-beat level Quality Gate outcomes for use in the decision process thus increasing the objectivity of the information used in the decision making process. Different levels of management are focused on different boundaries and hence Quality Gates should be implemented between various layers of the overall enterprise IT production process including system-project, project-release, release and IT, and the alignment between IT and business.

6. Initial Implementation of the Framework

The focus of the initial Quality Gates effort at FedEx was placed on intra-system heartbeat level quality gates. Automated static code analysis and code coverage analysis have been implemented during the build and test phases to improve code quality upon entry into the system testing phase.

The objective of performing static code analysis using an automated tool is to detect problematic programming practices and potential errors without actual execution during the development stage. Static code analysis has the potential to: 1) decrease production defects that are undetectable during black box unit testing and system testing, 2) increase productivity and code consistency due to adoption of good programming practices among large development teams, and 3) comply with internal or regulatory software quality initiatives. An open source software product for the static code analysis was adopted by the quality gates team to perform static code analysis using eleven customized rules sets (e.g., basic rule set, design rule set, optimization rule set) that are most applicable to the organization. This code analysis tool is capable of identifying potential problems in code such as suboptimal code, duplicated code, overcomplicated expressions, etc. Based on reports generated by code scans of this code analysis tool, developers now update their source code to ensure that all violations have been corrected.
The objective of performing code coverage analysis is to determine a quantitative measure of the percentage of program statements, branches, or paths that are executed by test cases in unit testing, integration testing, and system testing. A code coverage analyzer software was adopted to automate this process and the code coverage quality gate criterion is 90% unit test code coverage using structured testing/basis path testing.

These two quality gates are implemented between the build and test phases of the application discipline as an initial step in identifying and specifying quality gates throughout the enterprise IT production process. In addition, a process is under way to develop plans to implement heartbeat level quality gates for the other disciplines within the intra-system perspective. The next stage of implementing quality gates may focus on requirements documents (semantic scanning and requirements coverage), interfaces, or segmented testing. Additionally, implementation of strategic level quality gates throughout the organization will be investigated in more detail.

7. Conclusion and Future Research

Quality Gates have been used in variety of industries in order to increase quality of the artifacts that are the result of a production process. Quality Gates, which are normally implemented as entry and exit criteria between phases of a serial process, have been explored for use in software development. In this article we presented an overview of how Quality Gates have previously been applied and then proposed a framework of applying quality gates for enterprise IT production process.

Theoretically, building upon previous studies that have traditionally examined quality gates during the software development life cycle, this study proposes a holistic framework of implementing quality gates throughout the enterprise IT production process. In addition, we distinguish among quality gates at various levels such as intra-system level, project level, system level, release level, etc. Practically, the proposed framework provides guidance to organizations interested in implementing quality gates for software quality assurance.

Future research may focus on the quality gates implementation process at various levels of the software production process. It would also be interesting to explore how quality metrics obtained from heartbeat level quality gates can be used as criteria in strategic level quality gates. Another direction for our future research is to empirically investigate the organizational impact of quality gates implementation.

References


The Impacts of Service Quality Indicators on Customer Churn
A Data Mining Approach

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

Michael Cuenco, Crystal Shi
FedEx Corporation
Memphis, TN
{michael.cuenco, xcshi}@fedex.com

Abstract—Customer churn is a major concern for most businesses in a competitive space. Understanding the factors that contribute to churn can help firms design better customer retention strategies. Among key churn factors are customer characteristics and service quality indicators. Examples of customer characteristics are nature of the business, location, and loyalty; while examples of service quality indicators include missing packages, loss claims, and missed pickup windows. There are, however, a very large number (hundreds) of specific customer and service quality indicators that are known or hypothesized to affect churn. With this large number of indicators identifying the significant ones that matter for churn and quantifying their impact are crucial. In this research, we study the impacts of customer factors and service quality indicators on churn, focusing specifically on knowledge discovery approaches from data mining for this task. Significant contributions will be made to both the theory and practice of identifying and remediating customer churn.

Keywords—software systems testing, service quality indicators, customer churn, data mining

XIV. INTRODUCTION

One of the greatest challenges in testing and evaluating the quality of software-intensive systems is the identification and repair of subtle defects that escape into business operations. Such defects may impact systems use in myriad ways leading to poor service quality in critical business processes. The goal of our on-going research project is to design and implement a continuous software system quality assurance and testing approach that seeks to identify key software defects that impact the service quality of a business organization’s business processes.

Our previous research, as presented during the past two STEP workshops [3, 4], focused on representing the mappings between software systems and business processes. We investigated various data mining approaches to discover fault patterns in business process logs. We then proposed 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 in the software systems.

In this paper, we look more closely at service quality indicators gathered during key business processes and their impacts on customer churn. As discussed in Section II, customer churn is considered to be a principal factor in the success or failure of a business enterprise. However, surprisingly, empirical evidence demonstrating a causal relationship between specific service quality indicators and customer churn has been sparse and weak [2]. To study this question, we propose a data mining approach on leading service quality indicators to discover possible causal relationships with churn. Sections III and IV describe the two phases of the data mining approach to be used in the research. The paper concludes in Section V with our on-going research activities and future directions.

XV. CUSTOMER CHURN

Customer churn is a major concern for most businesses in a competitive space. Understanding the factors that contribute to churn can help firms design better customer retention strategies [5, 6]. Among the potential churn factors are customer characteristics and service quality indicators [1]. Examples of customer characteristics are nature of
the business, location and loyalty, while examples of service quality indicators include missing packages, loss claims and missed pickup windows. There is, however, a very large number (several hundred) of specific customer and service quality indicators that are known or hypothesized to affect churn. Among these large number of indicators identifying the significant ones that matter for churn, and quantifying their impact, is crucial.

In this research we propose to study the impact of customer and service quality indicators on churn, focusing specifically on pattern discovery approaches from data mining for this task. Current approaches for churn consider the large number of individual indicators, but often do so on an independent basis. For instance, each of the several hundred potential indicators can be individually examined for their influence in churn using linear models. While this will provide first order effects and important insights, a limitation of such approaches is that they do not consider interaction effects in the churn process. For example, missed pickup windows for small business customers who are in business for less than a year may contribute to churn, although individually the single indicator (missed pickup windows) may have little or no effect on churn.

Identifying such interactions can be done in a few different ways. First, domain experts may list a set of such interactions to examine. The downside is that this process may miss some important interaction effects. Second, interactions may be exhaustively enumerated and studied. Combinatoric intractability however makes this impractical for large problems (e.g. even with 100 indicators, there are approximately a million interaction effects at only three levels of interaction). Third, significant interactions may be identified in a more intelligent manner to avoid the combinatorial concerns from enumeration. Pattern discovery approaches are particularly effective for this and will be the focus of this research.

In this research we are working with one of the largest logistics providers in the world to study the impact of service quality factors on churn. The setting is non-contractual as well. Customers always have multiple options for shipping packages and can easily switch shipping volumes in reaction to events. This study will therefore be one of the first to empirically examine service quality and churn in a non-contractual setting.

We adopt an exploratory pattern discovery approach for this problem given the large number (several hundred) of customer and service quality indicators that are known to or hypothesized to affect churn, interactions among which are also known to be important. For instance, it may be the case that most service quality factors do not impact churn but package delays coupled with location (e.g. availability of alternatives) may be one of few important combinations. Rule discovery approaches are ideal for such problems since they do not seek any global model that will fit an entire data set but instead output any and all such (local) interactions of interest.

We adopt a two-phased approach. In phase one we assume the existence of a churn label provided by the group that leads data mining activities for the logistics company and learn churn segments. In phase two we learn churn segments by inductively determining unusually low volume segments rather than using the provided churn flag. The reason for doing so is to study the more general non-contractual setting where firms only observe volume and have to inductively determine whether there is churn. In such settings, methods to clearly label churn do not readily exist since much of the churn may only be observed as a drop in volume. However not all such drops may indicate churn. For instance, due to macroeconomic factors an entire industry segment may just be shipping less. Hence, data-driven methods that can intelligently flag such drops in volume are necessary.

XVI. Phase 1 - Learning Churn Segments

We assume a database table of customers where each row represents one customer with the following attributes: customer attributes, service quality indicators for this customer over a pre-determined time horizon and a single attribute indicating whether this customer has churned (or not). The churn attribute could also be a continuous measure capturing the degree to which a customer has churned, as determined by revenue/volume data trends for instance.

From such a database the approach described below will extract rules of the form:

\[
\text{Location} = \text{“San Diego”}, \; \text{Business Size} = \text{“Medium”}, \; \text{Missed Pickup Windows} = \text{“Low”} \Rightarrow \text{Churn} = 6\% \; (\text{Expected churn 1 to 2%})
\]

The methodology is based on the approach in [7]. In prior research this was applied to Web browsing data to learn segments where individual firms had higher or lower market share. Below are some real examples of segments learned from prior research:

(1) Household Size = 3 ∧ 35K < Income < 50K ∧ ISP = Dialup ⇒ marketshare_{Expedia} = 27.76%, support = 2.1%
In addition to customer specific attributes, two attributes in this database that were used to compute the market share are (1) the specific ecommerce site at which a purchase was made, and (2) the purchase amount. The market share for a specific site, e.g. Expedia.com, is computed as the dollar value of flight ticket purchases (satisfying the Left Hand Side (LHS) of the rule) made at Expedia.com, divided by the total dollar value of all flight ticket purchases satisfying the LHS.

More generally, “market share” is an example of one statistic that can be computed based on the segment satisfying the antecedent (LHS) of the rule. Besides market share, various other quantitative statistics on the set of transactions satisfying the LHS of a rule can be computed, including mean and variance of an attribute. We termed rules such as these as statistical quantitative rules (SQ rules). Below is a formal definition from [7]:

**Definition (SQ rule).** 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, an SQ rule is a rule of the form:

\[
X \Rightarrow f(D_X) = \text{statistic}, \text{ support } = \text{sup}
\]

where \( X \) is an itemset (conjunction of conditions) involving attributes in \( A \) only, \( D_X \) is the subset of \( D \) satisfying \( X \), the function \( f \) computes some statistic from the values of the \( B \) attributes in the subset \( D_X \), and support is the number of transactions in \( D \) satisfying \( X \). \( \square \)

Note that the statistic on the RHS of the rule can be computed using the values of multiple attributes, as shown in the market share examples. With respect to learning SQ rules from data, we formulated the problem as learning significant SQ rules that have adequate support. We defined an SQ rule to be significant if the specific statistic computed for the rule lies outside a certain confidence interval. This confidence interval represents a range in which the statistic can be expected by chance alone. This is an important range to identify if the rules discovered are to be interpreted as suggesting fundamental relationships between the LHS and the market share. For example, by chance alone if it is highly likely that the market share of Expedia is between 25% and 30% for any subset of data, then it is not at all clear that the rule relating income and Expedia’s market share (rule 1 in the example) is identifying a fundamental relationship between income and the market share.

**Definition (Significant SQ rule).** For a given significance level \( \alpha \in (0, 1) \), let \((\text{stat}_L, \text{stat}_H)\) be the \((1 - \alpha)\) confidence interval for a desired statistic, where this confidence interval represents the range in which the statistic can be expected by chance alone. An SQ rule \( X \Rightarrow f(D_X) = \text{statistic}, \text{ support } = \text{sup} \) is significant if statistic lies outside the range \((\text{stat}_L, \text{stat}_H)\). \( \square \)

The confidence intervals are learned using a non-parametric methodology where we use randomization to compute expected ranges for the function of interest. For example, if the churn rule example presented at the beginning of this section (for small businesses in San Diego) applied to 100 firms, we randomly sample 100 firms repeatedly to build a distribution of expected churn values. The actual churn rate in the segment is then compared to this distribution of values to determine whether the rule discovered is considered significant.

**XVII. Phase 2 - Learning Sequential Patterns**

We assume a database table of event sequences where each row represents a sequence of events ordered in time for one customer and a single attribute indicating whether this customer has churned (or not). From such a database an approach using sequential patterns can extract rules of the
In this case, this sequential pattern is assumed to be present in the rows corresponding to “churn” and is absent (or present with a much smaller frequency) in the “non-churn” rows. Such a pattern is termed a “distinguishing sequential pattern” in our prior work [4].

In this prior work we applied our method on “process logs” that were grouped into “good” and “bad” sets $D_G$ and $D_B$ respectively (these sets could correspond to sequences resulting in churn and no-churn for instance). Each process log instance is a sequence, such as ${S_1, P_1}$, ${S_2, S_3}$, ${S_4, P_2}$, ${P_3}$. An example of a sequential pattern is: ${S_1, P_1} \rightarrow {S_3} \rightarrow {P_3}$, support = 15%.

In this example, the pattern states that 15% of all sequences have ${S_1, P_1}$ in them followed by ${S_3}$ and then subsequently ${P_3}$. This pattern, for instance, is contained in the process log instance ${S_1, P_1}$, ${S_2, S_3}$, ${S_4, P_2}$, ${P_3}$.

**Inputs:**
Process log databases $D_G$ and $D_B$

Pattern discovery algorithm, $R$, that learns patterns from the process log instances (which are sequences)

**Output:**
A ranked list of distinguishing patterns

1. Generate a set of patterns $P = \{P_1, P_2, ..., P_M\}$ by applying $R$ to $D_G$.
2. Generate a set of patterns $Q = \{Q_1, Q_2, ..., Q_N\}$ by applying $R$ to $D_B$.
3. For each pattern $p \in P \cup Q$, compute strength_difference($p$, $P$, $Q$)
4. Rank all patterns based on strength difference values

If the sequential patterns that are learned from each group (good vs. bad) are similar then we can conclude that $D_B$ and $D_G$ do not have any systematic differences (with regard to the sequential patterns in them). If some sequential patterns are “different” these are output by our method. Such difference is defined in two ways:

(i) Patterns unique to each group can be identified. Specifically, $(P - Q)$ represents the set of patterns in $D_G$ but not in $D_B$. Similarly $(Q - P)$ will be useful since these are patterns that exist in $D_B$ but not in $D_G$.

(ii) There may be the same patterns in both groups, but their strengths may be different. The data mining literature has addressed the identification of such patterns. In particular, contrast sets are a way to find such differences between groups. A contrast set is a combination of attributes and their values which differ in support amongst separate groups. In a similar vein, prior research used a normalized support difference metric to compute the distance between the strengths of a pattern in two groups. Specifically if $s_1$ and $s_2$ are the supports of a pattern in two groups, this approach uses $|s_1 - s_2| / 0.5*(s_1 + s_2)$ as the difference.

In order to investigate this approach on the sample data, highly granular sequence data (e.g. at the daily level) may be needed. Incorporating customer attributes into these sequential patterns would be desirable as well – an aspect not directly addressed by our current approach. Given these two we propose to investigate this approach in Phase 2 of our collaboration after considering the output from Phase 1 (learning churn segments).

**XVIII. Research Status and Future Directions**

The data mining approach detailed above has been designed and prototyped in a business intelligence tool and tested on sample data. We are in the final stages of obtaining a large data set from our industrial partner that has highly granular information on service quality indicators at a customer level. This data is coupled with time series data on transaction volume and churn. Data mining on this target
data set will begin soon and we hope to present some preliminary results from the analysis at the STEP workshop in May.

In addition to learning churn segments and sequential patterns, two other open and interesting problems for investigation in the context of this project include:

1. Time series clustering. Specifically, revenue and volume data for each customer represents two time series data. Given a large number of such time series data, approaches for clustering these time series data could be of value. In particular such clustering might better identify the segment that each customer belongs to and may be used in determine whether churn is real or due to a broader decline in the industry. Similarly such approaches may be used to estimate share of wallet measures by identifying a hypothetical 100% loyal time series within each segment.

2. Determining optimal time windows. In order to determine the impacts of specific indicators on churn, the data needs to be prepared at some time window, usually predetermined. A broader question is what time window is optimal in order to build accurate churn models. While these may be straightforward to do experimentally, intelligent approaches for this problem can be explored as well in future work.

ACKNOWLEDGMENT

We gratefully acknowledge the collaboration of our industrial partner in this research. We are ably assisted in this project by two graduate students at the University of South Florida – Naresh Kalla and Jing Cheng.

REFERENCES


Interface Testing: Understanding and Addressing Software Interface Complexity in Large Organizations

Thomas Meservy, Jasbir Dhaliwal, and Murad Akmanligil

Software Testing Excellence Program (STEP)

A. Abstract

Many of today’s software systems contain complex interdependencies between modules. In many large systems, the interfaces between modules/systems are the source of a significant percentage of system faults that are discovered during systems and integration testing. We assert that changes to interfaces should be tested much earlier in the software development lifecycle. However, due to the complexity of the environment and numerous other factors, many organizations continue to struggle how to categorize and address interface related issues. This paper presents a framework for categorizing interface faults and characterizes salient interface faults derived from a case study at a large organization that has a complex application portfolio. Additionally, insights are shared from onsite interviews on how companies might address interface-related issues. Finally, an interface complexity index is presented that can be used to help companies focus resources on the most risky interface changes.

B. Introduction

Many of today’s software systems contain complex interdependencies between modules. When these modules are appropriately woven together they can provide useful services while also promoting code reuse. In large organizations modules are often developed by separate teams or even departments. The emergence of concepts such as service-oriented architecture and enterprise service bus has added another layer of complexity. Central to the effective integration of these disparate modules is a shared understanding of objectives of each module, how to invoke the module, and an understanding of the service expectations of each module. Essentially, effective integration relies on well-defined and well-understood interfaces.

Testing of complex software systems, especially as the testing focuses on testing the integration of modules, is a challenging endeavor. Approaches to integration testing are generally grouped into four categories: top-down, bottom-up, sandwich (combination of top-down and bottom-up), and big-bang. The basic approach utilized in the first three categories is to isolate changes around the interface by creating stubs or mock objects that return pre-specified data for a specific interface. The top-down approach starts at the highest level and works its way down to the smallest unit. The bottom-up starts by first testing at the lowest levels and combining modules into subsequently larger or longer sequences. The sandwich approach is a combination of the top-down and bottom-up approach, systematically working from both ends towards the middle. The big bang approach to integration testing essentially tests most or all of the modules together at once.
In large systems where modules are developed by disparate teams or in systems where multiple services are integrated, interfaces become a fragile link in development, testing, and maintenance of software systems. Confidence that the module or service will remain stable (i.e., interface will not change) and consistently deliver quality results becomes a salient issue in providing high quality software. Additionally, organizational factors also influence the definition of interfaces.

While there has been some interest in the area of defining and testing interfaces, more research is needed. In particular, more understanding is needed around practices of defining interfaces, integration testing of software that spans team or departmental boundaries, and understanding how to effectively test modules that are hosted externally as services.

This project presents pragmatic, prescriptive advice based on a case study conducted at a large organization where complex interdependencies exist between software modules developed and maintained by different teams/departments.

C. Interface Fragility

A variety of factors can increase or diminish the fragility of interfaces. Salient factors related to the fragility of interfaces can be grouped in to three main categories: product factors, technological factors, and organizational factors.

Product factors are related to the software product itself. For instance, a software product that has constantly changing requirements may require changes to the interface between modules. If the same version of the requirements are not communicated to all parties or used in a semantically consistent ways, then there is increased likelihood that the interface will be misused or mis-specified.

Technological factors relate to the technology that is used to implement the software modules and how those modules invoke each other. For example, the stability of the technology and its ability to interface and interact with other technologies impacts interface fragility. Organizational factors, in general, take into account the human aspect of designing, developing, deploying, and maintaining software systems. For example, in large organizations different software modules are designed, developed, deployed, or maintained by different groups with diverse skill sets, software production and testing processes, vocabulary, cultures, etc. Table 1 captures sample factors that impact the fragility of software interfaces.
Table 1- Salient Interface Fragility Factors

<table>
<thead>
<tr>
<th>Category</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Factors</td>
<td>Stability of interfaces</td>
</tr>
<tr>
<td></td>
<td>Changing Requirements</td>
</tr>
<tr>
<td>Technological Factors</td>
<td>Maturity of technology</td>
</tr>
<tr>
<td></td>
<td>Evolving technologies</td>
</tr>
<tr>
<td></td>
<td>Mixture of technologies</td>
</tr>
<tr>
<td>Organizational Factors</td>
<td>Team size and composition</td>
</tr>
<tr>
<td></td>
<td>Maturity of inter-group/inter-departmental development &amp; testing process</td>
</tr>
<tr>
<td></td>
<td>Service quality</td>
</tr>
</tbody>
</table>

D. Interface Faults

Fragile interfaces manifest themselves in the form of interface faults. Past empirical research in large software systems have suggested that a significant portion of all software faults are interface faults, that is, they stem from issues relating to the interface between software modules. Basili and Perricone (1984) found the 39% of the faults in their study were interface faults. Similarly, Perry and Evangelist (1985) found that an astounding 66% of the errors analyzed in their study arose from interface problems. However, other studies have reported a much smaller percentages of errors stemming from interface related issues (Weiss 1979; Bowen 1980; Thayer et al. 1978; Lutz 1993; Marick and Motorola 1979).

Interestingly, there is only a paucity of studies related to understanding and characterizing interface faults. One notable study conducted by Perry and Evangelist (1985) analyzed a selection of faults reported by testers of a large C application (~350,000 lines of code) in order to create a framework for categorizing interface related faults. The final framework contained 15 categories. Some of the most significant categories included inadequate error processing (15.1%), construction of the interface (12.9%), inadequate functionality (12.9%), inadequate post-processing (10.5%), and changes to the underlying data structures (9.4%).

It is important also to recognize that most studies related to interface faults are several decades old and that the current paradigm of software development and current technologies likely do not reflect the exact same subset of issues. However, many of the categories are still relevant. Additionally, many large organizations still utilize legacy software applications that use the technologies discussed in these articles.

In an effort to understand and categorize current software interface faults, we adapted\updated the framework presented in (Perry and Evangelist, 1985) to include categories that may be relevant to modern programming paradigms and also categories that are relevant to large software systems. Additionally, we further clustered the interface fault categories into four main groups: definition/structure, functionality, execution, communication.
<table>
<thead>
<tr>
<th>Category</th>
<th>Factor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition / Structure</td>
<td>Construction</td>
<td>Issues relating to how the interface is constructed</td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>Data types / size / number of the interface parameters are inappropriate or insufficient</td>
</tr>
<tr>
<td>Functionality</td>
<td>Inadequate functionality</td>
<td>These are faults caused by the fact that some part of the system assumed, perhaps implicitly, a certain level of functionality that was not provided by another part of the system. The data returned from a specific interface did not return all of the data that was expected.</td>
</tr>
<tr>
<td></td>
<td>Changes in functionality</td>
<td>A change in the functional capability of some unit was made in response to a changing need.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The data returned from a specific interface did not return what was expected due to changes</td>
</tr>
<tr>
<td></td>
<td>New functionality added</td>
<td>Issues that were a result of completely new functional capability was recognized and requested as a system modification. Additional functionality was added to an interface and was not appropriately consumed by the system that called the interface</td>
</tr>
<tr>
<td>Execution</td>
<td>Inadequate error processing</td>
<td>Interface errors were either not detected or not handled properly.</td>
</tr>
<tr>
<td></td>
<td>Misuse of interfaces</td>
<td>These are faults arising from a misunderstanding of the required interface among separate units. In other words the interface was called in a way that it was not intended to be used</td>
</tr>
<tr>
<td></td>
<td>Messaging Issues</td>
<td>Interface issues that were caused by the messaging systems (e.g., issues with Java Messaging Service). Could be configuration issues;</td>
</tr>
<tr>
<td>Communication</td>
<td>Human Communication</td>
<td>Someone failed to communicate modifications to one system/software unit to those responsible for other systems/units that depend on the first.</td>
</tr>
</tbody>
</table>
To further understand the types of interface faults that exist within large organizations and to investigate how interface issues might be addressed, we conducted a case study at a large international logistics company where a subdivision recently has been focused on minimizing interface faults and reducing the overall complexity of interfaces in their portfolio.

E. Case Context
FedEx is a Fortune 500 company that has a very diverse and complex application portfolio. This portfolio consists of customer-facing applications, numerous middleware systems, and backend systems that need to operate in concert in order to provide the expected level of service. The entire fleet of applications is mission critical and any down-time has a direct impact on revenue and customer satisfaction.

Like most large organizations, FedEx’s software portfolio has evolved over several decades and has increased in diversity as the company itself has grown through numerous acquisitions. The underlying software systems rely on numerous technologies, some of which have been around for decades and others which are relatively nascent. Furthermore, FedEx not only has numerous departments within its organization, but also has several different operating companies that have traditionally used disparate systems or historically have had different processes and procedures for designing, developing, and maintaining software systems. As FedEx is a global company, not only do their systems have to handle numerous complex issues dealing with different geographic locations, nations, cultures, currencies, etc., but they also have a somewhat distributed workforce that is responsible for systems development and maintenance. It is no surprise that with such a complex environment that FedEx is focused on software quality throughout the software development lifecycle.

The purpose of this case study was to understand what types of interface faults are common within FedEx (and presumably other large organizations) and investigate ways to minimize the impact of interface faults.

F. Data Collection and Analysis
The authors participated in face-to-face meetings and audio-only conference calls with numerous employees that had experience with interface-related issues. In all there were approximately 15 informants including testers, developers, team leads, and managers. Informants had varying perspectives due to their level of experience, skill set, role within the organization and history within their particular department/organizational subunit.

Initially, we attempted to assess the type of interface faults existed within the environment by reviewing defects that were extracted from a previous corporate load. Researchers filtered through the 1200 defects and attempted to identify interface faults. A subset of the defects were marked as probable and sent in return to FedEx. Unfortunately, the level of detail that was captured in the issue tracking system was insufficient to definitively assess whether or not the error was an interface fault. It also was insufficient to categorize the type of interface faults.

As an alternate, the researchers meet with employees who often come across interface related faults. The researchers discussed the framework with the participants and asked for salient examples of the
Table 3 captures some salient examples of interface related issues gathered from interviews

Table 3 – Root Causes of Interface Faults

<table>
<thead>
<tr>
<th>Root Cause of Interface Faults</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment/Configurations</td>
<td>Interview participants repeatedly describe how many interface related issues came down to an environment or configuration issue. Issues included jobs not being run, inadequate error handling due to upgrading infrastructure which introduced new error messages that weren’t produced previously; production and testing mismatches;</td>
</tr>
<tr>
<td>Improper Data Mapping</td>
<td>Interviewees reported interface functionality changes or errors due to improper data mapping. This occurred when one field from a backend system was mistakenly mapped to another field. Improper mapping – weight field should have been mapped to another field; moving of data from one application to another; in some instances this was not found until late in the test cycle (i.e., Level 3 testing)</td>
</tr>
<tr>
<td>Messaging System Issues</td>
<td>Interviewees reported issues related to the messaging technology. Issues include messages not arriving, not arriving in the correct order, message queues filling up because the bridge had gone down, etc.;</td>
</tr>
<tr>
<td>Human Communication</td>
<td>Human communication, or miscommunication, was frequently cited as a root cause of interface faults. That is, requirements were not understood or the interface was not used appropriately. This is particularly true when dealing with different operational units or across organizational boundaries as the vocabulary/terminology varied.</td>
</tr>
</tbody>
</table>

G. Approaches for Dealing with Interface Complexity

Interface issues are not a new phenomenon. From our interviews we discovered that over the years there have been attempts at defining and controlling interfaces changes at FedEx, but many of these have fallen to the wayside for a variety of reasons. However, we also discovered that there are current initiatives to both quickly address existing interface issues and also to institute process changes in an attempt to minimize future interface faults. Here we will briefly describe three of these new or ongoing initiatives.

SWAT Team

FedEx has a cross-department, cross-manager project team dedicated to triaging incoming software defects in an attempt to minimize defect “ping pong” or other shuffling of the defect between different
teams. The goal of this team is to minimize the analysis time frame for a defect, that is, the time it takes from the creation of the defect to having it assigned to the person/team that will make the ultimate fix. Additionally the team shields development and others from looking at defects that aren’t pertinent to them. This team handles 300 defects a day and only looks at “high” and “critical” defects. Interface issues, by definition, involve more than one module. Having a team that can authoritatively address interface issues across boundaries leads to quicker resolution.

**Cross Project Interface Reviews**

The SQA team has initiated cross project interface reviews in an attempt to bring awareness of pending interface changes associated with a particular corporate load. While the information about changes to interfaces is captured in requirements specifications, not all parties that consume the interface (directly or indirectly) are fully aware of the changes. Various approaches are being explored including:

- Timelines for interface reviews
- Executive summaries for multiple projects
  - Considerations of how to segregate projects including what resources to include and how often
- How and where to include interface reviews in the global development process

**Data In Motion**

Data in Motion (DiM) is a relatively recent initiative focused on reducing the number of software interfaces and standardizing on technologies/approaches for interfaces between modules/systems. Currently there are estimated to be over 10,000 interfaces utilizing 19 different technologies. The initiative will attempt to reduce the number of interfaces and decrease the number of technologies used throughout the organization. Additionally, long software chains (transactions that go through numerous different software modules) will be reduced by removing non-value-added layers.

In addition, the initiative is creating a common interface library which acts as a repository of information about all production interfaces. This approach is expected to be beneficial to testing, development, and architecture teams. Specifically it will allow testing to identify what systems are impacted earlier in the lifecycle and allows focused allocation of resources for end-to-end testing on just those systems that have changes. While this initiative is still in its infancy, it appears to be gaining momentum and would greatly benefit the organizational understanding of interfaces and understanding the impact of the changes.

**Developing an Interface Complexity Index**

If resources were unlimited, perhaps organizations would review every change to every interface and involve any party who might be affected. Obviously, however, resources need to be allocated to areas that will provide the greatest return on investment. During our research work, the idea of developing a metric for assessing the complexity of changes to an interface emerged.
Software metrics as a subject area is over 40 years old but still has barely penetrated into mainstream software engineering. A key reason for this is that software metrics have rarely provided valuable information to support quantitative, managerial decision making during the software development lifecycle (Fenton and Neil, 2000). Good support for decision-making implies support for risk assessment and reduction (Fenton and Neil, 2000). Previous software metrics can be dichotomized into internal and external attributes for software products (e.g., specifications, design, code, test data), software processes (e.g., development, testing), or resources (e.g., personnel, teams, organizations). Internal attributes include metrics such as size, modularity, correctness, coverage, time, effort, price, etc. External attributes include metrics such as maintainability, quality, stability, cost effectiveness. For more details see (Fenton and Neil, 2000).

In order to characterize the risk associated with changes to interfaces, we developed an interface complexity index that could be used to characterize the risk associated with interface changes. The factors of this index were derived based on component-based software complexity metrics in the academic literature (Gill and Grover, 2004) as well expertise of experienced software developers and testers. Table 4 displays the initial factors derived for this interface complexity index.

### Table 4 – Interface Complexity Index Factors

<table>
<thead>
<tr>
<th>Category</th>
<th>Factor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interface Changes (IC)</strong></td>
<td>IC-NewOrExisting</td>
<td>Whether or not this is a new interface or a change to an existing interface</td>
</tr>
<tr>
<td></td>
<td>IC-Changes</td>
<td>Total number of changes made to the interface</td>
</tr>
<tr>
<td></td>
<td>IC-Additions</td>
<td>Number of parameter additions</td>
</tr>
<tr>
<td></td>
<td>IC-Deletions</td>
<td>Number of parameter deletions</td>
</tr>
<tr>
<td><strong>Data Representation (DR)</strong></td>
<td>DR-Number</td>
<td>The number of underlying changes that are made to the data that may be passed to or through the interface</td>
</tr>
<tr>
<td></td>
<td>DR-Complexity</td>
<td>Assessment of the complexity of the data representation changes</td>
</tr>
<tr>
<td><strong>Messaging Protocol (MP)</strong></td>
<td>MP-Number</td>
<td>The number of different messaging protocols involved in the software transaction</td>
</tr>
<tr>
<td></td>
<td>MP-Complexity</td>
<td>The average complexity of the messaging protocol; likely derived from experience with each messaging protocol including the number of issues previously identified with each messaging protocol</td>
</tr>
<tr>
<td></td>
<td>MP-Threshold</td>
<td>The number of messaging protocols over a predetermined threshold involved in the software transaction</td>
</tr>
<tr>
<td><strong>Systems Impacted (SI)</strong></td>
<td>SI-Number</td>
<td>Number of systems this interface impacts</td>
</tr>
<tr>
<td>Category</td>
<td>Factor</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SI-Complexity</td>
<td>The average complexity of</td>
<td>The average complexity of the systems that are impacted; This metric is open for definition; a few options could include a metric that is based on a) the size of the system (e.g., LOC), b) the number of previously identified issues (e.g., X number of defects found in the last Y builds)</td>
</tr>
<tr>
<td>SI-Threshold</td>
<td>Number of systems with a</td>
<td>Number of systems with a complexity over a predetermined threshold</td>
</tr>
<tr>
<td>SI-Depth</td>
<td>A representation of how</td>
<td>A representation of how many layers down in a software stack this interface resides</td>
</tr>
<tr>
<td>SI-Technology Boundaries</td>
<td>The number and/or type of</td>
<td>The number and/or type of different technologies involved in the software transaction (e.g., legacy (mainframe) to ground)</td>
</tr>
<tr>
<td>SI-Organizational Boundaries</td>
<td>The number and/or type of</td>
<td>The number and/or type of different organizational units involved in the software transaction</td>
</tr>
</tbody>
</table>

To assess the risk associated with interface changes, a complexity metric would be derived for each change. Each factor in the interface complexity framework would be given a particular operationalized value for each interface change. These factors would then be weighted based on the prior known risk of each factor using a regression function similar to the following:

$$ Interface\ Complexity_i = \sum_{j=1}^{n} b_j InterfaceChangeFactor_{ij} + \sum_{k=1}^{n} b_k DataRepresentationFactor_{ik} + \sum_{l=1}^{n} b_l MessagingProtocolFactor_{il} + \sum_{m=1}^{n} b_m SystemsImpactedFactor_{im} + error $$

Even though the above framework is in its preliminary stages we believe that it may be useful in order to focus resources on the most important interface changes in an organization. However, much work still remains. Specific operational guidelines for these metrics, which are likely organization specific, need to be defined. Additionally, much of the data may not be readily available and as such organizations will have to decide if perceptual, subjective evaluations are sufficiently representative. Additionally, thresholds would need to be determined. If the interface complexity change is over the specified threshold it would trigger specific action within the organization such as cross-team interface change reviews. We plan to continue our work with FedEx in applying this interface complexity index.
H. Recommendations

From our research work in the area and the case study we conducted, we now present several suggestions/recommendations related to interfaces which can be found in Table 5.

Table 5 – Organizational Recommendations for Addressing Interface Issues

<table>
<thead>
<tr>
<th>Recommendations</th>
<th>FedEx Initiative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create teams and systems that allow you to understand the what interfaces exist in the organization and how they are used</td>
<td>DiM; Cross Project Interface Reviews;</td>
</tr>
<tr>
<td>Reduce the number and type of interfaces where possible</td>
<td>DiM</td>
</tr>
<tr>
<td>Standardize on messaging technologies</td>
<td>DiM</td>
</tr>
<tr>
<td>Develop an understanding of the types of interface faults that occur frequently; take time to address the root cause</td>
<td>STEP Project</td>
</tr>
<tr>
<td>Provide a means for business analysts, designers, developers, testers, architects, configuration personnel, and others to know - how software modules link together - how a change in an interface impacts other systems - the structure of the data that is passed between systems</td>
<td>STEP Project; DiM; Cross Project Interface Reviews;</td>
</tr>
<tr>
<td>Increase human communication and evaluation, especially when interface complexity is high; Utilize interface complexity index or a subset of the factors that are most salient in a particular organizational context in order to</td>
<td>Cross Project Interface Reviews; STEP Project</td>
</tr>
</tbody>
</table>

I. Conclusion

Even though technologies have evolved over the past several decades, we assert that interface issues remain as critical issues for large software development projects. In this paper we have presented an updated framework for categorizing interface faults, captured some common interface issues at a company with a complex application portfolio, shared a few approaches that have been used to address interface issues, and also developed an interface complexity index that could be used to focus limited resources on issues that might be most risky.

While you might not be able to eliminate all interface issues, the steps outlined in this paper should help to reduce the occurrence and impact of some interface issues that large organizations with complex software portfolios face.

J. Acknowledgments

We would like to thank Jimmy Middleton, William Perry and many others at FedEx for their help in setting up interviews and making arrangements that made this project possible.

K. References


Governance of Software Testing: Impact of Distinct Testing Unit, Reporting Structure, and One-to-One Matching

Xihui Zhang, University of North Alabama, xzhang6@una.edu
Colin Onita, University of Memphis, cgonita@memphis.edu
Jasbir Dhaliwal, University of Memphis, jdhaliw@memphis.edu

ABSTRACT
The proliferation of complex, interconnected, and real-time business applications is leading to software testing becoming a critical component of software development. This is leading to information technology (IT) managers struggling with pragmatic governance mechanisms for integrating testing with development. However, governance issues pertaining to how software testing is organized at strategic, tactical, and operational levels have not been received adequate attention in the literature. This study explores the impact of three specific governance mechanisms, including the existence of a distinct corporate testing unit, developers and testers reporting to different executives, and the existence of one-to-one matching between developers and testers, for the organization integration of testing with development. A national survey of 196 of software development professionals was undertaken to investigate these impacts on a set of dependent variables comprising organizational, group, and individual outcomes. The results indicate that these governance mechanisms have significant impacts that need to be considered for successful integration of development and testing.

Keywords: Software testing, IT governance, software testing governance.

1. INTRODUCTION
The efficient and effective production of software for business applications requires intensive collaboration between two major distinct set of information technology (IT) professionals: (1) developers who program code into functional software based on requirements, and (2) testers who validate and verify the software to establish its functionality and acceptability. The organizational integration of these two groups has been a difficult proposition over the years and continues to be a key IT management challenge (Miller, 2009). Recent work by Cohen et al. (2004), Pettichord (2000), and Zhang et al. (2008) suggests that it remains a conflicted relationship necessitating explicit organizational and managerial action for resolution.

In the early days of business systems (1950’s and 60’s), both development and testing functions were entrusted to the same individual given that testing had not emerged as a distinct field (Royce, 1970). The failure of the early approaches led to the advent of structured methods of software development in the form of software development lifecycles which recognized testing as a distinct sequential stage after coding. This led to the growth of software testing as a distinct profession and science – and the emergence of software development and testing integration as a crucial organizational IT governance challenge. Recent advances in agile methods for both software development and testing (Crispin & Gregory, 2009; Highsmith & Cookburn, 2001; Lee, 2008) have added
increased impetus to the need for resolving this challenge. The fact that the proportion of total IT acquisition expenditures that are spent on software testing is going up, because of the increased complexity, application inter-connectivity, global-scale, and real-time nature of modern business systems, also call for increased focus on this issue as a managerial and theoretical phenomenon in software engineering.

Given the dearth of empirical studies that have explored this phenomenon to provide guidance for industrial practices, software organizations are using a wide diversity of approaches (which are often contradictory) to cope while continuing to make the case that it is a critical area of concern. Consider the following two examples:

1. Software testers at Microsoft Corporation: (a) are not part of a distinct organizational unit for testing, (b) report to the same executives as developers, and (c) are matched to particular developers in agile development teams (Page et al., 2008).

2. Software testers at FedEx IT Services: (a) are part of a distinct organizational unit for testing, (b) report to a different executive than developers, and (c) are not matched to particular developers (Miller, 2009).

While both organizations are known for their innovativeness in the software engineering of business systems, they obviously are using completely contradictory IT governance methods for integrating development and testing. This paper investigates the underlying effectiveness of such IT governance practices for software testing by empirically exploring the organizational, group, and individual impacts of strategic, tactical, and operational software testing governance mechanisms.

The two specific research questions driving this research are:

1. What are the key components of a theoretical framework that can guide IT governance decisions pertaining to the integration of software development with testing?

2. What is the empirical impact of various IT governance mechanisms on organizational, group, and individual level variables pertinent to the integration of development with testing?

The paper proceeds as follows. Section 2 tackles the first research question and develops a theoretical framework that captures the key dimensions of software testing governance by drawing on the prior literature on both software engineering and IT management. Section 3 then describes an empirical study that was undertaken to investigate aspects of the theoretical framework. This is followed by Section 4 that details our research findings and Section 5 that explores the implications of our findings in relation to both industry best practices and theoretical development. Finally, Section 6 recognizes the limitations of our approach while providing pointers for future research as part of an overall conclusion.

2. THEORETICAL DEVELOPMENT
The objective of this study is to examine the impact of the governance of software testing on a set of dependent variables that represent impacts. Specifically, this study explores the impact of three governance mechanisms, including the existence of a distinct corporate testing unit, developers and testers reporting to different executives, one-to-one matching between developers and testers, which are governance mechanisms identified at strategic, tactical, and operational level, respectively. The dependent variables can be classified into three categories: organizational, group, and individual impacts. Organizational impacts are represented by software quality, value of testing, and developer/tester alignment. Group impacts are represented by strategic alignment, capability alignment, shared understanding, common language, and social systems of knowing. Individual impacts are represented by task conflict, relationship conflict, and process conflict, trust between developer and tester, partnership between developer and tester, managing developer’s expectations, tester knowledge of development methods, developer knowledge of testing methods, and job satisfaction. The dependent variables are chosen because they are closely related to the context of software development and testing.

Guided by theory and past research, a theoretical framework in Figure 1 is proposed which asserts that the existence of a distinct corporate testing unit, developers and testers reporting to different executives, and one-to-one matching between developers and testers will have significant impact on a set of dependent variables as listed in Figure 1. To simplify the data analysis process, we tested three separate models, each with only one independent variable and the same set of the dependent variables. We provide theoretical support for the hypothesized relationships in the following sections.
Figure 1. A Theoretical Framework

Relating IT Governance to Software Testing Governance
Weill and Ross (2004) have demonstrated the criticality of IT governance by showing that firms with better than average governance earn at least 20 percent higher return on assets than organizations with weaker governance. This suggests that it may be worthwhile for software engineering executives to carefully consider a governance perspective to integrating the development and testing functions.

Using an integration perspective to IT decision making and governance initially proposed by Teo and King (1999), four aspects of integration between development and testing need to be considered: (1) Administrative integration: in which budgets and schedules are pooled between software development and testing; (2) Sequential integration: in which software development decisions provide directions for software testing decision making; (3) Reciprocal integration: in which software development and testing decisions are mutually influential; and (4) Full integration: in which software development and testing decisions are concurrently made in the same process.

The literature on IT governance yields several nuanced and related definitions that can be applied to the case of the role of development and testing in software development. Generally, IT governance comprises the leadership, organizational structures and processes that ensure that the organization’s IT sustains and extends the organization’s
strategy and objectives (ITGI, 2003; Van Grembergen, 2002). Applying this definition to the case of the integration of development and testing functions in software development, we can define software testing governance as involving the leadership, organizational and integrative processes that ensure the successful implementation of software development strategy. Software testing governance needs to be differentiated from day-to-day software testing management that focuses on what specific software testing decisions are being made. Rather, software testing governance is the set of decisions about who makes software testing decisions (Weill, 2004) and how these decisions are made. In other words, it prescribes the structures and processes through which the organization’s testing objectives are set, and defines the means for attaining those objectives and monitoring performance.

The IT governance literature emphasizes the importance of the relationship/overlap between corporate/enterprise governance and IT governance and builds upon the former (Luftman & Brier, 1999; Sambamurthy & Zmud, 1999; Weill, 2004). Similarly, our approach involves defining and thinking about software testing governance using the IT governance literature as the base for theory development. Thus software testing governance represents the enterprise’s software engineering management system through which its portfolio of software development and testing efforts are directed and controlled. In essence, software testing governance can therefore be viewed as the distribution of software testing decision-making rights and responsibilities among software engineering stakeholders, and the procedures and mechanisms for making and monitoring strategic decisions regarding software testing.

Given that the key issue in software testing governance pertains to its integration with software development, it is also important to consider the relationship between governance and strategic alignment. Webb et al. (2006, p. 7) have taken such an approach to try amalgamating the range of nuanced definitions for IT governance by proposing the following definition: “IT Governance is the strategic alignment of IT with the business such that maximum business value is achieved through the development and maintenance of effective IT control and accountability, performance management and risk management.” Using this approach software testing governance can be viewed as the strategic integration of testing with development to ensure that value (quality) in software development can be maximized through the implementation and maintenance of effective control and accountability, performance management and risk management.

Borrowing from prior IT governance studies by Peterson et al. (2002), Peterson (2003), Weill and Ross (2004), and Van Grembergen et al. (2003), software testing governance can be deployed using a mixture of various structures, processes, and relational mechanisms. Petersen (2004) relates these to capabilities in governance and provides examples of structural capabilities, process capabilities and relational capabilities. De Haes and Van Grembergen (2008, 2009) also utilize this categorization comprising structures, processes, and relational mechanisms for governance. In our view, this categorization can be transposed on the three levels of the organizational management: strategic, tactical, and operational. In our model for software testing governance (see
Figure 1), structural mechanisms are represented at the strategic level, process mechanisms are represented at the process level, and relational mechanisms are represented at the operational level. Thus, strategic structures in software testing governance pertain to institutional issues relating to organizational design that specify the precise formal organizational role of the testing group. Similarly, tactical processes in software testing governance specify controlling, coordinating, and reporting guidelines between testing and development groups, while operational relational mechanisms in software testing governance clarify the participative and collaborative relationships between developers and testers as they work together in software engineering. Relational mechanisms are vital in this software testing governance framework as they dictate the informal day-to-day working interactions between developers and testers, even when the appropriate formal strategic structures and tactical and processes are in place (Weill & Broadbent, 1998; Keill et al., 2002; Callahan & Keyes, 2003).

Three Levels of Software Testing Governance

**Strategic structures for software testing governance**: Prior literature on formal structures for governing software testing is largely non-existent. However, various aspects can be culled from the IT governance literature as being pertinent to the integration of software testing and development. These include: the existence of a distinct organizational unit for software testing, its placing in the organizational hierarchy, formalized strategic steering committees for software engineering management, formal structures for measuring and managing strategic alignment between distinct but related organizational units, and formalized high-level participation on executive committees (De Haes & Van Grembergen, 2008, 2009). Amongst these considerations, the most significant pertains to the existence of a distinct organizational unit for software testing (Miller, 2009). It can be argued that the institutionalization of such a distinct testing unit facilitates formal planning and control governance of software testing. It also promotes the growth of professionalism and identity for the testing group and clarifies the specific focal points for strategic decision making pertaining to budgets, resources, methodologies, and strategic scope of testing. The existence of a distinct organizational unit also facilitates the measurement of return-on-investment and value metrics pertaining to the contribution of the unit at a strategic level. It also provides software testers the opportunity to provide input into strategic organizational deliberations that have the potential of impacting them. The existence of a distinct testing unit also provides the basis for strategic considerations pertaining to centralized, decentralized, and federated governance mechanisms (Sambamurthy & Zmud, 1999) as part of strategic analysis.

**Tactical processes for software testing governance**: There is also a dearth of studies that have focused on software testing governance at this level. The general IT governance literature identifies reporting structures, service level agreements, the use of methodologies such as balanced scorecards and COBIT, and charge-back arrangements as being pertinent (De Haes & Van Grembergen, 2008, 2009). Amongst these, relative reporting structures for development/testing, use of agile versus lifecycle software engineering methodologies and charge-back arrangements for software testing can be identified as being the most relevant to the integration of software development and
testing. Developers reporting to a different manager than testers can be expected to create significant integration and alignment issues as compared the case where they reported to the same manager. The use of agile methodologies (Crispin & Gregory, 2009; Highsmith & Cookburn, 2001; Lee, 2008) for software development is generally associated with having developers and testers report to the same executive such as at Microsoft Corporation (Page et al., 2008). This is because agile processes necessitate intensive collaboration between developers and testers working together in “scrum” whose work is coordinated in prescribed “sprints” (Larman & Vodde, 2008). Organizations subscribing to the use of systems development lifecycle methodologies can generally be expected to opt for reporting processes where developers report to different managers than testers. The structured stages of the lifecycle, whereby a testing phase generally follows a coding/development phase, facilitate this as prescribed by Teo and King’s (1997) notion of sequential integration. Given that units of code are passed over to testers by developers as formalized process handoffs, the two related activities can be managed using separate reporting mechanisms. Given that the role of the testing function is to verify and validate the work of developers by providing feedback about defects and bugs that are found in testing, chargeback processes whereby testing costs are “charged” back to development groups also represent a key governance aspect at this level.

**Operational relational mechanisms for software testing governance:** Significant literature exists in relation to the operational governing mechanisms for software testing. Most of this relates to the measurement and management of conflict between developers and testers (Cohen et al., 2004; Pettichord, 2000; Zhang et al., 2008). In addition to this, pertinent aspects that can be culled from the IT governance literature (Petersen, 2004; De Haes & Van Grembergen, 2008, 2009) and applied to our context include job rotation, co-location, cross training, knowledge management, and informal interaction between developers and testers. Given the relative roles that that developers and testers play in software engineering, these can be viewed as being sub-aspects of a higher level construct that can be termed: one-to-one matching between particular developers and testers. A specific tester working on a stable basis to provide defect and quality feedback to a particular developer can be expected over time to yield defined impacts.

**Three Levels of Impacts of Software Testing Governance**

Following the literature, we chose salient dependent variables that are important in the day to day management of software development organizations, and which are influenced by the choices regarding to software testing governance issues. Specifically, our paper includes constructs such as shared understanding, partnership, and common language that were deemed by Preston and Karahanna (2009) and Luftman and Kempaiah (2007) to be important components of a good IT strategy. Shared understanding includes the social aspects of employee relationships; Partnership measures the rapport between sub-units and their interaction including issues of trust, shared goals, and values; while common language measures the commonality of ideas and concepts between. Value of testing (Luftman & Kempaiah, 2007) deals with perceptions of the benefits of interaction as well
as the metrics used to quantify the performance output of a sub-unit and its relative contribution to the other sub-unit’s output.

Alignment is another important concept that has been studied in IS literature (Henderson & Venkatraman, 1993; Luftman & Kempaiah 2007; Preston & Krahanna, 2009) and is also present in our list of dependent variables. Henderson and Venkatraman (1993) indentify two main components of alignment – strategy and capability alignment, which we measure as alignment between two individual subunits of the IT department. Following Preston and Krahanna (2009) our paper uses social systems of knowing, knowledge of respective methods and managing expectations as salient variables that can be influenced by governance choices.

Clearly, the choices of software testing governance can have impacts on an organization as a whole (e.g., software development organizations), groups (e.g., development groups and testing groups), and individuals (e.g., developers and testers). As such, we categorize the dependent variables representing impacts of the choice of software testing governance into three levels, including organizational, group, and individual (see Figure 1 above). The dependent variables that represent organizational impacts include: software quality, value of testing, and developer/tester alignment. The dependent variables that represent group impacts include: strategic alignment, capability alignment, shared understanding, common language, and social systems of knowing. The dependent variables that represent individual impacts include: task, relationship, and process conflict, trust between developer and tester, partnership between developer and tester, managing developer’s expectations, tester knowledge of development methods, developer knowledge of testing methods, and job satisfaction.

**Organizational impacts:** For the organization impacts we investigate three salient outcomes of governance choices: software quality, value of testing, and developer/tester alignment. First we look at the overall software quality as an important organizational outcome. The quality of software developed is an essential indicator of and has important implications to the success of a software development organization. We posit that all three independent variables (i.e., the existence of a distinct corporate testing unit, developers and testers reporting to different executives, and one-to-one matching between developers and testers) will influence the quality of the software developed. Having a distinct corporate testing unit allows the testing unit to provide a more cogent and efficient testing strategy and implementation than when testing is only a small part of the development (Miller, 2009). Similarly, developers and testers reporting to different executives creates a stronger testing unit that is better able to both act as a validation entity as well as an improvement entity for the software developed. Finally, having one-to-one matching between developers and testers has been shown to improve the quality of software by providing immediate and personalized feedback about a piece of software (Page et al., 2008).

\[ \text{H1a: The existence of a distinct corporate unit for software testing will influence the quality of software developed.} \]
H2a: Developers and testers reporting to different executives will influence the quality of software developed.
H3a: One-to-one matching between developers and testers will influence the quality of software developed.

The second organizational impact investigated is the perceived value of testing in the organization (Luftman & Kempaiah, 2007). It is easy to see how both the having a distinct corporate testing unit and developers and testers reporting to different executives would improve the perceived value of testing to the organization. Having a clear delineation of departments and responsibility allows the organization to both clearly perceive and quantify the outputs and benefits of testing to the organization. It also allows testing to have a more coherent view of itself and to be more in control of its strategies and capabilities. One-to-one matching pairs up individual developers with individual testers and thus leads to the creation of personal rapport and relationships between testers and developers that lead to a better perception of testers in software development and in the overall organization.

H1b: The existence of a distinct corporate unit for software testing will influence organizational understanding of the value provided by testing.
H2b: Developers and testers reporting to different executives will influence organizational understanding of the value provided by testing.
H3b: One-to-one matching between developers and testers will influence organizational understanding of the value provided by testing.

The third organizational impact investigated pertains to the alignment (Henderson & Venkatraman, 1993; Luftman & Kempaiah, 2007; Preston & Karahanna, 2009) between the testing and development units. All three independent variables positively impact the alignment between the software development and testing subunits. The first two variables impact alignment by providing an independent scaffold on which both testing and development can build their strategies. Second, since both units are independent of each other, they can build internally the specific capabilities that they require to enact their stated strategies. Also, since both units are sovereign, independent units, they can interact on similar terms and reach a common understanding of software creation goals and strategies. One-to-one matching also leads to better alignment between development and testing due to the increase communication and rapport between individual developers and testers. Since they are in constant communication and interaction, individual developers and testers are more likely to create a common language and understanding of their jobs (Preston & Karahanna, 2009) as well as be better able to know what the needs of the other party are.

H1c: The existence of a distinct corporate unit for software testing will influence developer/tester alignment.
H2c: Developers and testers reporting to different executives will influence developer/tester alignment.
H3c: One-to-one matching between developers and testers will influence developer/tester alignment.

Group impacts: For the group impacts we investigate five salient outcomes of governance choices: strategic alignment, capability alignment, shared understanding, common language, and social systems of knowing. As part of alignment between developers and testers we investigate both the strategy and capability alignment (Henderson & Venkatraman 1993) of the development and testing groups. Having a distinct testing group and a distinct reporting structure helps with both strategy and capability alignment since both testing and development can create their own individual strategies as well as internally coherent ways of implementing said strategies buy building internal capabilities, by deploying the right tools and employing the correct processes. One-to-one matching also improves both the strategy and capability alignment of the testing and development groups by creating a common language, rapport and understanding between individual testers and developers (Preston & Karahanna, 2009).

H1d: The existence of a distinct corporate unit for software testing will influence strategic alignment between developers and testers.
H2d: Developers and testers reporting to different executives will influence strategic alignment between developers and testers.
H3d: One-to-one matching between developers and testers will influence strategic alignment between developers and testers.

H1e: The existence of a distinct corporate unit for software testing will influence capability alignment between developers and testers.
H2e: Developers and testers reporting to different executives will influence capability alignment between developers and testers.
H3e: One-to-one matching between developers and testers will influence capability alignment between developers and testers.

Shared understanding, common language, and social systems of knowing (Preston & Karahanna, 2009) are also influenced by software testing governance choices. Having a distinct testing unit and reporting to different executives will decrease the levels of shared understanding and common language, given that testers and developers are no longer working together under the same structure and leadership. This will also increase the level of informal interaction between developers and testers, positively influencing the levels of social systems of knowing. On the other hand, one-to-one matching will positively influence the levels of shared understanding and common language between developers and testers since they have a closer and more personal interaction.

H1f: The existence of a distinct corporate unit for software testing will influence shared understanding between developers and testers.
H2f: Developers and testers reporting to different executives will influence shared understanding between developers and testers.
H3f: One-to-one matching between developers and testers will influence shared understanding between developers and testers.

H1g: The existence of a distinct corporate unit for software testing will influence common language between developers and testers.
H2g: Developers and testers reporting to different executives will influence common language between developers and testers.
H3g: One-to-one matching between developers and testers will influence common language between developers and testers.

H1h: The existence of a distinct corporate unit for software testing will influence social systems of knowing between developers and testers.
H2h: Developers and testers reporting to different executives will influence social systems of knowing between developers and testers.
H3h: One-to-one matching between developers and testers will influence social systems of knowing between developers and testers.

**Individual impacts:** For the individual impacts we investigate nine salient outcomes of governance choices: task conflict, relationship conflict, process conflict, trust between developer and tester, partnership between developer and tester, managing developer’s expectations, tester’s knowledge of development methods, developer’s knowledge of testing methods, and job satisfaction.

Having a distinct corporate testing unit and having developers and testers reporting to different executives would lower the interaction frequency and intensity between developers and testers. This would lower the level of task conflict (conflict over work or task-related issues), relationship conflict (conflict over personal or identify issues), and process conflict (conflict over duty and resource delegation issues). As developers and tester are now working in different units and reporting to different executives, individual communication and interaction has a more formal structure and usually has to go through multiple levels of the two departments. This would lower the level of trust and partnership between developer and tester, and knowledge of other’s methodologies and tools. However, this would increase the level of job satisfaction for both developers and testers because both would feel their importance to the organization as they have distinct units and report to different executives.

One-to-one matching between developer and tester, on the other hand, would increase the interaction frequency and intensity between developers and testers. This would increase the level of task conflict, relationship conflict, and process conflict. One-to-one matching would be beneficial to both the level of partnership between developer and tester as well as to the level of understanding of each other’s methods. Since individual developers and individual testers are paired up for the duration of a project, they become closer to each other as well as closer to the methods and tools used by the other party. This increases the inherent partnership in the pair as well as the level of knowledge about the task at hand.
This would also increase the ability for testers to better manage developer’s expectations and the level of job satisfaction.

H1i: The existence of a distinct corporate unit for software testing will influence the level of task conflict between developers and testers.
H2i: Developers and testers reporting to different executives will influence the level of task conflict between developers and testers.
H3i: One-to-one matching between developers and testers will influence the level of task conflict between developers and testers.

H1j: The existence of a distinct corporate unit for software testing will influence the level of relationship conflict between developers and testers.
H2j: Developers and testers reporting to different executives will influence the level of relationship conflict between developers and testers.
H3j: One-to-one matching between developers and testers will influence the level of relationship conflict between developers and testers.

H1k: The existence of a distinct corporate unit for software testing will influence the level of process conflict between developers and testers.
H2k: Developers and testers reporting to different executives will influence the level of process conflict between developers and testers.
H3k: One-to-one matching between developers and testers will influence the level of process conflict between developers and testers.

H1l: The existence of a distinct corporate unit for software testing will influence level of trust between developers and testers.
H2l: Developers and testers reporting to different executives will influence level of trust between developers and testers.
H3l: One-to-one matching between developers and testers will influence level of trust between developers and testers.

H1m: The existence of a distinct corporate unit for software testing will influence partnership between developers and testers.
H2m: Developers and testers reporting to different executives will influence partnership between developers and testers.
H3m: One-to-one matching between developers and testers will influence partnership between developers and testers.

H1n: The existence of a distinct corporate unit for software testing will influence tester’s capability of managing developer’s expectations.
H2n: Developers and testers reporting to different executives will influence tester’s capability of managing developer’s expectations.
H3n: One-to-one matching between developers and testers will influence tester’s capability of managing developer’s expectations.
H1o: The existence of a distinct corporate unit for software testing will influence tester knowledge of development methods.
H2o: Developers and testers reporting to different executives will influence tester knowledge of development methods.
H3o: One-to-one matching between developers and testers will influence tester knowledge of development methods.

H1p: The existence of a distinct corporate unit for software testing will influence developer knowledge of testing methods.
H2p: Developers and testers reporting to different executives will influence developer knowledge of testing methods.
H3p: One-to-one matching between developers and testers will influence developer knowledge of testing methods.

H1q: The existence of a distinct corporate unit for software testing will influence job satisfaction.
H2q: Developers and testers reporting to different executives will influence job satisfaction.
H3q: One-to-one matching between developers and testers will influence job satisfaction.

3. RESEARCH METHODOLOGY
Measurement Items
All measurement items for both independent variables and dependent variables were either adapted from existing scales or derived from prior literature. The preliminary instrument was pilot tested for appropriateness and clarity, following Churchill (1979).

Specifically, the existence of a distinct corporate testing unit was measured by one item: Software testing represented an identifiable and distinct organizational unit. Developers and testers reporting to different executives was measured by one item: Developers reported to a different executive than testers. The existence of one-to-one matching between developers and testers was measured by one item: Testers were largely assigned to support particular developers. Respondents were asked to score these three measurement items on 7-point Likert-type scales anchored at (1) = strongly agree and (7) strongly disagree.

Software quality was measured using a six-item scale (see Table 1 for the exact measurement items, same below) adapted from scales developed and validated by Barki and Hartwick (2001), measuring six dimensions of the construct: functionality, reliability, usability, efficiency, maintainability, and portability. This adapted six-item scale is in accordance with software quality measurement scales recommended by Issac et al. (2003) and Ortega et al. (2003). Respondents were asked to score the measurement items on 7-point Likert-type scales anchored at (1) = not at all and (7) = definitely.
A seven-item scale was created for the measurement of the construct of value of testing, based on Luftman and Kepaiah’s (2007) framework. The constructs of developer tester alignment (3 items), strategic alignment (3 items), and capability alignment (3 items) were adapted and expanded from Preston and Karahanna (2009) and Henderson and Venkatraman (1993). Items for the measurement of shared understanding (4 items), common language (3 items), and social systems of knowing (3 items) were adapted from Preston and Karahanna (2009). Respondents were asked to score the measurement items on a 7-point Likert-type scales anchored at (1) = strongly agree and (7) = strongly disagree.

Task conflict, relationship conflict, and process conflict each was measured using a three-item scale adapted from scales developed and validated by Jehn and Mannix (2001) and Barki and Hartwick (2001), measuring how much, how often, and how intense of the specific conflict. Respondents were asked to score the measurement items on 7-point Likert-type scales anchored at (1) = none and (7) = a lot for the “how much” question, (1) = never and (7) = frequently for the “how often” question, and (1) = not intense and (7) = intense for the “how intense” question.

Trust between developer and tester was measured using a four-item scale adapted from scales developed and validated by Simon and Peterson (2000) and Peterson and Behfar (2003), measuring four aspects of the construct: expectations of truthfulness, certainty of trust, integrity, and living up to one’s word. Respondents were asked to score the measurement items on 7-point Likert-type scales anchored at (1) = always and (7) = never.

Based on Luftman and Kepaiah’s (2007) framework, a four-item scale was created for the measurement of the construct of partnership between developer and tester. The construct of managing developer’s expectations (4 items) was adapted from Preston and Karahanna (2009). Items for the measurement of the constructs of tester knowledge of development methods (5 items) and developer knowledge of testing methods (5 items) were also adapted from Preston and Karahanna (2009). Respondents were asked to score the measurement items on a 7-point Likert-type scales anchored at (1) = strongly agree and (7) = strongly disagree.

Job satisfaction was measured using a five-item scale adapted from scales developed and validated by Wright and Cropanzano (1998), measuring five dimensions of the construct: degree of satisfaction with the work itself, degree of satisfaction with co-workers, degree of satisfaction with the way being supervised, degree of satisfaction with opportunities for promotion, and degree of satisfaction with pay and benefits. Respondents were asked to score the measurement items on 7-point Likert-type scales anchored at (1) = strongly agree and (7) = strongly disagree.

Table 1. Constructs and Measurement Items

<table>
<thead>
<tr>
<th>Construct</th>
<th>Measurement Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinct Testing Unit</td>
<td>Software testing represented an identifiable and distinct</td>
</tr>
<tr>
<td>Reporting Structure</td>
<td>Developers reported to a different executive than testers</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>One-to-One Matching</td>
<td>Testers were largely assigned to support particular developers</td>
</tr>
<tr>
<td>Software Quality</td>
<td>The software developed is reliable (it is always up and running, runs without errors, and does what it is supposed to do).</td>
</tr>
<tr>
<td></td>
<td>It is easy to tell whether the software is functioning correctly.</td>
</tr>
<tr>
<td></td>
<td>The software can easily be modified to meet changing user requirements.</td>
</tr>
<tr>
<td></td>
<td>The software is easy to maintain.</td>
</tr>
<tr>
<td></td>
<td>The software is easy to use.</td>
</tr>
<tr>
<td></td>
<td>The software performs its functions quickly.</td>
</tr>
<tr>
<td>Value of Testing</td>
<td>There are established testing metrics for demonstrating the value of testing to the organization.</td>
</tr>
<tr>
<td></td>
<td>There are established development metrics to demonstrate the value of development to the organization.</td>
</tr>
<tr>
<td></td>
<td>The organization uses balanced measurements that are understood and accepted by both development and testing, to measure their relative contributions.</td>
</tr>
<tr>
<td></td>
<td>There are explicit service level agreements in place for assessing the contribution of testing to software development.</td>
</tr>
<tr>
<td></td>
<td>There are explicit benchmarking standards available for assessing the contribution of the testing group.</td>
</tr>
<tr>
<td></td>
<td>There are formal assessments and reviews conducted for evaluating the success of testing efforts.</td>
</tr>
<tr>
<td></td>
<td>Continuous improvement processes exist for advancing testing efforts.</td>
</tr>
<tr>
<td>Developer/Tester Alignment</td>
<td>The software testing strategy is congruent with the software development strategy in your organization.</td>
</tr>
<tr>
<td></td>
<td>Decisions in test planning are tightly linked to decisions in development planning.</td>
</tr>
<tr>
<td></td>
<td>Our testing and development strategy are closely aligned.</td>
</tr>
<tr>
<td>Strategic Alignment</td>
<td>The scope of the development group is tightly linked with that of the testing group.</td>
</tr>
<tr>
<td></td>
<td>The governance of the development group is in harmony with that of the testing group.</td>
</tr>
<tr>
<td></td>
<td>The resources of the development group are aligned with those</td>
</tr>
</tbody>
</table>
of the testing group.

| Capability Alignment | The software testing processes is congruent with the software development strategy in your organization.  
Our testing infrastructure is tightly integrated with development infrastructure.  
Our testing and development capabilities are closely aligned. |
|----------------------|---------------------------------------------------------------|
| Shared Understanding | Testing and development members have shared understanding of the role of testing in our organization.  
Testing and development members have a shared view of the role of testing as a critical component in meeting the goals of the corporate IS unit.  
Testing and development members have a shared understanding of how testing can be used to increase the quality and productive of our software development operations.  
Testing and development members have a common view about the prioritization of how resources are allocated within the corporate IS unit. |
| Common Language      | Developers and testers share a common language in our conversations.  
Testers primarily use development terminology when interacting with software developers.  
Testers avoid using testing jargon when interacting with software developers. |
| Social Systems of Knowing | Developers have regular informal contact with testers.  
Developers regularly socialize with testers outside of the work setting (social gatherings, golf, tennis, etc.).  
Developers have regular informal exchanges with testers. |
| Task Conflict        | How much work-related disagreement was there between developers and testers?  
How often did work-related disagreements occur between developers and testers?  
How intense were work related disagreements between developers and testers? |
| Relationship Conflict | How much emotional tension was there between developers and testers?  
How often did emotional tension develop between developers and testers? |
| Process Conflict | How intense was the emotional tension between developers and testers?  
  How much disagreement about who should do what was there between developers and testers?  
  How often did disagreements about who should do what occur between developers and testers?  
  How intense were disagreements about who should do what between developers and testers? |
|------------------|------------------------------------------------------------------------------------------------|
| Trust between Developer and Tester | To what extent were developers and testers truthful to each other?  
  To what extent could developers and testers trust each other?  
  To what extent did developers and testers show integrity in their interactions?  
  To what extent could developers and testers count on the other to live up to their word? |
| Partnership between Developer and Tester | The testing leadership plays a direct role in IS development planning.  
  Testing and development rewards/penalties are based on shared goals and risk factors.  
  There is a high level of trust between testing and development.  
  Development and testing commonly partner to sponsor and champion IS initiatives. |
| Managing Developer’s Expectations | Testers provide insight to developers on emerging testing technologies.  
  Testers regularly provide useful feedback that assists developers in improving their coding skills.  
  Testers educate developers about the capabilities of the testing group.  
  Testers try to give developers realistic expectations about the capabilities of testing. |
| Tester Knowledge of Development Methods | In your organization, testers are well informed about present software development plans, methods and techniques.  
  In your organization, testers are well informed about future software development plans, methods and techniques.  
  In your organization, testers are well informed about software development best practices.  
  In your organization, testers are knowledgeable about the |
potential and limitations of current software development.
In your organization, testers are knowledgeable potential and limitations of the “next generation” software development.

<table>
<thead>
<tr>
<th>Developer Knowledge of Testing Methods</th>
<th>In your organization, developers are well informed about present software testing plans, methods and techniques.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>In your organization, developers are well informed about future software testing plans, methods and techniques.</td>
</tr>
<tr>
<td></td>
<td>In your organization, developers are well informed about software testing best practices.</td>
</tr>
<tr>
<td></td>
<td>In your organization, developers are knowledgeable about the potential and limitations of current software testing.</td>
</tr>
<tr>
<td></td>
<td>In your organization, developers are knowledgeable potential and limitations of the “next generation” software testing methodologies.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Job Satisfaction</th>
<th>I am satisfied with the work that I do in my job.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I am satisfied with my coworkers.</td>
</tr>
<tr>
<td></td>
<td>I am satisfied with the way I am supervised.</td>
</tr>
<tr>
<td></td>
<td>I am satisfied with opportunities for promotion in my job.</td>
</tr>
<tr>
<td></td>
<td>I am satisfied with my pay and benefits.</td>
</tr>
</tbody>
</table>

**Data Collection**

An online survey instrument was then developed, and the survey link was distributed by individual emails to software development professionals. We used “Request for Research Assistance” for the subject line of the soliciting emails. In the body of the email, we provided information about the purpose of our study. We also assured recipients that their responses would be kept completely confidential and that there would not be a way for us to link their responses back to them or to their organizations. The respondents were offered as an incentive a summary report of the survey results. A second email, serving as a reminder, was sent three weeks after the first one.

We obtained a total of 1836 unique names and their corresponding emails from three major sources: a database provided by a software testing research center, an online directory of software testers and consultants, and SourceForge.net. All in all, 196 people (10.68%) responded to the online survey. Among them, 46.4% identified themselves as developers, another 42.3% identified themselves as testers, and the remaining 11.2% identified themselves as other software development professionals. Responses were removed from the final data set if (1) they were not from developers or testers, or (2) they
contained over 60% of missing values. As a result, a total of 159 responses were included in our data analysis: 80 were from developers, and 79 were from testers.

4. DATA ANALYSIS AND FINDINGS

Data Transformation
Before analyzing the data, we transformed all the data items. Specifically, we created three new data items (DTU, RS, and OM) from the original three data items for the independent variables, including the existence of a distinct corporate testing unit, developers and testers reporting to different executives, and the existence of one-to-one matching between developers and testers, respectively. For each new data item, we assigned 1 to it if the value associated with the original data item is 1, 2, or 3, and we assigned 2 to it if the value associated with the original data item is 5, 6, or 7.

For the dependent variables, the transformation was straightforward. New data items were created, each had a value that was a simple summation of the values of its associated original data items. For instance, the value of JS equals to JS1 + JS2 + JS3 + JS4 + JS5. The only exceptions are the three conflict constructs, i.e., task conflict, relationship conflict, and process conflict, where multiplication of the values of their associated original data items was used, given that each of their three original items is measuring quantity, frequency, and intensity of the specific conflict.

Data Analysis
Three independent-samples t tests were performed using SPSS v. 17 for Windows. “Exclude cases listwise” was used for missing values. For each t test, the test variables were the dependent variables, and the grouping variables were DTU, RS, and OM, respectively. Grouping variables were considered to have made a significant difference in test variables if the p-value (strength of significance) was .10 or less (a lower p-value indicates a stronger level of difference). The primary goal was to determine whether each of three independent variables, i.e., the existence of a distinct corporate testing unit, developers and testers reporting to different executives, and the existence of one-to-one matching between developers and testers, impacts the dependent variables.

Results
The existence of a distinct corporate testing unit. The results of the independent samples t test using the existence of a distinct corporate testing unit (DTU) as the grouping variable are presented in Table 2. There are 115 data points: 100 in group 1 and 15 in group 2. The value of “software quality” reported in group 1 is significantly (p = .056) higher than that of group 2. The value of “value of testing” reported in group 1 is significantly (p = .013) lower than that of group 2. The value of “developer/tester alignment” reported in group 1 is significantly (p = .005) lower than that of group 2. The value of “strategic alignment” reported in group 1 is significantly (p = .017) lower than that of group 2. The value of “shared understanding” reported in group 1 is significantly (p = .002) lower than that of group 2. The value of “common language” reported in group 1 is significantly (p = .022) lower than that of group 2.
The above results suggest that the formalization of a distinct corporate unit for software testing (group 1) improves: (1) the quality of software developed, (2) organizational understanding of value provided by testing, (3) developer/tester alignment, (4) strategic alignment between developers and testers, (5) shared understanding between developers and testers, and (6) common language between developers and testers.

**Developers and testers reporting to different executives.** The results of the independent samples t test using developers and testers reporting to different executives (RS) as the grouping variable are presented in Table 2. There are 119 data points: 92 in group 1 and 27 in group 2. The value of “value of testing” perceived in group 1 is significantly (p = .005) lower than that of group 2. The value of “strategic alignment” perceived in group 1 is significantly (p = .033) higher than that of group 2. The value of “social systems of knowing” perceived in group 1 is significantly (p = .033) higher than that of group 2. The value of “trust between developer and tester” perceived in group 1 is significantly (p = .014) higher than that of group 2. The value of “managing developer’s expectations” perceived in group 1 is significantly (p = .033) higher than that of group 2.

The above results suggest that developers and testers reporting to different executives leads to: (1) increased organizational understanding of value provided by testing, (2) reduced strategic alignment, (3) reduced social systems of knowing, (4) lower trust between developers and testers, and (5) reduced tester’s capability of managing developer’s expectations.

**The existence of one-to-one matching between developers and testers.** The results of the independent samples t test using the existence of one-to-one matching between developers and testers (OM) as the grouping variable are presented in Table 2. There are 105 data points: 33 in group 1 and 72 in group 2. The value of “software quality” reported in group 1 is significantly (p = .059) higher than that of group 2. The value of “partnership between developer and tester” reported in group 1 is significantly (p = .081) lower than that of group 2. The value of “tester knowledge of development methods” reported in group 1 is significantly (p = .077) lower than that of group 2. The value of “developer knowledge of testing methods” reported in group 1 is significantly (p = .047) lower than that of group 2. The value of “job satisfaction” reported in group 1 is significantly (p = .006) lower than that of group 2.

The above results suggest that one-to-one matching between developers and testers (group 1) improves: (1) the quality of software developed, (2) partnership between developers and testers, (3) tester knowledge of development methods, (4) developer knowledge of testing methods, and (5) job satisfaction.

**Table 2. Independent Samples Tests with DTU, RS, and OM**

<table>
<thead>
<tr>
<th>Independent Variable</th>
<th>Distinct Testing Unit</th>
<th>Reporting Structure</th>
<th>One to one Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dependent Variable</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Organizational Impacts</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Quality</td>
<td>1.930 (.056)</td>
<td>.085 (.932)</td>
<td>1.911 (.059)</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Value of Testing</td>
<td>-2.524 (.013)</td>
<td>-2.853 (.005)</td>
<td>.004 (.997)</td>
</tr>
<tr>
<td>Developer/Tester Alignment</td>
<td>-2.845 (.005)</td>
<td>-.026 (.980)</td>
<td>-1.615 (.109)</td>
</tr>
</tbody>
</table>

**Group Impacts**

<table>
<thead>
<tr>
<th>Strategic Alignment</th>
<th>-2.434 (.017)</th>
<th>2.152 (.033)</th>
<th>-1.282 (.203)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capability Alignment</td>
<td>-.580 (.563)</td>
<td>.666 (.507)</td>
<td>-.401 (.690)</td>
</tr>
<tr>
<td>Shared Understanding</td>
<td>-3.112 (.002)</td>
<td>.819 (.414)</td>
<td>-.666 (.507)</td>
</tr>
<tr>
<td>Common Language</td>
<td>-2.320 (.022)</td>
<td>.505 (.614)</td>
<td>-.522 (.603)</td>
</tr>
<tr>
<td>Social Systems of Knowing</td>
<td>-1.358 (.177)</td>
<td>2.529 (.013)</td>
<td>-.802 (.424)</td>
</tr>
</tbody>
</table>

**Individual Impacts**

<table>
<thead>
<tr>
<th>Task Conflict</th>
<th>.658 (.512)</th>
<th>.910 (.365)</th>
<th>.749 (.456)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationship Conflict</td>
<td>.678 (.499)</td>
<td>.807 (.421)</td>
<td>-.450 (.654)</td>
</tr>
<tr>
<td>Process Conflict</td>
<td>.406 (.686)</td>
<td>1.543 (.126)</td>
<td>.677 (.500)</td>
</tr>
<tr>
<td>Trust between Developer and Tester</td>
<td>-.369 (.712)</td>
<td>2.499 (.014)</td>
<td>.302 (.763)</td>
</tr>
<tr>
<td>Partnership between Developer and Tester</td>
<td>-1.088 (.279)</td>
<td>.447 (.656)</td>
<td>-1.762 (.081)</td>
</tr>
<tr>
<td>Managing Developer’s Expectations</td>
<td>-.559 (.577)</td>
<td>2.822 (.006)</td>
<td>-.801 (.425)</td>
</tr>
<tr>
<td>Tester Knowledge of Development Methods</td>
<td>-1.417 (.159)</td>
<td>1.455 (.148)</td>
<td>-1.786 (.077)</td>
</tr>
<tr>
<td>Developer Knowledge of Testing Methods</td>
<td>-.773 (.441)</td>
<td>-.286 (.776)</td>
<td>-2.014 (.047)</td>
</tr>
<tr>
<td>Job Satisfaction</td>
<td>-.410 (.683)</td>
<td>-.974 (.332)</td>
<td>-2.812 (.006)</td>
</tr>
</tbody>
</table>

**Hypothesis Tests**

The hypotheses were assessed by examining t-values and p-values generated from the three independent-samples t tests. The hypothesis test results are summarized in Table 3. Hypotheses H1a, H1b, H1c, H1d, H1f, H1g, H2b, H2d, H2h, H2l, H2n, H3a, H3m, H3o, H3p, and H3q are supported, and the remaining hypotheses are not supported.

**Table 3. Summary of Hypothesis Tests**

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>T-Value</th>
<th>P-Value</th>
<th>Support for Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1a: The existence of a distinct corporate unit for software testing will influence the quality of software developed.</td>
<td>1.930</td>
<td>.056</td>
<td>Supported</td>
</tr>
<tr>
<td>H1b: The existence of a distinct corporate unit for software testing will influence organizational understanding of the value provided by testing.</td>
<td>-2.524</td>
<td>.013</td>
<td>Supported</td>
</tr>
<tr>
<td>H1c: The existence of a distinct corporate unit for software testing will influence developer/tester alignment.</td>
<td>-2.845</td>
<td>.005</td>
<td>Supported</td>
</tr>
<tr>
<td>H1d: The existence of a distinct corporate unit for software testing will influence strategic alignment between developers and testers.</td>
<td>-2.434</td>
<td>.017</td>
<td>Supported</td>
</tr>
<tr>
<td>H1e: The existence of a distinct corporate unit for software testing will influence capability alignment between developers and testers.</td>
<td>-.580</td>
<td>.563</td>
<td>Not supported</td>
</tr>
<tr>
<td>H1f: The existence of a distinct corporate unit for software testing</td>
<td>-3.112</td>
<td>.002</td>
<td>Supported</td>
</tr>
</tbody>
</table>
will influence shared understanding between developers and testers.

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Weight</th>
<th>Significance</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1g: The existence of a distinct corporate unit for software testing will influence common language between developers and testers.</td>
<td>-2.320</td>
<td>.022</td>
<td>Supported</td>
</tr>
<tr>
<td>H1h: The existence of a distinct corporate unit for software testing will influence social systems of knowing between developers and testers.</td>
<td>-1.358</td>
<td>.177</td>
<td>Not supported</td>
</tr>
<tr>
<td>H1i: The existence of a distinct corporate unit for software testing will influence the level of task conflict between developers and testers.</td>
<td>.658</td>
<td>.512</td>
<td>Not supported</td>
</tr>
<tr>
<td>H1j: The existence of a distinct corporate unit for software testing will influence the level of relationship conflict between developers and testers.</td>
<td>.678</td>
<td>.499</td>
<td>Not supported</td>
</tr>
<tr>
<td>H1k: The existence of a distinct corporate unit for software testing will influence the level of process conflict between developers and testers.</td>
<td>.406</td>
<td>.686</td>
<td>Not supported</td>
</tr>
<tr>
<td>H1l: The existence of a distinct corporate unit for software testing will influence level of trust between developers and testers.</td>
<td>-.369</td>
<td>.712</td>
<td>Not supported</td>
</tr>
<tr>
<td>H1m: The existence of a distinct corporate unit for software testing will influence partnership between developers and testers.</td>
<td>-.1088</td>
<td>.279</td>
<td>Not supported</td>
</tr>
<tr>
<td>H1n: The existence of a distinct corporate unit for software testing will influence tester’s capability of managing developer’s expectations.</td>
<td>-.559</td>
<td>.577</td>
<td>Not supported</td>
</tr>
<tr>
<td>H1o: The existence of a distinct corporate unit for software testing will influence tester knowledge of development methods.</td>
<td>-.1417</td>
<td>.159</td>
<td>Not supported</td>
</tr>
<tr>
<td>H1p: The existence of a distinct corporate unit for software testing will influence developer knowledge of testing methods.</td>
<td>-.773</td>
<td>.441</td>
<td>Not supported</td>
</tr>
<tr>
<td>H1q: The existence of a distinct corporate unit for software testing will influence job satisfaction.</td>
<td>-.410</td>
<td>.683</td>
<td>Not supported</td>
</tr>
<tr>
<td>H2a: Developers and testers reporting to different executives will influence the quality of software developed.</td>
<td>.085</td>
<td>.932</td>
<td>Not supported</td>
</tr>
<tr>
<td>H2b: Developers and testers reporting to different executives will influence organizational understanding of the value provided by testing.</td>
<td>-2.853</td>
<td>.005</td>
<td>Supported</td>
</tr>
<tr>
<td>H2c: Developers and testers reporting to different executives will influence developer/tester alignment.</td>
<td>-.026</td>
<td>.980</td>
<td>Not supported</td>
</tr>
<tr>
<td>H2d: Developers and testers reporting to different executives will influence strategic alignment between developers and testers.</td>
<td>2.152</td>
<td>.033</td>
<td>Supported</td>
</tr>
<tr>
<td>H2e: Developers and testers reporting to different executives will influence capability alignment between developers and testers.</td>
<td>.666</td>
<td>.507</td>
<td>Not supported</td>
</tr>
<tr>
<td>H2f: Developers and testers reporting to different executives will influence shared understanding between developers and testers.</td>
<td>.819</td>
<td>.414</td>
<td>Not supported</td>
</tr>
<tr>
<td>H2g: Developers and testers reporting to different executives will influence common language between developers and testers.</td>
<td>.505</td>
<td>.614</td>
<td>Not supported</td>
</tr>
<tr>
<td>H2h: Developers and testers reporting to different executives will influence social systems of knowing between developers and testers.</td>
<td>2.529</td>
<td>.013</td>
<td>Supported</td>
</tr>
<tr>
<td>H2i: Developers and testers reporting to different executives will influence the level of task conflict between developers and testers.</td>
<td>.910</td>
<td>.365</td>
<td>Not supported</td>
</tr>
<tr>
<td>H2j: Developers and testers reporting to different executives will influence the level of relationship conflict between developers and testers.</td>
<td>.807</td>
<td>.421</td>
<td>Not supported</td>
</tr>
<tr>
<td>H2k: Developers and testers reporting to different executives will influence the level of process conflict between developers and testers.</td>
<td>1.543</td>
<td>.126</td>
<td>Not supported</td>
</tr>
<tr>
<td>Test</td>
<td>Hypothesis</td>
<td>Coefficient</td>
<td>P-value</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>H21</td>
<td>Developers and testers reporting to different executives will influence level of trust between developers and testers.</td>
<td>2.499</td>
<td>.014</td>
</tr>
<tr>
<td>H2m</td>
<td>Developers and testers reporting to different executives will influence partnership between developers and testers.</td>
<td>0.447</td>
<td>0.656</td>
</tr>
<tr>
<td>H2n</td>
<td>Developers and testers reporting to different executives will influence tester’s capability of managing developer’s expectations.</td>
<td>2.822</td>
<td>.006</td>
</tr>
<tr>
<td>H2o</td>
<td>Developers and testers reporting to different executives will influence tester knowledge of development methods.</td>
<td>1.455</td>
<td>.148</td>
</tr>
<tr>
<td>H2p</td>
<td>Developers and testers reporting to different executives will influence developer knowledge of testing methods.</td>
<td>-0.286</td>
<td>0.776</td>
</tr>
<tr>
<td>H2q</td>
<td>Developers and testers reporting to different executives will influence job satisfaction.</td>
<td>-0.974</td>
<td>.332</td>
</tr>
<tr>
<td>H3a</td>
<td>One-to-one matching between developers and testers will influence the quality of software developed.</td>
<td>1.911</td>
<td>.059</td>
</tr>
<tr>
<td>H3b</td>
<td>One-to-one matching between developers and testers will influence organizational understanding of the value provided by testing.</td>
<td>0.004</td>
<td>0.997</td>
</tr>
<tr>
<td>H3c</td>
<td>One-to-one matching between developers and testers will influence developer tester alignment.</td>
<td>-1.615</td>
<td>.109</td>
</tr>
<tr>
<td>H3d</td>
<td>One-to-one matching between developers and testers will influence strategic alignment between developers and testers.</td>
<td>-1.282</td>
<td>.203</td>
</tr>
<tr>
<td>H3e</td>
<td>One-to-one matching between developers and testers will influence capability alignment between developers and testers.</td>
<td>-0.401</td>
<td>0.690</td>
</tr>
<tr>
<td>H3f</td>
<td>One-to-one matching between developers and testers will influence shared understanding between developers and testers.</td>
<td>-0.666</td>
<td>.507</td>
</tr>
<tr>
<td>H3g</td>
<td>One-to-one matching between developers and testers will influence common language between developers and testers.</td>
<td>-0.522</td>
<td>.603</td>
</tr>
<tr>
<td>H3h</td>
<td>One-to-one matching between developers and testers will influence social systems of knowing between developers and testers.</td>
<td>-0.802</td>
<td>.424</td>
</tr>
<tr>
<td>H3i</td>
<td>One-to-one matching between developers and testers will influence the level of task conflict between developers and testers.</td>
<td>0.749</td>
<td>.456</td>
</tr>
<tr>
<td>H3j</td>
<td>One-to-one matching between developers and testers will influence the level of relationship conflict between developers and testers.</td>
<td>-0.450</td>
<td>.654</td>
</tr>
<tr>
<td>H3k</td>
<td>One-to-one matching between developers and testers will influence the level of process conflict between developers and testers.</td>
<td>0.677</td>
<td>.500</td>
</tr>
<tr>
<td>H3l</td>
<td>One-to-one matching between developers and testers will influence level of trust between developers and testers.</td>
<td>0.302</td>
<td>.763</td>
</tr>
<tr>
<td>H3m</td>
<td>One-to-one matching between developers and testers will influence partnership between developers and testers.</td>
<td>-1.762</td>
<td>.081</td>
</tr>
<tr>
<td>H3n</td>
<td>One-to-one matching between developers and testers will influence tester’s capability of managing developer’s expectations.</td>
<td>-0.801</td>
<td>.425</td>
</tr>
<tr>
<td>H3o</td>
<td>One-to-one matching between developers and testers will influence tester knowledge of development methods.</td>
<td>-1.786</td>
<td>.077</td>
</tr>
<tr>
<td>H3p</td>
<td>One-to-one matching between developers and testers will influence developer knowledge of testing methods.</td>
<td>-2.014</td>
<td>.047</td>
</tr>
<tr>
<td>H3q</td>
<td>One-to-one matching between developers and testers will influence job satisfaction.</td>
<td>-2.812</td>
<td>.006</td>
</tr>
</tbody>
</table>

5. SUMMARY OF KEY FINDINGS

Page 23 of 27
The formalization of a distinct corporate unit for software testing (group 1) improves: (1) the quality of software developed, (2) organizational understanding of value provided by testing, (3) alignment between developers and testers, (4) strategic alignment between developers and testers, (5) shared understanding between developers and testers, and (6) common language between developers and testers.

Developers and testers reporting to different executives leads to: (1) increased organizational understanding of value provided by testing, (2) reduced strategic alignment, (3) reduced social systems of knowing, (4) lower trust between developers and testers, and (5) reduced tester’s capability of managing developer’s expectations.

One-to-one matching between developers and testers (group 1) improves: (1) the quality of software developed, (2) partnership between developers and testers, (3) tester knowledge of development methods, (4) developer knowledge of testing methods, and (5) job satisfaction.

6. REFERENCES


Invited talks and presentations.
STATISTICAL PROFILE OF THE ECLIPSE BUG DATABASE

Eric Woolley,
Vasile Rus and Sajjan Shiva

Computer Science Department
University of Memphis
Bug Report Tracking Systems

Why is defect tracking necessary?

- You can’t test everything
- Developer communication
- Difficulties of distributed development (multi-location, multi-company, multi-cultural…etc)
Bugzilla

- Created by the Mozilla organization to help with their SDLC
- Open source but actively maintained and used by Mozilla (think Firefox browser)
- Current Installations at over 940 companies
  (NASA, Citigroup, Facebook, Motorola, Yahoo, PriceGrabber, Novell, and Eclipse)
Eclipse IDE

- Very popular, open source integrated development environment

- Originally just a Java IDE, now Eclipse projects include thick/thin-client, server-side frameworks, modeling and business reporting, and embedded and mobile
Who supports the Eclipse IDE?

- The Eclipse IDE is supported and maintained by the Eclipse organization.
- The organization consists of companies which provide different levels of resources
  - Strategic members (IBM, Oracle, SAP…)
  - Enterprise members (Cisco, Motorola…)
  - Solutions members (Adobe, Intel, HP…)
  - Associate members (IDG, Ti…)

A Major Issue

What is Bug Report Triage?

- Bug triaging is currently a manual process by which designated individuals or groups evaluate submitted bug reports to determine who or where the report should be assigned. The person triaging may also determine whether a report is a duplicate of a previous report.
Why is it so difficult?

- Bug reports are evaluated and assigned through a tiered structure

Classification
  - Product
  - Component
  - Version
  - Target
Bugzilla Life Cycle

- **New** bug from a user with a confirmation or a product without UNCONFIRMED state
- **UNCONFIRMED**
  - Bug confirmed or receives enough votes
  - Bug is reopened, was never confirmed
- **NEW**
  - Developer takes possession
  - Development is finished with bug
- **ASSIGNED**
  - Development is finished with bug
- **RESOLVED**
  - Bug is resolved
  -QA verifies, solution worked
  - Bug is closed
- **REOPEN**
  - Bug is reopened
  -QA not satisfied with solution
  - Bug is reopened
- **VERIFIED**
  - Bug is verified
  - Bug is closed
- **CLOSED**

Possible resolutions:
- FIXED
- DUPLICATE
- WORKSFORME
- INVALID
Breakdown of the Eclipse Database

For the period of October 2001 through September 2009

252,391 Reports with a Resolved Status
252,391 Tickets? Is that all?

- Actually, no
- 293,369 total tickets for the time period

Which means that about 41,000 tickets are either currently assigned and being worked on, or waiting to be assigned.
Bugzilla Life Cycle
Unique Life Cycle paths

252 Unique Duplicate Bug Report Paths

3 = Top 50
2 = Top 25
1 = Top 5
### Top 5 Duplicate Life Cycle Paths and Average Close Time

<table>
<thead>
<tr>
<th>Duplicate Bug Report Paths</th>
<th>Number</th>
<th>Percent</th>
<th>Average (hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Resolved</td>
<td>24866</td>
<td>70</td>
<td>1287</td>
</tr>
<tr>
<td>New Resolved Closed</td>
<td>3764</td>
<td>11</td>
<td>5384</td>
</tr>
<tr>
<td>New Assigned Resolved</td>
<td>1829</td>
<td>5</td>
<td>5164</td>
</tr>
<tr>
<td>New Resolved ReOpened Resolved</td>
<td>1054</td>
<td>3</td>
<td>4728</td>
</tr>
<tr>
<td>New Resolved Verified</td>
<td>659</td>
<td>2</td>
<td>3038</td>
</tr>
<tr>
<td>Other</td>
<td>3522</td>
<td>9</td>
<td>8315</td>
</tr>
</tbody>
</table>
Additional Data

- Average Life Cycle changes for all bug reports - 1.82
- Most Life Cycle steps for a duplicate bug report - 18
- Average number of hours to resolve all duplicate reports - 2745 hrs
- Longest time to resolve a duplicate bug report - 66814 hrs (7.6 yrs!)
What is Bug Severity

- Bug reports are also classified by how important a problem seems to be
- Users select this when submitting a report but it can be changed by the triage or the developer
- Severity levels:
  - Blocker
  - Critical
  - Major
  - Normal
  - Minor
  - Trivial
  - Enhancement
Breakdown of Duplicate Bug Reports by Severity

- Normal: 64%
- Enhancement: 14%
- Major: 11%
- Critical: 4%
- Minor: 4%
- Blocker: 2%
- Trivial: 1%
## Average Hours to Resolve Bug Reports by Severity

<table>
<thead>
<tr>
<th>Severity</th>
<th>Average (hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocker</td>
<td>1173</td>
</tr>
<tr>
<td>Critical</td>
<td>1461</td>
</tr>
<tr>
<td>Trivial</td>
<td>2096</td>
</tr>
<tr>
<td>Major</td>
<td>2154</td>
</tr>
<tr>
<td>Normal</td>
<td>2367</td>
</tr>
<tr>
<td>Minor</td>
<td>3416</td>
</tr>
<tr>
<td>Enhancement</td>
<td>5412</td>
</tr>
</tbody>
</table>
Concerns and Conclusion

Concerns

- High average number of hours to resolve bug reports
- 40,000 open or unassigned reports at any given time

Conclusion

- It is impressive that such an extremely distributed organization can function so well. That it can produce and maintain such a popular and widely used product is truly amazing.
Reasons for the Investigation
(future work)

- Familiarize ourselves with the Eclipse/Bugzilla defect tracking system in order to understand how bug reports, especially duplicates, move through the system life cycle.

- Automatically identify possible duplicate bug reports using machine learning and natural language processing techniques.
Resources

- www.bugzilla.org
- www.eclipse.org
Functional Testing In An Agile Development Environment

Speaker: Rajeshwari Bali
Date: May 3rd 2010
Agenda

- Introduction
- Traditional Software Development
- Overview of Agile Process
- Functional Testing In An Agile Paradigm
- Take Away
Agenda

- Introduction
- Traditional Software Development
- Overview of Agile Process
- Functional Testing In An Agile Paradigm
- Take Away
Introduction

About Speaker

- Working as a Test Quality Analyst in Mphasis, an HP Company for past 5 years in Transportation and Logistics domain
- Primarily working on Test Automation and also an SME in HP Business Process Testing (BPT) framework
- Have helped customers migrate their test suites to BPT automation framework and thereby experience tangible benefits from automation testing
- Honored with ‘Annual – Luminary Award’ in Dec 2009 from amongst ten thousand team associates across Applications vertical for being innovative at work and for creating saving and value for stakeholders
Agenda

- Introduction
- Traditional Software Development
- Overview of Agile Process
- Testing In An Agile Paradigm
- Way Forward
Traditional Software Development

- A sequential development process
- Phase-based
- Progress seen flowing steadily downwards (like a waterfall)
- Time boxed – plan driven
- Detailed documentation
- Based on concept BDUP
- Phases can iterate
- Everyone has a specialized role
- Defects found very late in life cycle
- Value delivered only at the end
- Resists change
Traditional Testing Practices

- An independent unit
- “Last defender of quality” stance
- Detailed preparation and up front planning
- Strict control management
- Heavyweight documentation
- Strict entrance and exit criteria with approvals
- Very expensive to achieve 100% test coverage and automation
- Well defined test management
Agenda

- Introduction
- Traditional Software Development
- Overview of Agile Process
- Testing In An Agile Paradigm
- Take Away
Agile Software Development

- Iterative development process
- Delivers business value early
- Self-organizing cross-functional teams
- Documents are produced as required
- Accepts changes
- Defects identified and fixed early
- Frequent customers collaboration
- Promotes high levels of
  - Visibility
  - Predictability
  - Quality
Scrum
The Most Popular Agile Process

“An iterative, incremental framework for project management and agile software development.”

Key Characteristics
- Short and fixed cycles
- Requirements as user-stories go to ‘product backlog’
- Dynamic backlog – open to change
- Frequent deliveries of working software
- Constant & close communication
- Self-organized team
- Total Quality
- Not Done until tested
- Inspect & adapt

Scrum is made up of
- Three Roles
  - Product Owner
  - ScrumMaster
  - Team
- Three Ceremonies
  - Sprint planning meeting
  - Daily scrum meeting
  - Sprint review meeting
- Three Artifacts
  - The product backlog
  - The sprint backlog
  - A Burndown chart
Agenda

- Introduction
- Traditional Software Development
- Overview of Agile Process
- Functional Testing In An Agile Paradigm
- Take Away
Functional Testing in an Agile Paradigm

- If its \textit{not tested}, its \textit{not done} (definition of done)
- Retaining distinct QA functions is a key success factor
  - Supplements standard agile process
  - Includes Unit testing, regression/ integration functional testing, acceptance testing
- Testing in agile has to be automated as possible
- Quality must be maintained throughout the agile project lifecycle

<table>
<thead>
<tr>
<th>Role of Functional Testing</th>
<th>Changing Role of a Tester</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highlight project’s status</td>
<td>Participate in entire agile SDLC</td>
</tr>
<tr>
<td>Provide critical information to the project</td>
<td>Agile testers are quality guides</td>
</tr>
<tr>
<td>Testing alone does not assure quality</td>
<td>Provide an insight from customers perspective</td>
</tr>
<tr>
<td>Not restricted just to “find” defects</td>
<td>Work as an integral part of agile team</td>
</tr>
<tr>
<td>Role defined to “prevent” defects</td>
<td>Adopt &amp; adapt to agile goals &amp; objectives</td>
</tr>
<tr>
<td>Feedback given helps to improve</td>
<td>Quality is part of entire SDLC from day one</td>
</tr>
<tr>
<td>Help to build quality into the code</td>
<td></td>
</tr>
</tbody>
</table>
Functional Testing Framework

**Process**
- Planning Sprint
  - Build Test Environment
  - Identify tools
  - Identify test teams

- Development Sprints (1 to n)
  - Requirements
  - Acceptance Test
  - New features at design
  - Update automate
  - Smoke Test

- System Test
  - Integration Test
  - Acceptance Test
  - Regression

- Release
  - Exploratory
  - Refactoring

**Components**
- System Testing
- Integration Testing
- Functional Testing
- Acceptance Testing
- Regression Testing

**Roles**
- Performed by:
  - Developer
  - QA team
  - BA
  - Customer

- Stakeholders
  - Developer
  - Development PM
  - Testing & QA Manager
  - BA & Customer

**Criteria**
- Test:
  - Categorization
  - Level of testing
  - Scope
  - How often is it run?
  - Degree of automation

**Tools**
- Integration System
  - QTP
  - Selenium
  - HP QC/QTP
  - Junit
  - JUnit
test
  - Maven
  - Jenkins
  - Eclipse
  - Sonar/QM
  - JUnit
  - Java
  - Word
  - HTML
  - Rad/MDA

**Benefits**
- Identification of issues
- Acceptance test
- Confidence in code & ability to respond to changes
- Integration System
- Quality of Design
- Exploration test
- Refactoring
Automated Functional Testing is Critical in Agile

- Regardless of Agile method employed
  - Testing should be iterative
  - Support growing product backlogs & regression
- Manual processes are not accurate & reliable – when time is tight

<table>
<thead>
<tr>
<th>Need for Speed</th>
<th>Need for Repeatability</th>
<th>Need for Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Fast iterations need a fast turnaround</td>
<td>• Testing the same things the same way</td>
<td>• Ensuring comprehensive code coverage</td>
</tr>
<tr>
<td>• Being responsive to change</td>
<td>• Regression testing</td>
<td>• Requirements based testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Feeding the metrics</td>
</tr>
</tbody>
</table>
Challenges of Automated Functional Testing

- Challenges that agile brings for automated testing
  - A development exercise — requires time to plan and develop
  - Supporting a frequently changing environment
  - Need for specialized tools

- Automating all tests might not be practically possible
  - Tests requiring human opinion and creativity
  - Tests in distributed environment
  - Tests in production environment

- Addressing the challenges
  - Automate under-laying components and not just the GUI
    - Ability to stub functionality
    - Exercising web services, DLLs, JARs, APIs
  - Perform usability and exploratory testing
  - Perform manual testing as needed for non-automatable components
TDD and Functional Testing

*Test-Driven Development*

- An agile technique for code construction
- Guarantees functional requirements are met
- Typically limited to unit testing
- Clear communication of requirements
- Applying TDD to functional test levels
  - Improves UI code quality
  - Reduces need for detailed documentation
  - Reliable regression & automated tests are produced
- With this approach
  - Executable functional tests (automated) are created
  - Developers satisfy functional test cases
  - Analyst can only perform manual functional testing
  - Unit tests will also involve UI layer

Summary of benefits

- Closes a gap in test coverage
- Assures high-quality UI code
- Heavy test documentation is avoided
- Improves communication within team
- Reduces delay in test automation
- Guarantees a reliable regression test suite
Evolving Testing To Embrace Agile

**Agile Development vs. Traditional Development**

- Individuals and Interactions **over** Process and Tools
- Working Software **over** Comprehensive Documentation
- Customer Collaboration **over** Customer Negotiation
- Responding to Change **over** Following a Plan

**Manual GUI Regression Tests**
- Familiar but tedious, poor coverage

**Automated GUI Tests**
- Traditionally very fragile, specialized tools needed

**Unit Tests**
- Development limited interest in creating unit tests

**Agile Testing**
- Smaller quantity, Automation is Key
- Focused Automation
- Large Quantity

**Source:** Mike Cohn's Testing Pyramid

---

**Image:**
- Diagram showing the evolution of testing methods from traditional to agile approaches.

---

**Date:** 3 May 2010

---

**Company:** Mphasis
Agenda

- Introduction
- Traditional Software Development
- Overview of Agile Process
- Functional Testing In An Agile Paradigm
- Take Away
Take Away

- **Functional testing in an Agile project**
  - Is essential to build quality early on and reduce the cost of defects
  - Must be automated as much as possible in order to be effective
  - Supplements standard Agile processes like Test-Driven Development
  - Can be evolved from a traditional testing model
References

- Agile Manifesto [http://agilemanifesto.org](http://agilemanifesto.org)
- Agile Journals
- Agile/QA testing – Elisabeth Hendrickson
- Agile testing directions – Brian Marick
- Manage Your Testing with SCRUM – Bob Galen and RGalen Consulting Group, LLC
- Functional Test Driven Development – Dave Nicolette
Thank You!

Q & A

OUR VALUES:
• We are Open, Transparent and Honest
• We are Collaborative
• We Demand Excellence
• We Honour our Commitments

© 2009 Mphasis
The information contained herein is subject to change without notice.