The Proceedings of the Fifth International Research Workshop on Advances and Innovations in System Testing

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

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, Computer Engineering, and FedEx Institute of Technology.

step.memphis.edu
May 2-3, 2011
Memphis, Tennessee
Increasing Software Quality in Large Organizations: A Case Study with a Focus on Software Interfaces

Murad Akmanligil and Ulrika Nachtigal, FedEx
Thomas Meservy and Jasbir Dhaliwal, University of Memphis

Abstract

Complexity of software remains a salient antecedent of software quality. While there are many complexity metrics, interdependence of software systems significantly contributes to the overall complexity of developing and maintaining high quality information systems especially in large organizations. Only a paucity of studies in the academic literature have focused on software interface defects, and although a small percentage of overall defects, the cost of these defects is typically enormous often impacting multiple teams across the enterprise. In this article we present academic research that has been conducted related to software interfaces and then present a case study to investigate what one large software organization that has a complex application portfolio has proactively done to address interface related issues in their organization.

Introduction

Many of today’s software systems contain complex interdependencies between modules and between software systems. The boundaries between these systems and modules are referred to as software interfaces. It is not atypical for data to have to flow through numerous systems and modules and across numerous interfaces in order to satisfy the requests of the end user. These numerous modules, if architected appropriately, allow maximization of code reuse, which is often heralded as a core value of software engineering. Organizations with large software portfolios might literally have thousands of interfaces with numerous different operational units/departments/teams responsible for various pieces of required code. Systems and modules are often linked together in longs chains of dependencies between systems and modules. It is at the boundaries between systems that a relatively small number of significant software defects occur.

Defects related to software interfaces often require multiple teams and organizations to assess, correct, test, and to deploy fixes for. Furthermore, because interface defects involve multiple systems, often these defects are only found in later phases of testing (i.e., integration testing). These factors contribute to a relatively higher cost associated with interface defects compared to other categories of defects. Many organizations would benefit from a careful inspection of interfaces within their organizations and then a purposeful inclusion of interface-related processes to minimize the impact of defect related issues.

In this paper we review literature related to software interfaces, software interface defects, and software interface complexity. We then present a case study of a software quality initiative enacted at a Fortune 500 company that has numerous software interfaces. We share their experience of promoting awareness of interface related issues throughout the organization. Based on the experience of a quality action team, we share pragmatic recommendations of how and when interfaces can be reviewed and
evaluated throughout the software development lifecycle. We also share our analysis of the frequency of interface defects and the type of interface issues based on our previously developed framework.

**Literature**

Literature regarding software interfaces and associated defects is sparse. However, within this literature perspectives on what constitutes an interface defect are varied. Basili and Perricone (1984) provide a very broad definition of interface errors suggesting that these errors “are associated with structures existing outside the module’s local environment but which the module used.” Chillarege et al. (1992) included interface defects as one of the eight categories of their widely cited orthogonal defect classification scheme and suggested that interface defects are those that correspond “to errors in interacting with other components, modules or device drivers via macros, call statements, control blocks or parameter lists.”

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). Other researchers observed that interface defects occur more often during certain phases of the software development lifecycle (e.g., during high level design) and also noted that these defects are often found later during the test cycles (e.g., during integration testing) (Chillarege 1992). They found that interface defects accounted for 12-39% of the defects depending on the stage of the lifecycle they were observing.

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%).

Recently researchers have begun to investigate interface complexity in an effort to characterize interdependencies between modules/system boundaries (Cataldo et al. 2010). Two primary activities of the design phase of the software development lifecycle significantly impact on the specification of interfaces: 1) decomposing the overall functionality into appropriately sized/packaged modules and 2) determining inter-dependencies/relationships between those modules and other modules/resources (Cataldo et al. 2010). During design, many technical and organizational issues related to interfaces arise and should be addressed. Cataldo et al. (2010) suggests that technical factors may include characteristics such as communication efficiency (Buschmann 2007), accessibility of a module’s functionality (Kawryhaw and Robilliard 2009), usability and understandability of interfaces (Ellis et al.
and the maintainability of the system (Buschmann 2007; Henning 2009). Organizational factors include factors that influence the need for coordination between departments/teams that may be responsible for different aspects of the systems or for resources that are accessed directly or indirectly via an interface by the system under construction. Unfulfilled coordination needs increase misunderstandings which typically lead to software defects (Cataldo et al. 2010; Cataldo et al. 2009; Curtis et al. 1998; de Souza 2004; Herbsleb et al. 2006). Ambiguity related to system needs can also lead to a misunderstanding about who needs to be part of the coordinated effort and what information should be shared (de Souza 2004; Grinter et al. 1999). The lack of a clear representation of interfaces throughout the enterprise (including a lack of understanding modules/systems that consume the functionality being created/altered as well as understanding the interfaces that are being consumed) increases the likelihood of interface related defects.

As part of their study, Cataldo et al. (2010) developed an interface complexity index that was largely based on an earlier study by Bandi and colleagues (2003) in order to identify interface complexity factors that were predictive of software failures. Complexity factors included size of the software (lines of code), average change in lines of code, the number of inflow syntactic data dependencies, the number of inflow syntactic functional dependencies, the number of logical dependencies, the clustering of logical dependencies, the sum of interface sizes, and the dispersion of interfaces sizes. Based on data collected for two large systems, an analysis was conducted which found all of the factors except for the number of inflow syntactic data dependencies and the number of inflow syntactic functional dependencies to be significant predictors of software failure. It is important to note, however, that only factors that could be automatically calculated were included in their model (i.e., no organizational factors).

Previous Work

In an effort to more fully characterize interface faults that exist in the systems/technologies that are prevalent today, we developed a framework for characterizing interface defects as part of a previous research endeavor (Meservy et al. 2010). The framework consists of four main categories: definition/structure, functionality, execution, communication (see Appendix for more detail). As part of the previous research activities, we attempted to validate the framework at FedEx, which is an organization with a large and complex application portfolio. At the time, the defect tracking tool at the case site lacked sufficient details to definitively categorize defects as interface defects let alone identify the relevant sub-category of the framework. However, we conducted interviews with employees who often encountered interface related faults and collected salient examples of interface issues from them in an effort to further validate the framework. Four major categories of root causes of interface faults emerged: environment/configuration, improper data mapping, messaging system issues, and human communication. Since the time of the previous study, FedEx has begun classifying defects as “interface defects” at the time they are captured in their defect tracking system and are attempting to educate stakeholders to relevant subcategories of interface defects. We describe this recent effort below. Initial results suggest that the framework, though it needs to be tailored to a specific context, can be useful in characterizing interface-related defects in large software development efforts.
Our previous work also investigated factors that contribute to interface complexity with the basic premise that interface complexity is an antecedent of interface defects as suggested by the literature. Based on interviews and our own experience, we developed an interface complexity index that could be used to assess the risk associated with interface changes (see Appendix). Major categories of factors include interface changes, data representation, messaging protocol, and systems impacted. The index is calculated as the weighted sum of all relevant complexity factors. The coefficients (weights) should be derived based on empirical data. The factors and categories of the complexity index should serve as a starting point for organizations looking for a systematic approach for risk-based decisions regarding software quality related to interfaces.

In an effort to further understand the impact of interfaces, interface complexity, and software quality initiatives related to interfaces, we conducted a case study at FedEx who had a quality initiative related to software interfaces.

Case Study

FedEx is a Fortune 500 company that relies heavily on software technology including numerous software systems/modules that operate in conjunction 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 employ a 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.

FedEx is cognizant of the numerous interfaces that exist in their portfolio of software applications and services and has initiated multiple initiatives related to interfaces. The purpose of this case study was to capture the composition of interface-related defects at FedEx and to convey the activities of a quality initiative related to interfaces that has been conducted over the last year. Here we share recommendations from the quality action team including suggested changes to the software development process. We also share planned future directions of the effort. Organizations with numerous software interfaces developed by large numbers of development teams will likely find these results beneficial.

FedEx created a Quality Action Team (QAT) around interfaces. Quality Action Teams are relatively small groups of individuals who are brought together to address a single issue. These individuals are familiar
with the issue because they frequently encounter it in their “normal” work. Individuals participating in QAT activities typically do so above and beyond their normal responsibilities. While the team does not make direct decisions, they do “work to understand a process, gather and analyze data, determine facts and recommend process improvements”\(^1\). FedEx has a long history of Quality Action Teams and in addition to increased quality, FedEx found the following additional indirect benefits by employing this approach: increased teamwork, improved employee morale, strengthened leadership skills, improved productivity, additional revenue due to decreased costs, and more satisfied customers (Kelly 1993).

The Interface QATs activities have been primarily focused on exploring the reduction of interface related defects across the organization. Early efforts of the QAT centered around establishing a shared understanding of what constituted an interface defect, characterizing the composition of interface defects within the organization, and then identifying root causes of the defects. Subsequently, process improvements were identified.

**Early Exploration**

The first activity the QAT engaged in was to standardize the definitions of the term *interface* and *interface defect*. After reviewing existing definitions and exploring extensions, the group adopted the definition of an interface as “data, functionality, information exchanges (either synchronously or asynchronously) between two software systems produce the desired results in terms of agreed upon expectations. The exchanges may span organizational boundaries, ownership and authority.”

Generating consensus as to the definition of an interface defect was not a trivial task. The team worked through existing definitions of interface defects and also explored many nuanced questions related to what constitutes an interface defect. Some questions included—Are defects that exist because of a code change in another module that are only manifested when calling that module from a specific interface considered interface defects? What if inappropriate data successfully traversed an interface but and caused an error in another module, is that an interface defect? Separate definitions were agreed upon for documentation versus software code. Document interface defects are documentation (e.g., business requirements specifications, systems requirements specification) that caused the logic/coding error surrounding code that mapped/translated data from one system’s data model into another while implementing an interface. Software interface defects are logic or coding errors surrounding code that maps or translates data from one system’s data model into another while implementing an interface.

These definitions were used for QAT activities; however, as is common in large organizations, various definitions of what an interface defect is exist throughout the organization. Standardization of definitions is an ongoing educational process across the organization.

**Categorizing Interface Defects**

In order to understand the current composition of interface defects, the QAT worked with a cross-department, cross-manager project team dedicated to triaging incoming software defects. The goal of

\(^1\) [http://www.uscg.m2il/directives/ci/5000-5999/CI_5224_7.pdf](http://www.uscg.m2il/directives/ci/5000-5999/CI_5224_7.pdf)
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. Over a period of time, this team identified interface related defects. The QAT reviewed each of these defects and narrowed the set of interface defects to only include those that met the QATs definition. Table 1 captures the relative percentage of different types of interface faults. This table provides an initial look into the relative frequency of certain types of interface defects.

Table 1. Interface Defects Root Cause

<table>
<thead>
<tr>
<th>Root Cause</th>
<th>Percentage of Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error in mapping between different data representations of data (i.e., date formats, service representations, string to numbers, etc.)</td>
<td>21%</td>
</tr>
<tr>
<td>Error in understanding optional vs. required vs. not allowed data elements (the place the data was put was correct the usage was incorrect)</td>
<td>18%</td>
</tr>
<tr>
<td>An error was returned by the interface was not handled correctly</td>
<td>15%</td>
</tr>
<tr>
<td>Other</td>
<td>15%</td>
</tr>
<tr>
<td>There was a ‘versioning’ issue regarding the interface (wrong client jar, wrong file format)</td>
<td>12%</td>
</tr>
<tr>
<td>Error in where the data was mapped to (data was in the wrong place)</td>
<td>6%</td>
</tr>
<tr>
<td>There was a security/access issue with the use of the interface (client had not been added to SAC matrix of service, enterprise security bus had not been updated)</td>
<td>6%</td>
</tr>
<tr>
<td>Interface does not support business functionality requested</td>
<td>6%</td>
</tr>
</tbody>
</table>

Over the last year, FedEx started explicitly tracking Interface defects in their defect tracking tool. To further understand interface changes and the impact on interface defects, the QAT investigated work requests related to another product for a given release and found that 87 changes were needed to support new interfaces and 51 changes were needed for existing interfaces for a total of 138 interface related changes. During the testing cycles leading up to the release, software defects associated with the release were analyzed. First, quality analysts identified potential interface defects. 22% of the overall defects included in this group. After discussions, testing and development agreed that only 16% of those potential defects were actually interface defects. The QAT further narrowed down the classification to only a handful of interface defects based on the definition used. In the end, only 1% of the total defects were unanimously agreed upon to be interface defects. Table 2 captures the root causes of the interface defects evaluated as part of this second effort.
In addition to the work done by the QAT, the authors analyzed four releases of interface defects in order to further validate the framework that was developed as part of the previous STEP initiative (Meservy et al. 2010). Table 3 captures the count of the number of defects for specific categories of the framework and also lists the severity of these defects. The keen reader may note that over half of the interface defects identified in each release were categorized as high severity or critical severity. We suggest that this provides additional confirmatory empirical evidence of the criticality of interface defects.

Table 3 – Classification of Interface Defects using Meservy et al. framework

<table>
<thead>
<tr>
<th>Category/Factor</th>
<th>Year 1 - Release 1</th>
<th>Year 1 - Release 2</th>
<th>Year 1 - Release 3</th>
<th>Year 2 - Release 1</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Crit</td>
<td>Med</td>
<td>Low</td>
<td>Total</td>
<td>Crit</td>
</tr>
<tr>
<td>Communication</td>
<td>1</td>
<td>1</td>
<td>1%</td>
<td></td>
<td>0%</td>
</tr>
<tr>
<td>Human communication</td>
<td>1</td>
<td>1</td>
<td>1%</td>
<td></td>
<td>0%</td>
</tr>
<tr>
<td>Definition/Structure</td>
<td>1</td>
<td>4</td>
<td>23%</td>
<td>23%</td>
<td>1</td>
</tr>
<tr>
<td>Construction</td>
<td>1</td>
<td>12</td>
<td>9%</td>
<td>9%</td>
<td>4</td>
</tr>
<tr>
<td>Parameters</td>
<td>1</td>
<td>1</td>
<td>4%</td>
<td>2%</td>
<td>5</td>
</tr>
<tr>
<td>Execution</td>
<td>1</td>
<td>1</td>
<td>2%</td>
<td>2%</td>
<td>1</td>
</tr>
<tr>
<td>Error processing</td>
<td>1</td>
<td>1</td>
<td>2%</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>Messaging Issues</td>
<td>1</td>
<td>1</td>
<td>2%</td>
<td>1%</td>
<td>1</td>
</tr>
<tr>
<td>Misuse of Interfaces</td>
<td>1</td>
<td>1</td>
<td>2%</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>Functionality</td>
<td>20</td>
<td>13</td>
<td>9%</td>
<td>43%</td>
<td>62%</td>
</tr>
<tr>
<td>Changes in Functionality</td>
<td>4</td>
<td>2</td>
<td>1%</td>
<td>7%</td>
<td>10%</td>
</tr>
<tr>
<td>Inadequate Functionality</td>
<td>16</td>
<td>11</td>
<td>8%</td>
<td>13%</td>
<td>36%</td>
</tr>
<tr>
<td>New Functionality Added</td>
<td>0</td>
<td>11</td>
<td>3%</td>
<td>7%</td>
<td>22%</td>
</tr>
<tr>
<td>Other/Insufficient Detail</td>
<td>0</td>
<td>0</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Total</td>
<td>23</td>
<td>28</td>
<td>15</td>
<td>69%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Figure 1 and Figure 2 are a visual representation of these defects. Figure 1 captures the major categories of the framework and Figure 2 displays the subcategories or factors. The pie chart in each figure shows the percentage of the total defects that were found to be in each category or factor and the stacked bar chart shows the number of defects in each category/factor at four different levels of severity.
From this analysis we can see that a relatively high number of interface defects are related to inadequate functionality, changes in functionality, and issues related to the construction of the interfaces, and passing the appropriate data as parameters. It is important to note that even though the categorization was carefully done, some ambiguity exists as limited amount of information was captured in the defect tracking system that was used in the analysis. However, these results suggest specific areas for further investigation.
Analysis of Existing Processes and Recommendations

The Interface QAT performed an initial audit of development processes to identify, based on their root cause analysis, areas where further specification of deliverables related to interfaces might be useful. They also established a series of measures to capture the number and type of interface changes for a given project/initiative, the state of documentation related to interface changes within a project/initiative, and the state of test coverage related to changes in interfaces. They conducted focus groups with several teams to see how they captured and conveyed interface changes related to a project. Their efforts identified 9 different potential areas of improvement to requirements, development, and testing processes which are listed in Table 4.

Table 4. Interface Improvement Areas

<table>
<thead>
<tr>
<th>Requirements/Development</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Define business data template for business requirements specifications</td>
<td>5b) Conduct follow up interface reviews</td>
</tr>
<tr>
<td>2) Enhance systems architecture interface definitions</td>
<td>6) Revise test scenario specification</td>
</tr>
<tr>
<td>3) Define template for systems requirements specifications</td>
<td>7) Document key interface impacts in detailed test plan specification</td>
</tr>
<tr>
<td>4) Define interface specification procedure</td>
<td>8) Identify interface validation points and data in test case specification</td>
</tr>
<tr>
<td>5a) Conduct interface reviews during planning</td>
<td>9) Leverage interface comparison tools</td>
</tr>
</tbody>
</table>

Interface improvement is not the sole responsibility of a single organizational unit, rather to be effective it requires requirements, development, and testing to embrace the importance of these activities at different phases during the development lifecycle. Figure 3 maps the suggested improvement areas to deliverables during specific phases of product development.
1 – Define Business Data Template for Business Requirements Specification

As early as the definition phase, interfaces should begin to be specified. The business requirements specification (BRS) captures the business problem that needs to be addressed and the business processes that are associated with the problem. The focus of the BRS is on the business side, not the technical side. However, at this stage a logical data model could be included or other business data elements could be captured. At this stage, conceptual interfaces (i.e., high level interfaces between business processes) could be captured as well as functional areas that new requirements might need to know about.

2 – Enhance systems architecture interface definitions

The systems architecture specification (SAS) is a project-specific deliverable that could be enhanced with additional interface details including adding 1) a high-level diagram of interfaces including an as-is specification and a to-be specification and 2) a list of interfaces that are impacted by the proposed changes. The table of impacted interfaces may include the interface name, whether it is new or existing, which products/applications consume the interface, and a description of how the consumer will be impacted.
3 – Define template for systems requirements specifications

The systems requirements specification (SRS) is also a project-specific deliverable. Data captured at this stage could include fields such as the interface name, interface status (i.e., existing or new), the location of the interface specification, which team/group provides the interface, contact information, consumers of the interface, and provider and consumer contact information. Interface information in the SRS should also include detailed information about attributes or parameters of the interface including parameter names, data types, domain values, defaults, optional or required flags, the status (i.e., existing –vs.- new), and any additional relevant comments.

4 – Define interface specification procedure

The deliverables discussed in the previous recommendations are related to specific projects. The QAT recommended capturing these details for an entire release across all projects in an Interface Specification that would become its own deliverable within the development process. An interface specification would provide an earlier holistic view of interface related changes and encourage a dialogue on who takes ownership of the interface.

5a and 5b– Conduct interface reviews and follow up reviews during planning

Requirements and Development are jointly responsible to hold interface reviews once the deliverables from high level design (i.e., SAS, SRS) are complete. Quality Assurance should schedule follow up reviews in the development phase when interface details are refined. Interface reviews may yield updates to requirements and identification of launch phase execution activities.

6 – Revise test scenario specification

The test scenario specification (TSS) is written during the planning phase based on business requirements. Once interface details are published via the SAS, SRS, and Interface Specification, TSS should be updated to include these details. Interface related data in the TSS might include the name of the interface, the test scenario description, the provider of the interface, the consumer(s) of the interface, and expected results.

7 – Document key interface impacts in detailed test plan specification

The detailed test plan specification (DTPS) should capture any assumptions that are specific to the impacted interface additions and changes. It is suggested that data flows for impacted interfaced be highlighted and that strategies for interfaces be explicitly defined. Detail associated with “golden” test cases that will be traced from the interface scenarios should also be identified.

8 – Identify interface validation points and data in test case specification

The test case specification (TCS) should contain coverage for validation points outlined in the TSS and DTPS and transaction level data from all interface test cases should be identified.

9 – Leverage interface comparison tools
The final recommendation by the Interface QAT was to leverage interface comparison tools. Within FedEx, there are some initiatives underway to automatically capture and document the interfaces in use. One 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. The QAT recommends using these existing tools to capture and share relevant information where applicable throughout the software development process.

**Conclusion**

The Interface QAT has made considerable progress toward understanding interface-related issues including identifying root causes of interface defects and also identifying ways to enhance documentation and processes to ensure high quality software. The goals of the QAT over the next year include: 1) reducing interface defects during the launch phase and before they reach production, 2) close known interface documentation and procedural gaps, and 3) raise interface visibility from the initiative level to the release level. In support of these goals, the QAT will be piloting their nine recommendations with four different initiatives. It is anticipated that empirical evidence will be gathered to capture the impact of these recommendations on the reduction of interface related defects.

At times, software development processes within large organizations can be perceived to be process heavy. The QAT feels strongly that these interface related process gaps not be mandatory for all projects/interfaces. Creating a mechanism similar to the interface complexity index created as part of our previous research may provide a systematic way focus on when interface related activities should be utilized.

Clearly interface related issues are important to large organizations. In this article we presented academic research that has been conducted related to software interfaces and then presented a case study that conveyed what one large software organization that has a complex application portfolio has proactively done to address interface related issues in their organization. While each context may provide a different set of challenges, we suggest that many organizations could learn from the experience and effort captured in this article as they work to increase software quality by focusing on interface related issues.

**Acknowledgments**

We would like to thank the many participants at FedEx that contributed to this case study including Jimmy Middleton, William Perry, and other members of the Interface QAT. Shalini Bhalla, a PhD student at the University of Memphis also was instrumental in reviewing and categorizing the interface defects.

**References**


## Appendix

### Framework for Classifying Interface Defects

<table>
<thead>
<tr>
<th>Category</th>
<th>Factor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Definition / Structure</strong></td>
<td>Construction</td>
<td>Issues resulting from how the interface was constructed</td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>Construction issues resulting from the data types / size / number of the interface parameters are inappropriate or insufficient</td>
</tr>
<tr>
<td><strong>Functionality</strong></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><strong>Execution</strong></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><strong>Communication</strong></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>
## 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</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><strong>(DR)</strong></td>
<td>DR-Complexity</td>
<td>Assessment of the complexity of the data representation changes</td>
</tr>
<tr>
<td><strong>Messaging Protocol</strong></td>
<td>MP-Number</td>
<td>The number of different messaging protocols involved in the software transaction</td>
</tr>
<tr>
<td><strong>(MP)</strong></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></td>
<td>SI-Complexity</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></td>
<td>SI-Threshold</td>
<td>Number of systems with a complexity over a predetermined threshold</td>
</tr>
<tr>
<td></td>
<td>SI-Depth</td>
<td>A representation of how many layers down in a software stack this interface resides</td>
</tr>
<tr>
<td></td>
<td>SI-Technology Boundaries</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></td>
<td>SI-Organizational Boundaries</td>
<td>The number and/or type of different organizational units involved in the software transaction</td>
</tr>
</tbody>
</table>
Using Operational Business Intelligence to Inform Systems Testing

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
Quality Leadership Department
FedEx Services
Memphis, TN
{michael.cuenco, xshi}@fedex.com

Abstract: Our on-going business intelligence (BI) project is studying the identification and remediation of customer churn based on the data mining of service quality indicators. We propose to extend this research by investigating the root causes of service related problems that relate to software systems defects and failures. The use of operational business intelligence dashboards has the potential to provide early warnings of faults in systems that are not found with traditional software testing methods during systems development. We report on the current status of our research project on customer churn and propose future directions for the use of BI dashboards – or in this case, Software Quality Dashboards - to inform systems testing.

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

I. 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 collaborative 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.

Highly automated industries, such as the automotive, shipping, and computer industries, provide numerous high profile examples of errors that are detected during operations, sometimes triggering product recalls that are costly to both manufacturers and consumers. One solution to this problem is to invest more in defect detection during systems development. However some defects may still pass through existing filters and the resulting system errors may not be detected until actual operations. Hence it is critical to have a continuous software system assurance and testing approach that seeks to identify defects after deployment. In most cases, these are the software defects that are the hardest to find.

To highlight the relevance and importance of this problem, we briefly repeat a motivating example presented in previous STEP papers (Hevner and Padmanabhan 2008, Hevner and Padmanabhan 2009).

A Systems Testing Example

A small package routing system, \( S \), automatically places a package in one of two trucks based on the areas each truck is scheduled to visit. This system, \( S \), depends on two other systems, \( S_1 \) and \( S_2 \), where \( S_1 \) scans the package to determine an address and \( S_2 \) takes as input an address and returns a geographical coordinate and route to the address. On the business process side, assume that the specific business process of drivers delivering the packages is \( P_k \) and that the preceding processes in the supply chain are \( P_1, P_2, \ldots, P_{k-1} \).

Suppose there is a defect in the software for \( S_2 \) in the processing of a specific zip code, say 90210, leading \( S_2 \) to provide an incorrect geographical coordinate, thereby placing the package in the wrong truck. Over time this may result in delays to some packages intended for 90210 (depending on how soon the driver detects the problem). Clearly there are no obvious errors here, such as a system crash. However there is a software defect that causes packages to be incorrectly loaded onto trucks. In such cases what we do observe are unacceptable process outcomes (delayed packages) associated with \( P_k \). This will lead to poor service quality indicators associated with these deliveries.

Without additional data, the unsatisfactory outcome at \( P_k \) may be attributed to a possibly wide range of reasons including human error or any of the prior processes or information systems that were used in the supply chain. However if each process step and each software system kept track of its actions in a data repository, then data mining approaches may be used in learning rules that might isolate the flaw(s). In this example, a rule of the form “\( S_2 \).handle(90210) \( \rightarrow \) \( P_k = \) bad” may directly help identify the source of the software defect in module \( S_2 \).

In the 2008 STEP paper (Hevner and Padmanabhan 2008) we defined a system connection graph of technology modules (Figure 1) to be a directed graph, \( S \), with \( k \) vertices \( S_1, S_2, \ldots, S_k \). Let \( M \) be the \( k \times k \) adjacency matrix...
for the graph where $M(i, j) = 1$ indicates that there is an edge from $S_i$ to $S_j$. The graph $S$ represents technology modules and any relationships among them. We use the term “technology modules” to refer to entities that can be software and/or hardware.

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

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

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

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

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

We use this graphic representation of modules and processes to support a data mining approach for operational software systems assurance. The goal is to leverage near real-time system and process data to learn rules relating system events to specific outcomes both positive and negative. Service failures may lead to the identification and correction of software defects in critical system modules.

The subsequent sections provide a framework for our future research plans. In Sections 2 and 3 we present a summary of our research progress on the data mining of operational service quality data for the study of customer churn (Hevner et al. 2010, Padmanabhan et al. 2010). Section 4 proposes an operational business intelligence (BI) dashboard as an illustration of how near real-time monitoring of service quality can alert decision makers to real-time business concerns that may be related to software system defects and failures. Section 5 concludes the paper with a discussion of our future research directions.

![Figure 1: Operational Graph representing the systems, processes, and their interactions](image-url)
II. CUSTOMER CHURN

Before presenting details on our churn related research, it is useful to motivate the application of these ideas explicitly for the software quality assurance context. For instance, assume that this research uncovers various multivariate conditions under which customers are more (or less) likely to churn, for instance that small enterprises that experience a certain kind of service failure with a logistics provider are likely to churn with higher rates than normal. Most firms today have, or are investing in, business intelligence dashboards where such information might surface, prompting strategies for marketing or operational interventions. However, given the extent to which IT systems play a role in operations (e.g. as in Figure 1), should firms consider investing in equivalent dashboards (e.g. Software Assurance Dashboards) for continuous software quality assurance? A finding of high churn accompanying a service failure may possibly make its way into such a dashboard, triggering the technology and scoring group to delve into their own versions of root-case analyses that might examine whether any algorithmic or IT issue might have contributed to such service failures. Such proactive monitoring and management of IT systems is at the heart of our proposal and research here.

Over the past year, our research team has studied methods for data mining service quality indicators gathered during key business processes and their relationships with customer churn. 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 (Gustafsson et al. 2005). To study this question, we performed data mining on leading service quality indicators to discover possible causal relationships with 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 (Keaveney 1995). Among the potential churn factors are customer characteristics and service quality indicators (Graves et al. 1998). 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.

We studied 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.

The setting for this study is non-contractual. Customers always have multiple options for shipping packages and can easily switch shipping volumes in reaction to events. This study is therefore 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 1 we assume the existence of a churn label and learn existing churn segments retrospectively by generating one record per churned shipper. In such a case, we take a perspective of an analyst “looking back” at completed two years’ data, and asking to see what might explain the differences between churned and non-churned shippers. While such results will be useful in strategic planning, they are not actionable from an operational point of view, since the models are not predictive by design. In Phase 2 we learn data-driven methods that can intelligently learn churn alerts that can be used in real-time to determine if a customer is likely to churn. To do this, we take a predictive modeling approach, motivated strongly by the methodology suggested in a churn prediction competition summarized in Neslin et al. (2006).

2.1 PHASE 1 – RETROSPECTIVE ANALYSIS: 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:

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

The methodology is based on the approach in Zhang et al. (2004). We term rules such as these as statistical quantitative rules (SQ rules). Below is a formal definition:

**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:

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

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 \( X \).

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.

**Definition (Significant SQ rule).** For a given significance level \( \alpha \in (0, 1) \), let \( (\text{stat}_L, \text{stat}_R) \) 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 \( X \implies f(D_X) = \text{statistic}, \text{support} = \text{sup} \) is significant if \( \text{statistic} \) lies outside the range \((\text{stat}_L, \text{stat}_R)\).

The confidence intervals are learned using a non-parametric methodology where we use randomization to compute expected ranges for the function of interest. The actual churn rate in the segment is then compared to this distribution of values to determine whether the rule discovered is considered significant.

### 2.2 Phase 2 – Prospective Analysis: Learning Alerts

We model our predictive model on a recent study (Neslin et al. 2006) that summarizes the results of a churn tournament.

In which the goal was to predict the accuracy of churn models. Logistic regression was the most commonly used technique (by 45% of the submissions) and we therefore use it in our approach as well. There are three additional aspects of the prediction methodology that we directly follow based on the churn tournament setup (Neslin et al. 2006). These have to do with the treatment of time, evaluation metrics relevant for churn, and evaluation datasets.

**Time:** Neslin et al. (2006) use a three month period in which predictors are derived, followed by predicting churn in the fifth month. There is an explicit one month gap in which no transactional data is used in the model. This reflects the real-world setting where a churn management program can take some time to implement. Hence if a churn alert is derived from a recent three month window it will provide the firm one month to roll out any retention program aimed at preventing churn. We use a similar approach to construct each data point. If the record represents a churned customer, based on the known churn date we ignore the immediately preceding 30 day period and then build predictors based on a prior 6 month window (we tested sensitivities as well). For the non-churned customers we simply pick a random 6 month window to construct the predictors (waiting one month and then computing the dependent variable is irrelevant since the dependent variable is known to be zero as they are all non-churned customers).

**Evaluation Datasets:** The tournament used two different validation datasets (Neslin et al. 2006). The first is a sample of different customers but in the same time period as the one in which the model was trained. For each of the new customers in the validation data sets, predictors are similarly computed over the appropriate time horizon (3 months for the tournament) and the trained model is used to predict their churn in the fifth month. Hence, this tests if the model built on one set of customers has predictive power when applied to a different set of holdout customers in the same time period. The second validation data set is used to test the “shelf life” of the churn model. Here, the model is applied for a three month period in the future for new customers to test if the trained model retains its predictive accuracy in the future. In our analysis we use the first approach where the hold out is a separate sample of 30% of the customers in each segment.

**Evaluation Metrics:** The tournament study uses the lift curve obtained from the models and the Gini coefficient (area under the lift curve and above random prediction). Unlike measures of overall classification accuracy, these metrics are well-studied and particularly relevant for churn since firms can use a sorted list of customers (from most to least likely to churn) and apply profit-based decisions upon which customers to focus retention programs. We therefore use these same metrics for the analysis in this paper as well.

**Unbalanced Priors:** Evaluation metrics such as lift and Gini favor models that can effectively identify the churned customers higher up in a sorted list. With low churn priors (often in the low single digits for several firms) predicting the major class always is likely to have extremely high classification accuracy but a lift measure which is no better than random. In such cases models are often trained by
altering the class distributions in the training sample. Traditionally oversampling (repeating the minor class records several times) or undersampling (randomly sampling only a portion of the major class) are commonly used to construct training datasets in these cases. In our research we create training datasets with all the churned customers and add to this a random sample of the non-churners to bring the total prior to various experimental baselines (50%, 36% and 25% respectively).

Variables: The variables we use as predictors include behavioral antecedents, primarily past shipping volumes and service failures or failure rates for each time period. Hence for any window over which predictors are to be constructed we generate a variety of attributes capturing central tendencies, deviations, and trends in the weekly shipping volumes. When using six months of data as predictors, each of the variables - observed at a weekly level in the data e.g. a failure rate or volume - creates several features (such as overall average, standard deviation, and slope from the entire 6 month window, for instance).

III. RESULTS AND DISCUSSION

Our research goals are to provide convincing empirical evidence that (1) service quality factors, in combination with customer demographics and behaviors, do correlate with instances of customer churn (retrospective analysis), and (2) such customer information can be used to predict future customer churn (prospective analysis). The importance of solid empirical evidence as a first step towards rigorous theory building and systems design cannot be overstated. Previous research from multiple fields on customer churn has been largely atheoretical with data bases in specific sales domains and over limited time periods. Ours is the first to apply a broad range of state-of-the-art statistical methods to a rich customer dataset that includes sophisticated service quality factors. Our on-going research collaboration gives us the future potential for building both theories and systems and then studying the longitudinal impacts of these artifacts over time in an operational industrial setting.

3.1 RESULTS OF RETROSPECTIVE ANALYSIS

The churn rates are calculated for different levels of each SQI factor. This was unsupervised in that the levels were determined independently of the churn rates. Tables 1 and 2 present these results. The tables are similar in that each column shows the churn rates as the failure rate of the SQI factor is varied. The difference between the two tables is the specific SQI factors studied. Table 1 contains service issues related to shipping processes. Table 2 contains service issues related to the problem resolution processes.

For a majority of SQI factors, 80% to 90% of the customers have zero failures over the entire two year period emphasizing the overall service quality excellence recognized for FedEx in the literature (Zeithaml et al. 2008). Given the baseline churn of ~9.4% artificially set in the dataset, we highlight, in Tables 1 and 2, the tipping points computed as the first time in which the churn jumps to 12% or higher for sufficient numbers of customers (100 or more customers in the potential segment). While these numbers vary for the different SQI factors, two clear insights emerge:

- For the first category of SQI factors (Table 1), customers seem to tolerate some failures up to a certain tipping point (up to 4% failure rates do not seem to affect churn significantly for these factors). For the second category (Table 2), few cells are marked due to the baseline numbers being very low, indicating that there are on the whole very few problem resolution issues due to a high level of service quality. However a single factor, V15, did have a substantial impact, increasing churn by almost a third.

Table 1: Tipping Point Analysis: SQI Factors – I

<table>
<thead>
<tr>
<th>Failure Rate Range</th>
<th>Issues Related to Shipping – Churn Rates across Selected SQI Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V12</td>
</tr>
<tr>
<td>None</td>
<td>8.5%</td>
</tr>
<tr>
<td>0 to 2%</td>
<td>8.9%</td>
</tr>
<tr>
<td>2 to 4%</td>
<td>11.1%</td>
</tr>
<tr>
<td>4 to 6%</td>
<td>11.4%</td>
</tr>
<tr>
<td>6 to 8%</td>
<td>11.4%</td>
</tr>
<tr>
<td>8 to 10%</td>
<td>12.7%</td>
</tr>
<tr>
<td>10 to 20%</td>
<td>8.5%</td>
</tr>
<tr>
<td>20 to 30%</td>
<td>9.6%</td>
</tr>
<tr>
<td>30 to 40%</td>
<td>11.8%</td>
</tr>
<tr>
<td>40 to 50%</td>
<td>3.4%</td>
</tr>
<tr>
<td>50 to 60%</td>
<td>6.3%</td>
</tr>
<tr>
<td>60 to 70%</td>
<td>12.5%</td>
</tr>
<tr>
<td>70 to 80%</td>
<td>0.0%</td>
</tr>
<tr>
<td>80 to 90%</td>
<td>0.0%</td>
</tr>
<tr>
<td>90 to 99.99%</td>
<td>12.2%</td>
</tr>
<tr>
<td>Exactly 100%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>
### Table 2: Tipping Point Analysis: SQI Factors – II

<table>
<thead>
<tr>
<th>Failure Rate</th>
<th>V15</th>
<th>V16</th>
<th>V6</th>
<th>V7</th>
<th>V8</th>
<th>V9</th>
<th>V10</th>
<th>V11</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>9.1%</td>
<td>9.0%</td>
<td>9.5%</td>
<td>9.4%</td>
<td>9.5%</td>
<td>9.2%</td>
<td>9.4%</td>
<td>9.1%</td>
</tr>
<tr>
<td>0 to 2%</td>
<td>12.1%</td>
<td>11.3%</td>
<td>7.7%</td>
<td>8.7%</td>
<td>0.0%</td>
<td>4.3%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>2 to 4%</td>
<td>6.5%</td>
<td>10.1%</td>
<td>10.6%</td>
<td>15.5%</td>
<td>0.0%</td>
<td>14.7%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>4 to 6%</td>
<td>9.1%</td>
<td>9.2%</td>
<td>8.0%</td>
<td>9.1%</td>
<td>23.1%</td>
<td>7.8%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>6 to 8%</td>
<td>6.1%</td>
<td>13.8%</td>
<td>6.7%</td>
<td>0.0%</td>
<td>25.0%</td>
<td>6.3%</td>
<td>0.0%</td>
<td>16.7%</td>
</tr>
<tr>
<td>8 to 10%</td>
<td>8.6%</td>
<td>7.8%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>13.3%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>10 to 20%</td>
<td>8.7%</td>
<td>13.0%</td>
<td>11.6%</td>
<td>7.7%</td>
<td>5.7%</td>
<td>10.2%</td>
<td>17.6%</td>
<td>9.8%</td>
</tr>
<tr>
<td>20 to 30%</td>
<td>13.2%</td>
<td>6.8%</td>
<td>8.7%</td>
<td>0.0%</td>
<td>3.8%</td>
<td>10.7%</td>
<td>14.6%</td>
<td>7.6%</td>
</tr>
<tr>
<td>30 to 40%</td>
<td>2.8%</td>
<td>9.8%</td>
<td>10.0%</td>
<td>0.0%</td>
<td>7.3%</td>
<td>10.7%</td>
<td>7.5%</td>
<td>10.2%</td>
</tr>
<tr>
<td>40 to 50%</td>
<td>10.0%</td>
<td>10.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>7.3%</td>
<td>10.1%</td>
<td>0.0%</td>
<td>9.9%</td>
</tr>
<tr>
<td>50 to 60%</td>
<td>3.4%</td>
<td>17.2%</td>
<td>8.3%</td>
<td>0.0%</td>
<td>7.2%</td>
<td>11.5%</td>
<td>6.3%</td>
<td>10.5%</td>
</tr>
<tr>
<td>60 to 70%</td>
<td>9.1%</td>
<td>0.0%</td>
<td>33.3%</td>
<td>0.0%</td>
<td>6.5%</td>
<td>9.6%</td>
<td>3.3%</td>
<td>11.5%</td>
</tr>
<tr>
<td>70 to 80%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>6.5%</td>
<td>10.2%</td>
<td>5.9%</td>
<td>9.9%</td>
</tr>
<tr>
<td>80 to 90%</td>
<td>33.3%</td>
<td>25.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>20.0%</td>
<td>12.7%</td>
<td>11.1%</td>
<td>15.7%</td>
</tr>
<tr>
<td>90 to 99.9%</td>
<td>14.3%</td>
<td>13.0%</td>
<td>22.2%</td>
<td>0.0%</td>
<td>7.2%</td>
<td>11.6%</td>
<td>7.8%</td>
<td>12.0%</td>
</tr>
<tr>
<td>Exactly 100%</td>
<td>6.5%</td>
<td>13.6%</td>
<td>30.0%</td>
<td>0.0%</td>
<td>7.3%</td>
<td>11.0%</td>
<td>8.9%</td>
<td>12.3%</td>
</tr>
</tbody>
</table>

### Table 3: Critical Single Factors

<table>
<thead>
<tr>
<th></th>
<th>The Non-Zero Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQI</td>
<td>Churn Jump</td>
</tr>
<tr>
<td>V15</td>
<td>33.36%</td>
</tr>
<tr>
<td>V16</td>
<td>26.01%</td>
</tr>
<tr>
<td>V14</td>
<td>20.90%</td>
</tr>
</tbody>
</table>

### Table 4: Value of SQI Factors in Interactions: Summary of Rules

<table>
<thead>
<tr>
<th>Confidence Threshold</th>
<th>Lift over Baseline</th>
<th>Demographical + Behavioral</th>
<th>SQI Alone</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>59.7%</td>
<td>5,865</td>
<td>3,328</td>
<td>73,991</td>
</tr>
<tr>
<td>16%</td>
<td>70.4%</td>
<td>2,611</td>
<td>2,161</td>
<td>34,311</td>
</tr>
<tr>
<td>17%</td>
<td>81.0%</td>
<td>1,071</td>
<td>1,112</td>
<td>14,982</td>
</tr>
<tr>
<td>18%</td>
<td>91.7%</td>
<td>339</td>
<td>590</td>
<td>5,920</td>
</tr>
<tr>
<td>19%</td>
<td>102.3%</td>
<td>171</td>
<td>342</td>
<td>2,720</td>
</tr>
<tr>
<td>20%</td>
<td>113.0%</td>
<td>46</td>
<td>219</td>
<td>1,366</td>
</tr>
<tr>
<td>21%</td>
<td>123.6%</td>
<td>15</td>
<td>81</td>
<td>318</td>
</tr>
<tr>
<td>22%</td>
<td>134.3%</td>
<td>5</td>
<td>30</td>
<td>100</td>
</tr>
<tr>
<td>23%</td>
<td>144.9%</td>
<td>0</td>
<td>7</td>
<td>32</td>
</tr>
<tr>
<td>24%</td>
<td>155.6%</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>
• Tolerance to a certain tipping point, while observed in several SQI factors, is not universal. Three factors demonstrated a much higher churn going from zero to even a very small failure rate. Table 3 summarizes this effect. As perhaps expected, the top two factors in Table 3 are related to problem resolution processes. Hence, while on the whole there are very few problem resolution failures, when they do happen they seem to make a difference.

Table 4 summarizes the results from learning churn rules. At a minimum support of 0.25% each churn rule applies to at least 224 customers. The “lift over baseline” is defined in the standard way as (confidence-prior)/prior.

These results suggest that interactions do matter. Combining SQIs with other known factors seem to be useful to learn high churn rules (the seven rules where the combinations seem to matter have churn rates that are almost three times the prior). Also interesting from this summary is that as the lift requirement is higher, SQI factors tend to play a greater rule, suggesting that these are useful in finding high churn segments in the data.

3.2 RESULTS OF PROSPECTIVE ANALYSIS

We develop a logistic regression model for predicting churn that uses a variety of transactional signals and service quality signals constructed from the data over a defined time interval. Our study is based on segmenting the customer base into four categories based on transactional volume (Low/High) for two distinct products in this sector – Product_A and Product_B shipping. Given this segmentation we build four different logistic regression models, one for each segment (e.g. “Low Product_A + High Product_B”). This segmentation was in part driven by industrial priorities and domain knowledge that recognizes the importance of these different segments.

The analysis of the lift tables clearly indicates predictive power in these churn models. One segment analyzed appeared to have the best top decile lift where the top 10% of customers capture ~30% of the total churned customers. The top decile lifts for the other segments were a little lower but still demonstrate that the model is able to identify customer churn reasonably well as evidenced by the strong lifts in the top decile. Qualitatively, examining the variables in the final model we find that all models have a combination of SQI factors as well as control-chart based indicators that simply capture volume trends and drops in transactional volume. This result is consistent with the conclusions drawn earlier that combinations of service quality factors with transactional variables are significant indicators of churn.

As evidenced by the number of simple transactional variables in the final model (i.e. shipping volume drops), it does appear that these are clearly also useful in predicting churn. Many current approaches rely on such transactional drops to flag potential churn. Hence standard practices based on statistical process control approaches that monitor declining volumes are supported by these models.

Due to space considerations, the complete results of the Phase 2 study are presented in Padmanabhan et al. 2010.

3.3 CONTRIBUTIONS TO RESEARCH AND PRACTICE

Our study makes the following key contributions. First, ours is one of the first research studies in BI to demonstrate a link between a variety of service quality factors and churn in a large non-contractual service setting. While prior literature has demonstrated the importance of behavioral, transactional and customer perception indicators, none has comprehensively considered service quality factors in an empirical setting.

Second, our simultaneous consideration of two separate analyses (retrospective and prospective) provides a complete and interesting picture for academics and practitioners. The retrospective analyses use all information prior to the churn date to determine what distinguishes churn and non-churn. By doing so, we integrate all the known demographic and behavioral factors with SQI factors and show that interactions among them are particularly important when distinguishing these groups.

The predictive modeling approach instead uses information in a specific time window prior to churn, along with several transactional trend indicators commonly used in control chart models for churn. This form of analysis is motivated specifically from an operational perspective to support churn management initiatives. This analysis also shows the importance of service quality in predicting churn behavior.

Third, our analyses also highlight an interesting “tipping point” observation when considering service quality factors independently. It is observed that many service quality factors seem to have a small window of error initially within which customers seem to churn no differently from the baseline; however, once this threshold is exceeded the segments demonstrate a marked increase in churn priors. This important finding supports the development of well-defined rule-based monitors to signal alerts when key data thresholds are exceeded for a particular customer. To our knowledge we are not aware of previous work that highlights the possible importance of tipping points for churn modeling.

Fourth, while customer surveys have strongly suggested that service quality matters, it is well known that what people say and actually do might well be different. Our analyses show that among a large number of service quality factors, a few indeed do matter. This empirical confirmation is therefore an important beginning research contribution to a better understanding of churn behaviors that will lead to theory development. Given the lack of comparables with prior work, we do not have the ability yet to determine if there is a match or not between the service quality measures in surveys versus those from this empirical study.

Further, recent discussions in the IS community have lamented the lack of industry participants in the research process and as co-authors of research papers. The field of BI
is especially ripe for providing opportunities for this. Our project is one such effort that provides a balance of applied perspective as well as research contributions that can be useful to support both industry and academic interests. Specifically from an industry perspective, FedEx has identified the following specific benefits from our research collaboration analyzing the effective use of SQI customer data:

- Facilitates better decision making to improve operational processes and customer experience.
- Supports prioritization by providing strategic and actionable recommendations.
- Provides a source of ideas about potential quality improvement projects.
- Provides important feedback into the SQI goal setting process to assist in making business tradeoffs in a constrained resource environment.
- Integrates SQI based churn modeling approach into existing customer risk management approach.

- Identifies which SQI metrics impact customer retention.
- Provides fuller information at the individual customer level.

IV. BUSINESS INTELLIGENCE DASHBOARD FOR CUSTOMER CHURN

The previous empirical analysis demonstrates clearly the potential for predicting customer churn via effective use of available information to include customer demographics, transactional behaviors, and service quality factors. We can envision the design of an innovative business intelligence (BI) dashboard that monitors customer transaction activity on a near real-time basis. A prominent BI goal would be to react quickly and appropriately to information patterns that have been identified as leading to customer churn. An initial prototype design of the customer dashboard is shown in Figure 2.
Figure 2: Customer BI Dashboard Design Prototype
The dashboard design objectives are to highlight the most relevant and timely customer information to key account managers and decision-makers. A rich customer dataset includes information on demographics, transactional behaviors, and service quality factors. Research on customer churn, driven by this current study, will identify critical data patterns to be monitored near real-time. As a data pattern exceeds a threshold value, an alert is flashed on the screen with a full description of the identified pattern and potential remedial actions to be performed. For a customer churn alert, remedial actions may include retention programs such as personal contacts to identify concerns in the customer relationship and future product discounts. Our research team is in the process of designing, building, and evaluating BI customer dashboards as discussed in our future research directions in the next section.

V. RESEARCH STATUS AND FUTURE DIRECTIONS

Our results clearly show that service quality factors in conjunction with transactional variables are significant from a predictive modeling perspective. In addition a ‘tipping point analysis’ shows that for a variety of service factors there may be a tipping point above which churn rates increase significantly.

The current BI systems are functionally advanced and have the ability to leverage real-time transactional data such as information on service failures. However, despite their growing popularity BI capabilities still fall short of the “sense and respond” capabilities envisioned by IS academics. Central to such capabilities is the intelligence between transactional data and the response rules that might be in such a system. Our research presents an approach that can bridge the gap between information and operational response, thereby taking an organization one step closer to operational excellence.

An important direction of our future research will be to customize the BI dashboard to study the root causes of alert situations. We plan to integrate the operational systems/processes graph (Figure 1) into the dashboard (Figure 2) to highlight the processes and underlying software systems that relate closely to the alert. Patterns of alerts may indicate the presence of software defects in operational systems. We will explore various dashboard designs to best support the monitoring and testing of software systems during operations using design science research methods (Hevner et al. 2004).

ACKNOWLEDGMENTS

We gratefully acknowledge the support of FedEx management in this research project. Ms. Anuja Nagpal contributed to the project as a research assistant.

REFERENCES


Abstract- Currently, the rate at which Web applications are growing is very high, and organizations developing these applications are pursuing more users by attempting to increase their interest. However, in this competitive world it is very difficult to determine a user’s interest in a company’s applications; therefore companies need more focus on the usability of their applications. This paper discusses an automated tool to test the usability of Web applications. As a case study, a web-based requirements engineering management tool was tested by considering various factors including the time spent on every page-load of every webpage, the time spent on the entire website, the number of page clicks, the number of back presses, and the total number of pages visited. In addition, the usability testing tool recorded suggestions provided by the users.

I. INTRODUCTION

Today many large-scale applications from Internet companies present fundamental hurdles for some group of users and smaller sites are no better. Therefore developers need to make Web Application more understandable and straightforward to use. In order to accomplish these goals, rigorous tests need to be conducted on these applications as users interact with the system. This type of testing is known as Usability Testing.

Traditional usability testing [1] has occurred in labs where sophisticated recording gears were used to track user actions. Since this process is very expensive and takes more developer time, gradually automated testing came into evolution. No matter how valuable in-person testing is, effective automaton is able to increase the value of overall testing by extending its reach and range. Some of the improvements due to automated testing include improved efficiency, reduced testing costs, and accurate results in less time. Automated tests that are run with minimal human intervention across a vast set of web pages would be impractical to conduct in person. So these automated tests can guide and inform the software development process by providing information about the software as it is being written. This testing not only helps the creators to fix the problems in the software but also identifies the potential issues in various internal releases by assessing each release quickly and consistently.

This paper discusses a tool named ‘AUTO’ that conducts automated tests in order to identify usability problems and limitations. As a case study, AUTO tests the usability of the tool Requirements Elicitation Manager (REM 3rd version). In particular, AUTO considers various factors such as the time spent on every page-load of every webpage, the time spent on the entire Web Application, the number of page clicks, the number of scrolls and the number of back presses. In addition, this usability testing tool records suggestions from the users.

II. RELATED WORK

Research in the area of Usability Testing has been quite popular over the last few years. The discussion below describes the research conducted by several groups.

Rogers [1] discusses about the ways to perform usability testing. For example, she discusses how a web application can be tested by calculating the time spent on the page, the task and number of errors recorded, and so forth. She also presents information about experimental design, the methods in usability testing, and the role of field studies in evaluation. She also explains field studies with some good examples where an application is tested in natural home environments.

Hosseini - Khayat, Hellmann, and Maurer discuss the tool Active Story Enhanced in [2]. They mentioned that since the data collected from usability testing plays an important role in detecting usability issues in a user interface, support for capturing and presenting this data needs to be conducted in a meaningful way. Active Story Enhanced remotely collects forms of usability data and provides new visualizations of this data to aid in the detection of usability flaws.

In [3], Shewanown describes the developer perceptions of results of an empirical automated usability evaluation approach. He discusses how the automated usability evaluation approach can effectively rate the Usability
quality of the application rather than identifying the specific usability issues in it directly. He also specifically mentions about a tool, Web Tango which is based on empirical measures of usability and works better than any other guideline review tool available today.

Khayat states what type of usability data can be collected and which representations of it are useful for detecting flaws [4]. He also describes Wizard of Oz testing, which is a type of usability testing that is performed on prototypes of a system. This form of usability testing is of particular interest to agile teams because of the short time frames between releases, the difficulty and the cost of gathering of participants, and the cost of execution.

In [5], Bastien describes work conducted in the field of user testing that aims at specifying or clarifying the test procedures and at defining and developing tools to help conduct user tests. He conducted some usability testing experiments and discusses how usability testing could be effective by considering some important factors. He suggests the number of participants that should take part in a user test should be at least 49. The test procedure to be implemented is called paired-user testing. He also describes usability testing tools like Morae and Kaldi.

Scholtz and Laskowski discuss tools and techniques that can be used to better Usability Testing [6]. The approach they followed includes two types of tools: Usability Awareness tools for beginners and tools for professionals. They mainly focused on developing tools and techniques to speed evaluation (rapid), tools that reach a wider audience for usability testing (remote), and tools that have built-in features (automated).

Harty [7] describes the benefits of automated usability tests when compared to in-person tests. He explains that automated testing can help to catch many types of problems like identifying minor usability issues especially when several techniques and approaches are used in combination. He also explains that even though test automation is imperfect and limited, it can be useful in catching various problems that would trip up some of your users.

Several research groups are currently developing automated tools for usability testing on different applications. The data provided by these tools helps in making tools more understandable to the user and helps in locating bugs in the user interface. The listing below includes some of these automated tools.

- **Active Story Enhanced**: ASE [8] is an open source tool set designed to help agile teams create prototypes and review usability data in an affordable and quick way in line with agile practices.
- **M-Pathy**: M-Pathy [9] is a remote usability tool that tracks mouse-movements and clicks without installing anything on the user’s computer.
- **Morae**: Morae [10] is a complete suite of usability testing tools that includes a recorder module for capturing user interaction, observer module to display sessions and manager module to analyze the tests.
- **Webnographer**: Webnographer [11] tracks user behavior on your website or prototype. Some of the actions that it can perform include the collection of web analytics and information on clicks, time in each form item, time on page, pages visited, task completion rates, and time on task.

The list above is not an exhaustive list of Usability testing tools; rather, it is a representative sample. Even though the above tools have some useful features, they also have some disadvantages. M-pathy concentrates only on the user’s input interaction with the application such as mouse movements and clicks, while Webnographer concentrates on the events that occur on pages such as the time spent on tasks or pages and the corresponding information. Some tools like Morae have heavy installation requirements such as a high end graphic card and webcam; such tools are expensive as well. Other tools like ASE are difficult to use.

AUTO provides some features that overcome the major drawbacks of the above mentioned tools. First, it runs on any Web Application irrespective of the platform on which it has been developed. Second, AUTO handles scroll events in addition to the click events of the visitor. It also records events on the page and the corresponding information for every page-load of every page. AUTO collects feedback from the visitors. It gives the developer a detailed list of the test flows of the visitors, and the resulting usability issues.

AUTO requires the least installation accessories for it to run when compared with the other tools. Furthermore, it is absolutely free of cost. In conclusion, because AUTO provides all these features, it is a more powerful tool when compared to many other tools available.

### III. AUTO REQUIREMENTS

The requirements for “AUTO” include the following:

1. AUTO should collect the details of each visitor and records the test information based on the Visitor type. The Visitor information follows:

   - Visitor Type
   - First Name
   - Last Name
   - Job profile
   - Age
   - Phone Number
   - Email ID
2. AUTO should be capable of running on any Web Application irrespective of the platform on which the application was developed.

3. AUTO should collect both data from the tests conducted.

   - Project
   - Test Start Time
   - Page
   - Page Start Time
   - Clicks
   - Scrolls
   - Total Back Presses
   - Page End Time
   - Test End Time

4. AUTO shall collect feedback from every visitor after he/she interacted with REM Application.

5. AUTO should provide a detailed view and information about the Usability Issues resulting in the REM Application.

IV. APPROACH

AUTO monitors the way the user is using the interface of an application.

The tool runs in two ways, keeping in mind that the user using the system may be enough knowledgeable to know how to use the new application or the user may not be comfortable with new applications. Therefore, AUTO was tested on two different groups (Knowledgable users and In-Experienced users).

AUTO collects various data from the time the user runs the application. It tracks down various things such as the time spent on a single task, the time spent on a single webpage and the time as a whole, the number of page clicks, the number of page scrolling and the number of back presses. Furthermore, it records the suggestions provided by the users in feedback forms.

In our experiment, the developers initially make certain group of users to interact with a Web Application via our AUTO tool. The developers then depending upon the test results obtained from the users, assign some base values for all the items that need to be tested like Time spent on a page, time spent on the whole test.

When the new users interact with the system, values are obtained for all the tested items. AUTO analyzes the Usability issues of the Web Application comparing these base values with the test results obtained from various visitors.

If any value obtained in the usability testing shows a major difference when compared with the base value (i.e., if it is more in number than the base value), then we need to figure out what the usability issue might be on that specific page or task. This statistics information will tell us where we need to re-factor the application. If any value of the current statistics set obtained does not show a major difference when compared with the base value (i.e., if it is less than or almost equal in number to the base value), then we can assume that there is no usability issue in that page or task.

V. AUTO

AUTO Overview:

The AUTO Tool provides a user friendly interface where the developers and visitors could easily interact with the AUTO tool and implement AUTO’s functionalities without any confusion. For better explanation of the tool, we classify it into Visitor view and a Developer view. In the Visitor view, we will be discussing about the components of the tool that the visitor is going to interact. In the Developer view, we discuss about the components of the tool that the developer (administrator) is going to interact.

1. Visitor View:

   A. Home Page:

   The visitor’s interaction with the tool starts with the home page of the tool. In this tool, the initial information of the visitors are collected from him/her before he/she is allowed to interact with the Web Application. In our case study, the visitor will be interacting with the Requirements Elicitation Manager (REM).

   AUTO primarily classifies the visitors based on their Visitor Type which includes In Experienced Visitors and Knowledgable Visitors.
In-Experienced Visitor: A visitor chooses this option if he/she hasn’t interacted with any version of REM Application before and is going to interact with REM Application for the first time.

Knowledgable Visitor: A visitor chooses this option if he/she already interacted with previous versions of REM Application and is going to interact with its later version again.

So once the details are given by the visitor the tool re-directs him/her to interact with the REM Application via a user friendly Web browser interface.

B. Web Browser:

i) Basic Web Browser Interface:

This Web browser Interface is a customized browser which works at the same level of efficiency as any regular Web browser such as Internet Explorer or Mozilla Firefox. The visitor is provided with some basic functionalities which includes back and forth navigation of the web pages and the visitor could end the interactive session whenever they want.

ii) REM Application loaded in Web Browser Interface.

The visitor will be re-directed to this web browser interface once he/she provides his/her information in the Home-Page. The Web browser interface will have REM Application loaded in it when the visitor interacts with it.

So the visitor begins interacting with the REM Application by logging into it and performs necessary operations according to his/her interest by navigating through different Web pages. The visitor’s interaction with the REM Application is recorded in a step-by-step fashion and the required data is collected dynamically. The data that is collected from the visitor’s interaction with the REM Application include:

- Project
- Test Start Time
- Page
- Page Start Time
- Clicks
- Scrolls
- Total Back Presses
- Page End Time
- Test End Time

Finally, once the visitor completes interacting with the REM Application, he/she can terminate the session. The visitor will then be re-directed to a Feedback Page.

C. Feedback Form:

The Feedback Page contains questions like:

- How do you rate the quality of the Application?
- Do you think the application is interactive enough?
- Do you find the interface rich and attractive?
- Do you think the application is time consuming?
- Do you think that the interface was difficult to understand?
- Do you think the application can be improved?

Once the visitor has provided his/her feedback regarding REM Application, he/she shall submit the results and AUTO re-directs the visitor to the Home-Page terminating the job of the visitor with AUTO.

2. Developer view:

A. Home Page:
The developer is the administrator of the AUTO tool. The developer’s interaction with the tool also starts with the home-page of the tool. The developer is initially checked for authentication to gain access to the test results of the visitor’s interaction with REM. The authentication details include:

- Username
- Password

The tool allows the developer to access the test results if the provided authentication details are correct.

B. Administrator Page:

The developer gets redirected to the Administrator’s Page once his/her authentication details are verified. The developer is provided with two primary features, whose descriptions follow.

i. Create/Delete Developer:

The developers that have access to the tool can allow/restrict access to other developers. The developer can perform the following functions:

- **Allow access to a new developer**: The developer will need to assign a ‘Username’ and ‘Password’ for each new developer who wants to access the test results.
- **Restrict access to an existing developer**: In order to restrict access to an existing developer, the developer ‘Username’ to be denied access shall be selected from the list of developer names.

ii. Usability Issues:

The developer can view the usability issues of the REM Application as a whole in a page by page fashion based on the Visitor Type. So once the developer selects the Visitor Type, the Usability issues recorded on all the pages of the REM Application will be populated below. The Usability issues that are displayed page by page include:

- The list of visitors who took more time than expected on that page.
- The list of visitors who performed more clicks than expected on that page.
- The list of visitors who performed more scrolls than expected on that page.
- The list of visitors who had more back presses than expected on the overall test.
- The list of visitors who took more time than expected on the overall test.

The developer can also view the Individual test flows of the visitors and could also edit the testing conditions if necessary.

C. Individual User Test Flows:

The developer in order to analyze the usability issues in detail can also go ahead and view the way the visitor interacted with the REM Application in a step by step fashion. Once the developer selects his choice, the corresponding Visitor information, Visitor Test Flow and the Visitor Feedback are displayed.

i. Visitor Information:

This includes the visitor information of the selected visitor.

ii. Visitor Test Flow:

This includes the flow of the selected visitor’s interaction with the REM Application.
The information that contributes to the overall Test are:

- Project
- Test Start Time
- Test End Time
- Total Back presses

The information that contributes to every page-load of every page are:

- Page
- Page Start Time
- Page End Time
- Clicks
- Scrolls

iii. Visitor Feedback:

The feedback submitted by the selected visitor.

D. Testing Conditions:

The developer can view the testing conditions if necessary. All the testing conditions related to the project ‘REM3’ are displayed if there are any. Each testing condition includes the following:

- Project
- Visitor Type
- Page
- Page Time
- Clicks
- Scrolls
- Total back presses
- Test Time

These testing conditions are the base values set for each of the above mentioned parameters by the developer initially. The developer estimates the base values for these testing conditions from the tests conducted by a group of persons of each Visitor Type individually. The developer shall make sure that the base values should be accurate since these values are compared to the test results in order to access the Usability Issues. However, the developer can even perform operations on editing testing conditions. They include:

1. Modify/Delete Testing Condition:

The developer can Modify or Delete an existing testing condition if necessary.

2. Add a Testing Condition:

The developer can add a testing condition to the existing testing condition set if necessary.

VI. AUTO ANALYSIS

The test results generated by the tool ‘AUTO’, can be further analyzed in identifying the usability issues in the REM Application.

1. General Analysis:

In the above identified Usability Issues, we have issues shown in page by page fashion. We consider the ‘Time’ taken by the visitor on every page and the overall test for analyzing the usability Issues.

The important things that we considered here are:

- The list of visitors who took more time than expected on a particular page.
- The list of visitors who performed more clicks than expected on that page.
- The list of visitors who performed more scrolls than expected on that page.
- The list of visitors who had more back presses than expected on the overall test.
- The list of visitors who took more time than expected on the overall test.
The issues shown up here are the test results obtained from the visitor's interaction with REM Application whose values exceeded the base values for each page and the overall test in the testing conditions set. So the primary thing that we consider here, as said before, is the ‘time’ spent by the visitor on each page or the overall test. If the visitor spent more time on a page than expected, then it means that he/she (visitor) faced some problem in interacting with that page. The reason may be either he/she is not able to find the task in the page or he/she is confused in implementing the functionality of a task which took him/her more time than expected. However, both of these conditions are considered as Usability Issues.

The other things that the developers consider here are the clicks, scrolls and back presses which show how long the visitor’s interaction deviated from the expected performance.

So the developer can concentrate more on the pages which show at-least two visitors listed per page. Now the developer can decide if the application’s code related to that specific page can be re-factored or not, hence making it more usable to the visitor.

ii. **Critical Analysis**

No doubt the general analysis of the usability issues help the developer in identifying the page that had issues and the visitors who faced the usability Issues, but it would be more helpful for the developer if he/she could further analyze and identify the information about tasks which caused usability issues to some extent. The developer in order to identify the task which caused Usability Issues needs to switch to the Individual Users Test Flow section where he will be able to find more information relevant to this current context.

### iii. Graph Analysis

The results obtained from the AUTO tool regarding the REM Application as said before, include the pages on which the usability issues may exist. We assume that there might be a usability issue on a page if at least two visitors of the specific knowledge type took more time than expected on a Page. In this context, in assessing the usability issues on REM Application, the pages that were having issues were:

1. **Glance.php**
2. **Add_requirements.php**

**Glance.php** : This page refers to the Home Page displayed once the visitor logs into the REM Application.

**Add_requirements.php** : This page refers to the requirements page where a visitor can add requirements to the project.
The visitors had usability issues on the above two pages. The following graphs show the time spent by some group of visitors than expected on the above mentioned pages:

i. **In-Experienced Visitors**:

- X-axis denotes Glance.php and Add_requirements.php numbered as 1 and 2.
- Y-axis denotes the time spent by In-Experienced visitor on each page in seconds.

The graph in the In-Experienced visitors (section i) shows the visitors who spent more time than expected on the pages Glance.php and Add_requirements.php where the actual expected times for those pages are 10 and 55 respectively.

ii. **Knowledgable Visitors**:

- X-axis denotes Glance.php and Add_requirements.php numbered as 1 and 2.
- Y-axis denotes the time spent by Knowledgable visitor on each page in seconds.

The graph in the Knowledgable visitors (section ii) shows the visitors who spent more time than expected on the pages Glance.php and Add_requirements.php where the actual expected times for those pages are 7 and 35 respectively.

So depending upon all these results obtained from different analysis and feedback forms, now the developer will be in an easy position to narrow down the tasks and web pages that lead to usability. Hence he/she could work on coding the respective portions and making them more usable to the visitors.

VII. STRENGTHS AND LIMITATIONS OF ‘AUTO’

*Strengths:* ‘AUTO’ possesses some unique strengths when compared to other usability testing tools. It runs on any Web Application irrespective of the platform on which it has been developed. It generates test results based on the Visitor Type. It records the entire test flow of the visitor from the beginning to the end of a session. It collects data dynamically from the tests conducted for every page-load of all pages.

*Limitations:* The lack of a video monitoring feature is probably a weakness of the AUTO tool. This feature would have provided more evidence to the developer about how a visitor interacted with the REM Application. In AUTO, we attempt to provide the developer a clear view of what happens in a step by step fashion. However, a video monitoring feature would have been more convenient for a developer to track the usability issues quickly. The lack of support for collecting audio and video data from the Web browser interface and the lack of support for touch based interactions are limitations of AUTO.

VIII. FUTURE WORK

Further work on AUTO involves embedding the video monitoring feature and extending its support to touch-based interactions. An evaluation can also be done to compare the efficacy of audio and video data versus the data collected by AUTO in detecting usability flaws. Research is also in progress which involves test-driven development of user interfaces, in which AUTO plays an important role.
IX. REFERENCES:


Abstract—Test driven development (TDD), also known as test-first development, is not simply a testing technique but a development approach where the unit tests are written before functional code in small and rapid iterations. Research reports on the effectiveness of TDD in different phases of software development such as design, cost estimation, execution-based testing, and maintenance. This paper discusses a case study of a web-based routing application where both a test-first approach and test-last techniques were applied and compared for development time and code quality. Apart from the differences in metrics when using the two approaches, the paper includes a discussion of the positive personal experiences of the developer when using TDD.

Keywords-component; Agile Methods; Test-Driven Development; Test-First Development; Verification

I. INTRODUCTION

Developing clean, flexible code on time and that works is the minimum standard of professional behavior for every software developer. Most of the time developers compromise quality for time. Test-driven development (TDD) guides developers to reach professional standards [17]. TDD is a combination of test-first development and refactoring. In short, test-first development is an engineering technique where the production code is written after unit tests with the caveat that only the amount of code needed to pass a single unit test should be written. Refactoring is a technique where the design of the code is improved without affecting its schema or purpose.

The steps in TDD follow.
1. Add a unit test case with just enough code to fail the test.
2. Next run the test suite to ensure that the new test does in fact fail.
3. Update the functional code to make it pass the new tests.
4. Run the tests again. If they fail, update functional code and retest.
5. Now look for and remove duplication; clean up the test code.

Note that the final step is the refactoring step

In summary, TDD creates automated unit tests that define code requirements before writing the code itself. Kent Beck, the creator of TDD, suggests the principle “Fake it till you make it” in his book Test-Driven Development by Example. He says that focusing on writing only the code necessary to pass tests has the effect of generating clean, working code.

“TDD produces clean code” means that TDD makes it easy for developers to modify and improve the code without breaking it, that is, introducing new errors. When all the test cases pass, the program is complete, the developers have code that works, and later this code can be sent to Software Quality Assurance for further verification.

Test-driven Development does not replace traditional testing instead it defines a proven way to ensure effective testing. [14] Note that the number of passed tests in TDD gives an estimate about the project status helping the developers to achieve clean, flexible, code that works on time. TDD also helps to improve the design process by demanding programmers give thought to design [16]. Furthermore, TDD both decreases the number of times the developer needs to perform debugging tasks and increases the test coverage [19]. These activities result in higher quality code at the rate of little extra effort in writing and maintaining the unit tests.

This paper is organized as follows. Section 2 discusses the related work. Section 3 explains our experiment setup and details. Finally, section 4 presents the results.

II. BACKGROUND & RELATED WORK

Bhat and Nachiappan [19] describe two industrial case studies they conducted at Microsoft to evaluate the efficiency of test-driven development. Case study A was performed by the Windows networking team. The team consisted of 6 developers with high domain, language and manager expertise working in the C/C++ language. Case study B was performed on the MSN (Microsoft Network) division team with 8 developers who had medium domain, language and manager expertise and who were programming in C/C++.

The researchers compared the impacts of TDD and non-TDD development processes, on quality and overall development time. Teams using TDD observed significant improvement in quality (2.6-4.2 times) when compared to non-TDD developed code. (Here quality was estimated as defects/LOC). However, teams using TDD increased the overall development time by 15-35%. The researchers also noticed that the tests served as automatic documentation when the code was maintained or used. Very similar outcomes were observed in [6]. In particular, the results proved better performance predictability and more time
Müller [10] developed a new metric Assignment Controllability (AC) to evaluate the effect of TDD on program code. In particular, AC(m) is defined as the number of controllable assignments in method m divided by the number of all assignments in method m. Controllability is a dataflow problem. The author compared five TDD software systems with three open source software systems. The results identified that the number of controllable assignments in a method is higher when the system is developed using TDD.

Erdogmus, Morisio, and Torchiano [7] used undergraduate students as their subjects to compare TDD with iterative test-last development. The results of the experiment showed 50% more productivity in the code developed using the TDD approach. The TDD group was also able to improve the design quality, and there was an observed increase in the programmers’ confidence level. However, there was no significant difference in the overall quality of the final products.

Kaufmann and Janzen [15] also conducted an experiment with undergraduate computer science students comparing TDD with the test-last approach. The results of the experiment showed 50% more productivity in the code developed using the TDD approach. The TDD group was also able to improve the design quality, and there was an observed increase in the programmers’ confidence level. However, there was no difference observed in the code complexities between the two groups.

Steinberg [5] discusses an experiment he conducted to study the effect of unit test on entry point, coupling and cohesion in a Java programming course. He found that using TDD produced cohesive code; in addition, the coupling factor was also found to be low. Similarly, Edwards [18] made observations based on a classroom project “automated grading tool” developed by students in two classes (59 students in each class). Both classes developed the same project with one class using TDD and the other class using a traditional approach. The analysis showed that the class using TDD produced code with 45% less defects. In another experiment conducted by Lui and Chan [8], it was observed that TDD improved task estimation and process tracking. In this study, the results also showed that the teams using TDD had lower defect rates with faster defect fix capability.

Williams, Maximilien and Vouk [9] conducted a year-long experiment using professionals at IBM as test subjects. Even though no significant impact was observed in new projects using TDD, the defect rate reduced by 40% compared to the baseline produced by prior projects developed using the traditional approach.

In their experiment conducted with computer science graduate students, Müller and Hagner [11] did not find any significant difference between the test-first and the test-last approaches in terms of development time. The researchers noticed that the final reliability of the TDD program was lower, though it passed the unit tests. A positive outcome was that TDD improved the developers’ understanding of the
program, which helped the developers both to use the existing methods correctly and faster.

Pancur, et. al. [12] conducted an experiement with 38 undergraduate computer science students, and discovered no significant differences between the test-first and the test-last approaches. However, the researchers noticed TDD resulted in slightly lower code quality and code coverage. Furthermore, the subjects did not find the approach effective, and expressed difficulty in adapting to TDD. A similar result was observed by Geras, Smith, and Miller [1]. In particular, the researchers observed no productivity difference when using TDD and a traditional approach. It was noted that TDD produced more tests, and the programmers ran the test cases more frequently; moreover, the product failed at the acceptance testing phase even though it had passed all the unit tests.

Abrahamsson, Hanhineva and Jääniloja [14] presented the results of a case study on mobile Java application, where a product was developed both by a professional developer and three student developers. The results in this experiment indicate no great benefits of writing unit tests beforehand. The researchers concluded TDD was not suitable for the application domain.

III. CASE STUDY

A. Goal

The goal of this research study was to conduct an experiment to compare the effectiveness of test-driven development when using agile methods to the traditional approach of test-last development. As a case study, we observe the development and resulting performance of two versions of a web based graphical routing interface – “iCampus” using the two approaches. To evaluate the effectiveness of the development techniques, we compare the output quality and the development time factors of the two applications.

B. Test-First (vs) Test-Last

When using the test-last development approach or the traditional software development process model, a developer writes the unit tests after the production code is written. In contrast, with TDD or the test-first approach, the developer employs an iterative model where the unit tests should be written before writing the methods that perform the functionality. Figure 1 shows a graphical representation of the steps followed in test-first development, whereas Figure 2 illustrates the steps of test-last development.

When using test-first development in an agile approach, all the requirements from the client are gathered as prioritized user stories. For each user story, the developer adds a unit test with just enough code to fail the test. Then he or she runs the test suite to make sure that the test does indeed fail. If the test case passes then a new user story is selected and the cycle repeats. Else the functional code is updated until the test passes. The test suite is run again to make sure that the test passes then the tests and the code undergo refactoring. In case the test still fails the code is updated until the test passes. Then a new user story is selected and the cycle repeats until all the user stories are implemented. Then the application passes through acceptance testing and the defects observed add to the new set of user stories.

In test-last development, all the requirements are first gathered from the client. Then after planning and designing phases, the application is developed as per the requirements. Once all the requirements are implemented, unit tests are written and the test suites are run. The functional code is updated to pass the failed tests. The code is updating until all the tests pass. Then the test code and the functional code are refactored. The application goes through system and acceptance testing. A new cycle starts with new requirements from the client and/or defects observed from previous cycle.

The test-last approach focuses on development and may look faster at first but it gets harder to write tests in the end. In addition, because of a drop in the code coverage, there are more chances of uncovered bugs in the system resulting in lower quality tests. In contrast, test-first development focuses on testability of the requirements. This results in an increased focus on the code and allows planning for future code. Furthermore, improved feedback in small incremental steps provides a more accurate view of the code.
C. Metrics Used

Each approach has its own set of advantages and disadvantages. Therefore, it is the responsibility of the software engineer to analyze the project requirements and to pick the right approach for a particular project. In our experiment, we evaluate the effectiveness of the two approaches in terms of input time and output quality. These metrics can be handy to estimate the usefulness of the two approaches.

a) Time for Development

We compare the development efforts of the two software development process models in person hours.

Development Time (in person hours)
= Time to write Source Code + Time to write Unit Tests

b) Code Quality

In the literature, code quality has various definitions. In this study, we use a definition that defines code quality in terms of the functional correctness of the code. In other words, we estimate the code quality in terms of defects per lines of code. We define defect as a test case whose expected output did not match with the actual output during the system testing. In particular, the definition of code quality follows,

\[
CQ = \frac{\text{No. of defects}}{\text{LOC(Source Code + Test code)}}
\]

D. Project Description

The project is both a web based and a mobile based graphical routing application that helps users navigate through the University of Memphis campus map. More specifically, given the source and the destination buildings on the campus, the application displays the shortest route between these buildings. Furthermore, the tool estimates the time needed to traverse the route, along with suggestions as to the best doors to enter and exit the buildings.

Another feature of the tool is its assistance for handicapped individuals. In particular, the user may request the shortest wheelchair access route between two locations. A third feature of the tool is keyword search. In particular, when a user enters a search keyword such as computer labs, fee, library, food court the application returns all the associated locations and details specific to that entry. A fourth feature is the display of routes based on a student’s class schedule. The application is compatible to all web browsers. Note that the overall goal of the project is to launch the product as a real product for the University of Memphis faculty, staff, and students. The product is set to go live in August 2011.

E. Experiment Design

We developed the application described above in the three steps. In the first step, we developed the application as two projects with three iterations each. The first project iCampus-1 was developed in Java using an agile process that included test-first development. The second project iCampus-2 was developed in Flex Actionscripting following the Waterfall Model, which employs the test-last paradigm. Then in the second step, we compared the two projects at the end of the third iteration of each and selected the best approach with respect to programming language. The decision was based on which project had better results as per the input time and output quality. Finally, in the third step, we continued to develop the project using the Java application.

The following discussion briefly explains the first step to compare the two software development process models while developing iCampus for 3 iterations.

a) EXPERIEMENT 1: iCAMPUS -1

iCampus-1 was developed following the test-first approach using JavaScript and, J2EE. The unit tests were developed in JUnit. The programmer involved in the experiment had two years of experience in Java/J2EE programming and her technical and language expertise were high. Table I provides an overview of experiment 1. Table II shows the experiment results of the iCampus-1 after 3 iterations that implemented 10 requirements, henceforth we name it as iCampus-1.1.
TABLE I. iCAMPUS-1 EXPERIMENT SETUP

<table>
<thead>
<tr>
<th>Experiment 1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmer Experience</td>
<td>2 Yrs</td>
<td></td>
</tr>
<tr>
<td>Technical Expertise (Low, Medium, High)</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>Language Expertise (Low, Medium, High)</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>Programming Language</td>
<td>Java/J2EE</td>
<td></td>
</tr>
<tr>
<td>Testing Tool</td>
<td>JUnit</td>
<td></td>
</tr>
<tr>
<td>Software Development Process</td>
<td>Test-Driven Development</td>
<td></td>
</tr>
</tbody>
</table>

TABLE II. iCAMPUS-1.1 EXPERIMENT OBSERVATIONS

| Project Size (Similar complex requirements) | 10 |
| Source Code LOC                             | 1518 |
| Test Code LOC                               | 1105 |
| System Test                                 | 39  |
| System Tests Failed                         | 4   |
| Defect Density (Defects per KLOC)           | 1.5 |
| Development Time (in person hrs)            | 151 |

b) EXPERIEMNT 2: iCAMPUS-2

The second project of the iCampus application was developed in Flex Action scripting, and the unit tests were developed in FlexUnit. For this project, the Waterfall Model (Test-Last Approach) was followed. The developer involved in experiment had one year of experience in Flex programming and her technical and language expertise were Medium. Although the expertise level of the programmer in experiment 1 is higher than in experiment 2, the comparison is balanced due to two facts: 1) agile programming demands more highly skilled programmers, and 2) the programmer in the study was new to the test-first approach. Table III provides an overview of experiment 2, while Table IV shows the experiment parameters of the iCampus-2 after 3 iterations that implemented 10 requirements; henceforth we name it as iCampus-2.1.

TABLE III. iCAMPUS-2 EXPERIMENT SETUP

<table>
<thead>
<tr>
<th>Experiment 2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Experience</td>
<td>1 Yrs</td>
<td></td>
</tr>
<tr>
<td>Technical Expertise (Low, Medium, High)</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>Language Expertise (Low, Medium, High)</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>Programming Language</td>
<td>Flex Action scripting</td>
<td></td>
</tr>
<tr>
<td>Testing Tool</td>
<td>FlexUnit</td>
<td></td>
</tr>
<tr>
<td>Software Development Process</td>
<td>Waterfall Model</td>
<td></td>
</tr>
</tbody>
</table>

TABLE IV. iCAMPUS-2.1 EXPERIMENT OBSERVATIONS

| Project Size (Similar complex requirements) | 10 |
| Source Code LOC                             | 1193 |
| Test Code LOC                               | 586 |
| System Test                                 | 39  |
| System Tests Failed                         | 6   |
| Defect Density (Defects per KLOC)           | 3.7 |
| Development Time (in person hrs)            | 140 |

IV. RESULTS

We compare the experiment observations of iCampus-1 that was developed using the Test-First approach and iCampus-2 developed using the Test-Last approach. Henceforth to keep the results relative and simple, we measure the experiment observations of iCampus-1.1 in terms of iCampus-2.1. Where R is the size of the project, $S_{LOC}$ is the source code lines of code, $T_{LOC}$ is the test code lines of code, $ST$ is the system tests, $DD$ id the defect density, $D$ is the System tests failed and $T$ the number of hours (time ) to write source code and systems tests.

TABLE V. EXPERIMENT RESULTS

<table>
<thead>
<tr>
<th>Experiment Parameters</th>
<th>iCampus-1.1</th>
<th>iCampus-2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Size (Similar complex requirements)</td>
<td>$R$</td>
<td>$R$</td>
</tr>
<tr>
<td>Source Code LOC</td>
<td>$1.3\times S_{LOC}$</td>
<td>$S_{LOC}$</td>
</tr>
<tr>
<td>Test Code LOC</td>
<td>$1.8\times T_{LOC}$</td>
<td>$T_{LOC}$</td>
</tr>
<tr>
<td>System Test</td>
<td>$ST$</td>
<td>$ST$</td>
</tr>
<tr>
<td>System Tests Failed</td>
<td>$D$</td>
<td>$1.5D$</td>
</tr>
<tr>
<td>Defect Density (Defects per KLOC)</td>
<td>$DD$</td>
<td>$2.4DD$</td>
</tr>
<tr>
<td>Development Time (in person hrs)</td>
<td>$1.07\times T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Defects per KLOC (defect density) metric of the iCampus-1.1 were 1.5 and iCampus v.2.1 was 3.7, which means that iCampus-2.1 had 2.4 times more defect density than the iCampus-1.1.

TABLE VI. iCAMPUS-1.2 EXPERIMENT OBSERVATIONS

| Project Size (Similar complex requirements) | 20 |
| Source Code LOC                             | 3637 |
| Test Code LOC                               | 1875 |
| System Test                                 | 80  |
| System Tests Failed                         | 9   |
| Development Time (in person hrs)            | 294 |
The results show that the test-first development approach showed a significant rise in quality for little extra effort to write and maintain the unit tests. Henceforth we decided to select test-first approach to complete the project and so we continued to develop the iCampus-1. Table VI shows the experiment parameters of the iCampus-1 at the end of the project. Note that the entire project included 6 iterations to implement 20 requirements. Henceforth, we name it as iCampus-1.2.

V. PERSONAL EXPERIENCES OF THE DEVELOPER

Initially, when the programmer was introduced to the new approach of test-first development, she did not find any significant benefits. She actually felt that the process was more tedious and slower than the traditional approach of test-last development.

However, after the completion of the project, the programmer believed the test-first approach to be more effective. She observed that the new approach forced simplicity and reduced redundancy in the code, because it was mandatory for the developer to write a test case that failed before writing the production code and to pass a test case before writing the next test case. In addition, by writing the test cases in code first, she was inadvertently bound to follow good design principles, and she also noticed an increase in her confidence as the tests passed. The natural development of the unit tests not only formed a good set of automated test cases, but also made refactoring more feasible.

VI. THREATS TO VALIDATION

Even though a real-time project was used in the case study, the project size was small and the results were based on a small number of developers. Another threat to the validation process was that the subject was aware of the experiment, and hence there are chances that the subject’s performance could have been biased.

VII. FUTURE WORK & CONCLUSION

Typically we can say that TDD approach showed a significant rise in quality for little extra effort to write and maintain the unit tests.

The experiment should be repeated on larger projects and in different environments. In addition, future experiments could include a few more metrics (e.g., the test-coverage metric) for accurate evaluation.

ACKNOWLEDGMENT

The authors would like to thank University of Memphis, Information Technology Division (ITD) for giving us the opportunity to work on the project.

REFERENCES

Complexity Measures for Implementing Quality Gates

Rick Prendergast
FedEx Services

Chen Zhang, Thomas Meservy, E. Ted Lee, Jasbir Dhaliwal
University of Memphis

Introduction

Developing high quality software is often a complex activity which requires a careful orchestration of numerous technologies, processes and individuals. Assuring quality in such an environment is a non-trivial task. Over the years a number of different approaches have emerged to increase software quality. In our previous research, we suggested that quality gates could be used as a mechanism to better control the quality of production processes throughout the software development lifecycle.

A basic premise of quality gates is that quality will be evaluated periodically throughout the software production process. Quality gates, which builds on the idea of stage-gate systems (Cooper 1990), act as control checkpoints between phases for the artifact that was produced in the previous phase. Each quality gate results in the certification that the artifact is sufficiently complete to continue to the next phase, which is often termed exit criteria. In addition, a quality gate may also be required to meet specific entry criteria. The emphasis of quality gates is on predefined quality criteria that regulates the continuance of a production process.

While the concept of quality gates is prevalent in other production processes, only minimal research on quality gates in software development exists. Previously, we developed a framework that applies the concept of quality gates at different levels of the software production process including quality gates at the heartbeat level (which are low level metrics derived from the software that is being developed) and strategic level quality gates (which are higher level quality gates that may span multiple systems, projects, releases, etc.) (Ambartsoumian et al. 2011). These quality gates can enable management to efficiently deploy resources through objective assessments of quality and also enable them to make accurate risk-based decisions.

Assessment at each quality gate is based on a number of pre-determined quality measures/metrics. Obviously criteria will vary for quality gates at different phases in the project as well as at different system levels (code level, system, release). However, we suggest that even for a single quality gate at the same point of the lifecycle and at the same level, criteria may vary based on the complexity of the artifact. That is, depending on the level of complexity, these metrics can be used to guide the selection of specific quality gate criteria.

In this paper we focus on complexity metrics that can be used in conjunction with quality gates at different levels of the quality gate framework. First, we briefly summarize our
previous work related to quality gates and then review relevant literature related to project complexity and software complexity. Next, based on the literature, we introduce a complexity framework that can provide guidance when developing and characterizing complexity metrics related to quality gates. We then present the results of a case study conducted at FedEx including a discussion of an initiative to develop complexity metrics that can be derived by using automated tools. We finish by suggesting possible future directions of research related to complexity metrics in quality gates.

**Literature Review on Quality Gates and Complexity in Software and Project**

**Quality Gates**

Quality gates provide a means of evaluating the quality of a software artifact during the production process. Numerous benefits are identified in the literature related to quality gates including the ability to assess quality (Younack 2010; Schneider 2004), decreased project risk (Charvat 2010), reduced development time (Charvat 2010), enhanced project communication (Charvat 2010; Younack 2010), focus on quality (Charvat 2010), better planning and control (Schneider 2004; Valeri and Rozenfeld 2004), and shared responsibilities (Younack 2010). Obviously there are challenges to implementing quality gates within organizations including determining how to structure and define quality gate criteria, how quality gates are related to each other, who is responsible for monitoring each gate, how the quality gates are used as a decision making mechanism, etc. (Ambartsoumian et al. 2011). Quality gates in software development have been studied by a few researchers (Karlstrom and Runeson 2006; Wallin 2002; Nguyen 2006; Tarrani and Zarate 2010), however, most of these studies only generally discuss the integration of stage-gate models and the software development life-cycle at a high level with only limited details on how quality gates are actually implemented.

Thus, previously (Ambartsoumian et al. 2011), we developed a model of how quality gates could be applied throughout the enterprise IT production process including how quality gates could be applied across projects, systems, releases, etc.

Like most quality gates research in software development, the framework we developed captures the idea that quality gates can be implemented as gates between software development phases (e.g., scope, build, and test). The framework as depicted in Figure 1 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).
Figure 1: A Framework for Quality Gates for Enterprise IT Production Process

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 focus 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.

**Software Complexity**

Software complexity has been investigated extensively by computer science and information systems researchers since the 1980s. The primary objective of developing measures of software complexity is to facilitate cost estimation, resource allocation, and program evaluation (Kearney et al. 1986). A large number of metrics have been proposed and evaluated to measure software complexity. Some of the most frequently cited measures include:

- **Size metrics**
  - Such as number of statements (Weyuker 1988) and lines of code.
  - These metrics tend to penalize high-level programming languages.

- **Algorithm complexity metrics.**
  - For example, McCabe’s cyclomatic complexity measures the number of linearly independent paths in a program’s algorithm.
  - It is computed based on the number of edges, the number of nodes, and the number of connected components in a program flow graph (McCabe 1976).

- **Halstead programming effort (Halstead 1977) metrics,**
  - These are calculated based on the number of distinct operators, number of distinct operands, total number of operators, and total number of operands in the source code.
  - These metrics are easy to evaluate but have some limitations. For example, they are based on the source code and hence are dependent on the
Function point metrics
- These were first developed by IBM in the 1970s.
- Functions points are the weighted total of five aspects of a software product – inputs, outputs, logical files, interfaces, and inquiries (e.g., Albrecht and Gaffney 1983).
- Function point metrics are independent of the programming language. A number of variant forms of function point metrics have been developed and adopted since the introduction of function point metrics.
- These metrics have been adopted for industry benchmarking, productivity improvement, and new application estimation purposes. However, they are costly to obtain because the assessment of function point metrics requires subjective evaluations of professionals.

Design complexity
- This metrics consists of three dimensions of complexity: domain, data, and decision complexity (Harter et al. 2000, Jones 1996).
- Domain complexity measures the degree of difficulty in the algorithms and calculations in the software product. Data complexity measures the degree of difficulty in developing the systems due to complex data structures and relationships. Decision complexity is associated with the degree of difficulty in the decision paths and decision structures.
- All three complexity measures are dependent on subjective assessment.

Class design complexity of object-oriented systems
- Chidamber and Kemerer (1994) develop and evaluate six object-oriented design metrics such as weighted methods per class, depth of inheritance tree, and number of children.
- They also provide suggestions to managers on how to use these metrics to improve the design and development of object-oriented systems.

Software complexity of object-oriented systems
- This metrics consists of complexity measures at four levels – variable, method, object, and system (Tegarden et al. 1995). The cohesion and the coupling aspects of the system at each of these levels are measured.
- Variable level complexity is related to the definition and use of variables in the system.
- Method level complexity is associated with the definition and use of methods in the system.
- Object level complexity is the aggregated measure based on variable complexity, method complexity, and inheritance structure.
- System level complexity represents high level complexity of system size and organization.
Project Complexity

In the project complexity literature, project complexity is defined in terms of the number of varied elements and the interdependency between those elements (Baccarini, 1996). Baccarini (1996) also proposes two types of project complexity: organizational complexity and technological complexity. According to Baccarini, organizational complexity refers to the number of, and relationships between, hierarchical levels, formal organizational units, and specializations while technological complexity refers to the number of, and relationships between, inputs, outputs, tasks, and technologies. As another dimension of project complexity, Turner and Cochrane [1993] propose uncertainty, which is the extent to which the project goals and means are ill defined and are subject to future changes. Based on these two dimensions, Williams [1999] identifies two distinct aspects of project complexity: structural complexity (the underlying structure of the project) and uncertainty-based complexity (the uncertain or changing nature of the project). He argues that uncertainty adds to the complexity of a project so it can be viewed as a constituent dimension of project complexity. Based on the aforementioned dimensions/aspects of project complexity and their experience in working with a number of CIOs, Xia and Lee (2003) propose a conceptual framework with two dimensions (as shown in Figure 2) for information systems development project (ISDP) complexity in their study to conceptualize and develop valid measurements of the key dimensions of ISDP complexity. The two dimensions used in their framework are Organizational vs. Technological and Structural vs. Dynamic and there are four components of ISDP complexity in the framework: structural organizational complexity, structural IT complexity, dynamic organizational complexity, and dynamic IT complexity:

<table>
<thead>
<tr>
<th>Structural</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organizational</td>
<td>Technical</td>
</tr>
<tr>
<td>Structural Organizational Complexity (SORG)</td>
<td>Dynamic Organizational Complexity (DORG)</td>
</tr>
<tr>
<td>Structural IT Complexity (SIT)</td>
<td>Dynamic IT Complexity (DIT)</td>
</tr>
</tbody>
</table>

Figure 2. A conceptual Framework for ISDP Complexity (Xia 2003)

1) *Structural organizational* reflects the nature and strength of the relationships among project elements in the organizational environment, including project resources, support from top management and users, project staffing, and skill proficiency levels of project personnel;

2) *Structural IT* captures the complexity of the relationships among the IT elements, reflecting the diversity of user units, software environments, nature of data processing, variety of technology platforms, need for integration, and diversity of external vendors and contractors;
3) *Dynamic organizational* captures the pattern and rate of change in ISDP organizational environments, including changes in user information needs, business processes, and organizational structures; it also reflects the dynamic nature of the project’s effect on the organizational environment; and

4) *Dynamic IT* measures the pattern and rate of changes in the ISDP’s IT environment, including changes in IT infrastructure, architecture, and software development tools.

Another framework is proposed by Bosch-Rekveldt et al (2010) to better characterize and understand project complexity in a large engineering project using TOE (Technical, Organizational, and Environmental). Even if this framework is not specifically for ISDP, it may be adapted for ISDP. As in the framework of Xia and Lee (2003), this framework addresses the technical complexity and organizational complexity, but with one more distinct aspect of project complexity, environmental complexity. Based on literature review, practice, and two case studies, the elements of complexity are clustered into a framework of technical, complexity, organizational, and environmental complexity, called the TOE framework. On a lower level, the authors further group the elements into subcategories as in Table 1 (the 50 elements appear in an Appendix):

<table>
<thead>
<tr>
<th>Technical</th>
<th>Organizational</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goals</td>
<td>Size</td>
<td>Stakeholders</td>
</tr>
<tr>
<td>Scope</td>
<td>Resources</td>
<td>Location</td>
</tr>
<tr>
<td>Tasks</td>
<td>Project team</td>
<td>Market conditions</td>
</tr>
<tr>
<td>Experience</td>
<td>Trust</td>
<td>Risk</td>
</tr>
<tr>
<td>Risk</td>
<td>Risk</td>
<td></td>
</tr>
</tbody>
</table>

The subcategories for technical and organizational complexity can be well adapted for ISDP, but some of the subcategories for environmental complexity may not be suitable for ISDP, market conditions, for example.

Unlike the framework of Xia and Lee (2003), this framework does not explicitly cover the dynamic nature of project complexity even if the authors contend that many of dynamic elements of project complexity are taken into consideration and included in this framework. The authors argue that their work primarily focuses on elements contributing complexity that can be assessed before project execution is started and the intended use of this framework in the early phases of a project.


**Framework for Complexity in Quality Gates**

We suggest that complexity dimensions from the literature can be used to create a framework that can be used in conjunction with Quality Gates to enhance the quality within an organization. Building on the literature reviewed about we propose an extension to the previously suggested complexity frameworks. We then suggest potential metrics that could be used as a basis for implementing complexity quality gates throughout the enterprise.

Building on the work of Baccarini (1996) and Xia and Lee (2003) we adopt the technical and organizational categories as one dimension of our framework. Technical and organizational aspects of complexity are widely accepted in the general project management literature and the software risk literature. IT production processes not only include technological components such as hardware, software, and infrastructure, but also involve organizational components such as people, teams, and processes.

Most software systems include more than just a single module or system, but consist of numerous interconnected modules or systems. Measures can be obtained to characterize the complexity of individual modules or to characterize the complexity of dependencies between modules/systems. Thus, the proposed framework further decomposes the technical complexity category into intra-system complexity and inter-system complexity.

For the other dimension of our framework we adopt the structural and dynamic categories suggested by Xia and Lee (2003). Dynamic complexity for technical artifacts may include not only the pattern and rate of change as suggested by Xia and Lee but also may include aspects that can only be captured when the code is executed. Table 2 displays the proposed framework and provides a definition of each of the framework cells.
Table 2. Complexity Framework

<table>
<thead>
<tr>
<th></th>
<th>Structural</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Technical – Intra System</strong></td>
<td><em>Structural Intra-System technical Complexity</em></td>
<td><em>Dynamic Intra-Systems technical Complexity</em></td>
</tr>
<tr>
<td></td>
<td>Metrics that capture the complexity of a single technical system/application/artifact</td>
<td>Metrics that capture the pattern and rate of change of a single technical system/application/artifact from one point in time to another</td>
</tr>
<tr>
<td><strong>Technical – Inter System</strong></td>
<td><em>Structural Inter-System technical Complexity</em></td>
<td><em>Dynamic Inter-System technical Complexity</em></td>
</tr>
<tr>
<td></td>
<td>Metrics that capture the complexity because of dependencies across technical systems/applications/artifacts</td>
<td>Metrics that capture the pattern and rate of change of the interdependencies across technical systems/applications/artifacts from one point in time to another</td>
</tr>
<tr>
<td><strong>Organizational</strong></td>
<td>Structural Organizational</td>
<td>Dynamic Organizational</td>
</tr>
<tr>
<td></td>
<td>Metrics that reflect the nature and strength of the relationships among project resource, project personnel, and stakeholders</td>
<td>Metrics reflecting changes in the business processes, organizational structures, etc.</td>
</tr>
</tbody>
</table>

**Case Study**

In this section we present a case study of a FedEx initiative that focused on complexity metrics that could be automatically derived from software artifacts. These complexity metrics can either serve as quality gate criteria or they could be used in conjunction with other quality gate criteria.

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 described earlier in this paper, we developed a theoretical “quality gates” framework where pass or fail outcomes of thresholds are assessed at key points in the development life cycle at both the heartbeat level and the strategic level. In the case study, we focus on the initial implementation of heartbeat-level quality gates. Specifically, the initial implementation involves an automated quality gates system, which consists of cloud based services designed to aggregate information provided by automated quality assessment tools across the enterprise. In addition, such information is stored for three years to provide trend reports, dashboards, and detailed drilldowns to a portfolio of systems.

The first step of implementing heart-beat level quality gates focuses on identifying and implementing automated software tools to assess complexity of work products throughout the enterprise. Perspectives of the usefulness of complexity measurements vary between disciplines and by level of responsibility. From the senior management perspective, complexity measurements can serve as an input into the estimation of technical quality index, maintainability, technical debt, and other software quality metrics, which have the potential to increase software quality transparency and to facilitate the management of in-house, outsourced, and co-sourced software development efforts. From the middle management perspective, objective complexity measurements help managers assess new and existing applications to allocate resources more effectively.

There exist a number of proprietary and open source automated tools that can be applied to assess complexity metrics within particular technical disciplines and to aggregate metrics across disciplines. Radulescu (2009) provides a summary of the open source automated tools that can be used to measure software complexity. The following table presents some tools and metrics that are currently being used to various degrees to assess complexity of work products.
<table>
<thead>
<tr>
<th>Tool</th>
<th>Complexity Metric</th>
<th>Description of Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMD <a href="http://pmd.sourceforge.net/">http://pmd.sourceforge.net/</a></td>
<td>Cyclomatic Complexity</td>
<td>1-4 is low complexity, 5-7 indicates moderate complexity, 8-10 is high complexity, and 11+ is very high complexity</td>
</tr>
<tr>
<td></td>
<td>NPath complexity</td>
<td>the number of acyclic execution paths through that method. A value above the threshold of 200 generally suggests that measures should be taken to reduce complexity.</td>
</tr>
<tr>
<td></td>
<td>NCSS (Non Commenting Source Statements)</td>
<td>the number of lines of code for a given constructor, method, type and class. NCSS ignores comments, counts actual statements and counts split lines of code as one.</td>
</tr>
<tr>
<td>CheckStyle <a href="http://checkstyle.sourceforge.net/">http://checkstyle.sourceforge.net/</a></td>
<td>BooleanExpressionComplexity</td>
<td>Restrict the number of number of &amp;&amp;,</td>
</tr>
<tr>
<td></td>
<td>ClassDataAbstractionCoupling</td>
<td>This metric measures the number of instantiations of other classes within the given class.</td>
</tr>
<tr>
<td></td>
<td>ClassFanOutComplexity</td>
<td>The number of other classes a given class relies on.</td>
</tr>
<tr>
<td></td>
<td>Number of Attributes</td>
<td>The number of distinct state variables in a class serves as one measure of its complexity.</td>
</tr>
<tr>
<td></td>
<td>Number of Remote Methods</td>
<td>For each class, a remote method call is defined as an invocation of a method that is not declared in the class itself, a class or interface that the class extends or implements, or a class or method that extends the class. The value is the count of all the remote method calls in all of the methods and constructors of the class.</td>
</tr>
</tbody>
</table>
Using automated tools to assess complexity involves minimal human intervention and subjective judgment. However, one challenge in applying software complexity metrics is to establish an acceptable range of values for the metrics based on a number of factors such as project requirements, industry best practices, and organizational context.

These metrics that can be automatically calculated are largely focused on the structural intra-system technical category of the complexity framework. Because this quadrant is limited to a single module/system and because accessible tools exist that can calculate established metrics based on the static code, this quadrant serves as a reasonable starting point when applying the complexity framework within a large organization.

It is important to note that caution must be exercised when applying complexity metrics as inappropriate use of these metrics may have negative effects by rewarding poor programming practices (Kearney 1986). These metrics alone may not lead to a reduction of software or maintenance costs. Like other software metrics, complexity assessment “must adhere to the science of measurement if it is to gain wide-spread acceptance and validity” (Fenton 1994, pg. 205). Thus we strongly recommend that these metrics be used only after empirical validation and in conjunction with other quality criteria.

The evaluation of the structural, intra-system technical category of the framework, if assessed and acted upon properly, will likely result in software quality improvement. However, we suggest that other categories of the framework not be ignored. For example, structural complexity metrics can be assessed on a regular basis to help managers identify trends and patterns and implement proper interventions to further improve software quality. In addition, as the implementation of the complexity framework moves to the inter-system level and the organizational level, careful planning and significant efforts may be required to identify and involve the stakeholders who will make decisions or take actions based on the metrics. Specifically, the stakeholders can provide inputs to identify metrics useful to them, develop methods to assess metrics, define decision criteria, and design reporting mechanisms (Westfall 2005).

**Conclusion**

In this paper we presented a complexity framework that can be used in conjunction with the quality gate framework proposed in our earlier study. We also presented a case study involving an initiative to capture complexity metrics using automated tools. While generating metrics for the structural, intra-system technical category is a reasonable and promising beginning, we suggest that additional effort be invested in establishing and applying scientifically developed and empirically validated complexity metrics.
References


http://www.tarrani.net/LifeCycleQualityGatesZT.pdf, Feb 2010


## Appendix

TOE (Technical, Organizational, Environmental) Framework with 50 elements

<table>
<thead>
<tr>
<th>TOE Sub-ordering</th>
<th>Elements defined</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong> Goals</td>
<td>Number of goals</td>
<td>What is the number of strategic project goals?</td>
</tr>
<tr>
<td><strong>T</strong> Goals</td>
<td>Goal alignment</td>
<td>Are the project goals aligned?</td>
</tr>
<tr>
<td><strong>T</strong> Goals</td>
<td>Clarity of goals</td>
<td>Are the project goals clear amongst the project team?</td>
</tr>
<tr>
<td><strong>T</strong> Scope</td>
<td>Scope largeness</td>
<td>What is the largeness of the scope, e.g. the number of official deliverables involved in the project?</td>
</tr>
<tr>
<td><strong>T</strong> Scope</td>
<td>Uncertainties in scope</td>
<td>Are there uncertainties in the scope?</td>
</tr>
<tr>
<td><strong>T</strong> Scope</td>
<td>Quality requirements</td>
<td>Are there strict quality requirements regarding the project deliverables?</td>
</tr>
<tr>
<td><strong>T</strong> Tasks</td>
<td>Number of tasks</td>
<td>What is the number of tasks involved?</td>
</tr>
<tr>
<td><strong>T</strong> Tasks</td>
<td>Variety of tasks</td>
<td>Does the project have a variety of tasks (e.g. different types of tasks)?</td>
</tr>
<tr>
<td><strong>T</strong> Tasks</td>
<td>Dependencies between tasks</td>
<td>What is the number and nature of dependencies between the tasks?</td>
</tr>
<tr>
<td><strong>T</strong> Tasks</td>
<td>Uncertainty in methods</td>
<td>Are there uncertainties in the technical methods to be applied?</td>
</tr>
<tr>
<td><strong>T</strong> Tasks</td>
<td>Interrelations between technical Processes</td>
<td>To what extent do technical processes in this project have interrelations with existing processes?</td>
</tr>
<tr>
<td><strong>T</strong> Tasks</td>
<td>Conflicting norms and standards</td>
<td>Are there conflicting design standards and country specific norms involved in the project?</td>
</tr>
<tr>
<td><strong>T</strong> Experience</td>
<td>Newness of technology (world-wide)</td>
<td>Did the project make use of new technology, e.g. non-proven technology (technology which is new in the world, not only new to the company!)?</td>
</tr>
<tr>
<td><strong>T</strong> Experience</td>
<td>Experience with technology</td>
<td>Do the involved parties have experience with the technology involved?</td>
</tr>
<tr>
<td><strong>T</strong> Risk</td>
<td>Technical risks</td>
<td>Do you consider the project being high risk (number, probability and/or impact of) in terms of technical risks?</td>
</tr>
<tr>
<td><strong>O</strong> Size</td>
<td>Project duration</td>
<td>What is the planned duration of the project?</td>
</tr>
<tr>
<td><strong>O</strong> Size</td>
<td>Compatibility of different project management methods and tools</td>
<td>Do you expect compatibility issues regarding project management methodology or project management tools?</td>
</tr>
<tr>
<td><strong>O</strong> Size</td>
<td>Size in CAPEX</td>
<td>What is the estimated CAPEX of the project?</td>
</tr>
<tr>
<td><strong>O</strong> Size</td>
<td>Size in</td>
<td>What is the (expected) amount of engineering hours in the</td>
</tr>
<tr>
<td>TOE</td>
<td>Sub-ordering</td>
<td>Elements defined</td>
</tr>
<tr>
<td>-----</td>
<td>--------------</td>
<td>-----------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Engineering hours</td>
</tr>
<tr>
<td>O</td>
<td>Size</td>
<td>Size of project team</td>
</tr>
<tr>
<td>O</td>
<td>Size</td>
<td>Size of site area</td>
</tr>
<tr>
<td>O</td>
<td>Size</td>
<td>Number of locations</td>
</tr>
<tr>
<td>O</td>
<td>Resources</td>
<td>Resource and skills availability</td>
</tr>
<tr>
<td>O</td>
<td>Resources</td>
<td>Experience with parties involved</td>
</tr>
<tr>
<td>O</td>
<td>Resources</td>
<td>HSSE awareness</td>
</tr>
<tr>
<td>O</td>
<td>Resources</td>
<td>Interfaces between different disciplines</td>
</tr>
<tr>
<td>O</td>
<td>Resources</td>
<td>Number of financial resources</td>
</tr>
<tr>
<td>O</td>
<td>Resources</td>
<td>Contract types</td>
</tr>
<tr>
<td>O</td>
<td>Project team</td>
<td>Number of different nationalities</td>
</tr>
<tr>
<td>O</td>
<td>Project team</td>
<td>Number of different languages</td>
</tr>
<tr>
<td>O</td>
<td>Project team</td>
<td>Cooperation JV partner</td>
</tr>
<tr>
<td>O</td>
<td>Project team</td>
<td>Overlapping office hours</td>
</tr>
<tr>
<td>O</td>
<td>Trust</td>
<td>Trust in project team</td>
</tr>
<tr>
<td>O</td>
<td>Trust</td>
<td>Trust in contractor</td>
</tr>
<tr>
<td>O</td>
<td>Risk</td>
<td>Organizational risks</td>
</tr>
<tr>
<td>E</td>
<td>Stakeholders</td>
<td>Number of stakeholders</td>
</tr>
<tr>
<td>T</td>
<td>Sub-ordering</td>
<td>Elements defined</td>
</tr>
<tr>
<td>---</td>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>E</td>
<td>Stakeholders</td>
<td>Variety of stakeholders' perspectives</td>
</tr>
<tr>
<td>E</td>
<td>Stakeholders</td>
<td>Dependencies on other stakeholders</td>
</tr>
<tr>
<td>E</td>
<td>Stakeholders</td>
<td>Political influence</td>
</tr>
<tr>
<td>E</td>
<td>Stakeholders</td>
<td>Company internal support</td>
</tr>
<tr>
<td>E</td>
<td>Stakeholders</td>
<td>Required local content</td>
</tr>
<tr>
<td>E</td>
<td>Location</td>
<td>Interference with existing site</td>
</tr>
<tr>
<td>E</td>
<td>Location</td>
<td>Weather conditions</td>
</tr>
<tr>
<td>E</td>
<td>Location</td>
<td>Remoteness of location</td>
</tr>
<tr>
<td>E</td>
<td>Location</td>
<td>Experience in the country</td>
</tr>
<tr>
<td>E</td>
<td>Market conditions</td>
<td>Internal strategic pressure</td>
</tr>
<tr>
<td>E</td>
<td>Market conditions</td>
<td>Stability project environment</td>
</tr>
<tr>
<td>E</td>
<td>Market conditions</td>
<td>Level of competition</td>
</tr>
<tr>
<td>E</td>
<td>Risk</td>
<td>Risks from environment</td>
</tr>
</tbody>
</table>

\(^1\)L = based on literature data, E = based on empirical data, B = based on both literature and empirical data.
Abstract— There has been tremendous growth in new and exciting Java web applications in the past few years. Remote usability testing allows you to evaluate the most important aspect of these websites, that is, their usability by gathering information from remote users. Usability testing itself is a technique for ensuring that the users of a system can carry out intended tasks efficiently. Several different tools for usability testing have been developed in different technologies, but there is no open source tool available that works for all Java web applications. This paper describes a remote usability testing tool called JRUTT (Java Remote Usability Testing Tool) that offers the capability of usability testing with users through user behavior log files and task elicitation. With JRUTT, we perform a case study on the system Security Enhanced Metric (SET4SCRUM) to obtain objective assessments of its design.

I. INTRODUCTION

Web applications have become the most prevalent and varied form of human–computer interface. Creating a Web application allows thousands of potential users with diverse background, goals and knowledge levels, to access its information [4]. Obtaining usable Web sites is still difficult even though a Web site can easily be developed. With so many Web pages being designed and maintained, it is almost impossible to properly address usability issues without automation. For these reasons, interest in automatic support for usability evaluation of Web sites is rapidly increasing. [10]

Currently, user interactions are being captured on a large scale and to obtain meaningful evaluation. So, it is crucial that users should interact with the application in their daily environment rather than a controlled environment [2]. Also, it is impractical to have usability testing completely achieved only by evaluators who directly observe users’ behaviors. Note that there has been work for qualitative usability with audio and video capture. But, we provide a remote usability tool JRUTT, to provide quantitative data and to help discover usability issues with a combination of both remote and unmoderated testing [17].

Secure software development is a complex and challenging problem, especially in agile development methodologies that require collaborative environments. Secure software has become increasingly important in the realm of web applications, where application vulnerabilities account for 66% of attacks [16]. This phenomenon presents a growing reliance on secure software within small to medium sized companies. The tool Security Enhanced Metric Taxonomy for SCRUM, a metric based approach provides the ability to construct or to enhance a secure agile software development process most suitable for secure software engineering. With JRUTT, we perform a case study on SET4SCRUM and provide the results and conclusions derived.

This paper is organized as follows. Section 2 presents the literature review. Section 3 provides the specification of the tool including all types of requirements, and the design of the tool. Section 4 describes the tool itself and all the case study details including the results collected after performing the case study, evaluation of JRUTT and evaluation of SET4SCRUM. Section 5 presents our conclusions and ideas for future work.

II. LITERATURE REVIEW

Usability and accessibility are two ways of measuring software quality. There has been tremendous growth in new and exciting free or cheap web site usability testing tools in the past few years. A strong advantage of usability tests is they can be relatively inexpensive and easy to conduct [9]. At one extreme, usability tests can be held in a specially designed laboratory with one-way glass and systems enabling facilitators to interact with and observe the tester, recording their on-screen actions, facial expressions, and verbal feedback [6]. This approach may be more precise, but there are costs associated with hiring or owning a laboratory, recruiting testers and sending staff to facilitate and observe the testing. A simpler approach is to conduct the test at a user's desk with a facilitator simply observing and taking notes with a pen and paper. [1]

There are several ways in which automated tests can help identify problems and limitations in web based applications, where fixing them makes the software more usable and/or accessible. The work complements, rather than replaces, other human usability testing. No matter how valuable in-person testing is, effective automation is able to increase the value of overall testing by extending its reach and range [11]. Automated tests that are run with minimal human intervention across a vast set of web pages would be impractical to conduct in person. Conversely, people are capable of spotting many issues that are difficult to program a computer to detect. Therefore, we follow the combined
approach while performing the case study. Section 4 provides the details of our case study.

There are around twenty four remote usability testing tools that have replaced the usage of expensive recruitment firms, labs and massive amounts of time to create, deploy and report on usability tests. These tools are A Paper and Pencil, Concept Feedback, Chalkmark, Clickheat, ClickTale, Clixpy, Crazy Egg, Ethnio, Feng-GUI, Five Second Test, Feedback Army, Loop11, Mechanical Turk, Morae, Open Hallway, Silverback, Simple Mouse Tracking, Usabilla, UserFly, UserTesting.com, UserVue, Google Analytics, Google Website Optimizer and Website Grader [14].

These tools label themselves as usability testing tools, but they do not actually offer the capability of doing usability testing with users through task elicitation. Some of these tools are nothing more than survey tools or, Web analytics tools with new and improved visuals. Furthermore, among these tools there is only one complete open source tool “Clickheat” [8].

In “Clickheat” there are rather specific server requirements and a few other restrictions that must be understood. As with other click map tools, there is no way to ask users why they clicked where they did, or what they expected to find by clicking [5]. For large sites with massive amounts of data, there is not currently a way to download database formatted data for analysis. Clickheat does not meet the main goals of usability testing [13]. Therefore, there is no open source tool available for all Java web applications and hence we define a new remote usability testing tool for Java web applications and perform the case study following the combined approach.

III. SPECIFICATIONS, REQUIREMENTS AND DESIGN

The incremental development approach was used to develop the tool because of time constraints and also to have working software at all times. The following sections discuss the complete requirements and design analysis.

A. System Hardware Requirements

<table>
<thead>
<tr>
<th>Client Side</th>
<th>Processor</th>
<th>RAM</th>
<th>Disk Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Explorer 7.0</td>
<td>Pentium II at 500 MHz</td>
<td>64 MB</td>
<td>1 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Processor</th>
<th>RAM</th>
<th>Disk Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Tomcat 5.x</td>
<td>Pentium III at 1 GHz</td>
<td>512 MB</td>
<td>1 GB (Excluding data size)</td>
</tr>
<tr>
<td>Mysql 5.1</td>
<td>Pentium III at 1 GHz</td>
<td>512 MB</td>
<td>1 GB (Excluding data size)</td>
</tr>
</tbody>
</table>

B. System Software Requirements

- Client on Intranet: Client Software, Web Browser, Operating System (any)
- Web application Server: Apache Tomcat, Operating System (any)
- Data Base Server: Mysql
- Development End: Java, Java Bean, Servlets, HTML, DB2, OS (Windows)

C. Functional Requirements

JRUTT will provide both quantitative and qualitative data about a Java web application through task elicitation.

- It will give you both the success and failure rates of basic tasks that have to be performed on the application.
- It allows the user to input their suggestions and comments about the application.
- It records time spent on a basic tasks of the application.
- It distinguishes between the success and failure time spent on the tasks.
- It allows the application to maintain satisfaction ratings.
- It calculates the ease of use of each task.
- It recruits the users for test by intercepting the application.
- It will provide an Administrator interface to monitor all the recorded data.

D. Nonfunctional Requirements

- Usability: The font size and colors of the tool interface are visible 2.5 feet from the screen. The interface has neatly organized layout. The buttons will be displayed on all the pages so that it is easy for the user to navigate from any page to any other page.
- Reliability: JRUTT get its information from the database and the tables are critical to its functionality. Power failures and unclean shutdowns may affect the integrity of the database. Therefore measures like backing up the data are in place to prevent loss or corruption of data.
- Performance: Performance of JRUTT depends on how fast the tool is able to push the data onto the appropriate tables in the database. On a particular page, if information of the user is selectively logged, then the relevant pages are displayed quickly (i.e. the system response time ranges between 8-10 seconds). If all the user information is logged, then populating the relevant database takes time and subsequent steps are delayed. A powerful database engine can lead to increase in performance.

E. Design: N-TIER ARCHITECTURE

Simply stated, an n-tier application helps us distribute the overall functionality into various tiers or layers and is shown in figure 1:

- Presentation Layer
• Business Rules Layer
• Data Access Layer
• Database/Data Store

Each layer can be developed independently of the other provided that it adheres to the standards and communicates with the other layers as per the specifications. This is one of the biggest advantages of the n-tier application. Each layer can potentially treat the other layer as a “Block-Box”.

In other words, each layer does not care how other layer processes the data as long as it sends the right data in a correct format.

• The Presentation Layer: The Presentation Layer is also known as the client layer. This layer is comprised of components that are dedicated to presenting the data to the user. For example: Windows/Web Forms and buttons, edit boxes, Text boxes, labels, grids, and so forth.

• The Business Rules Layer: The Business Rules Layer encapsulates the Business rules or the business logic. To have a separate layer for business logic is of a great advantage. This is because any changes in Business Rules can be easily handled in this layer. As long as the interface between the layers remains the same, any changes to the functionality/processing logic in this layer can be made without impacting the others. A lot of client-server applications have failed to implement successfully as changing the business logic was a painful process [7].

• The Data Access Layer: The Data Access Layer is comprised of components that help in accessing the Database. If used in the right way, this layer provides a level of abstraction for the database structures. Simply put changes made to the database and, tables do not affect the rest of the application because of the Data Access layer. The different application layers send the data requests to this layer and receive the responses from it.

• The Database Layer: The Database layer is comprised of the Database Components such as DB Files, Tables, Views, and so forth. The actual database could be created using a tool such as MySQL Query browser. a view for Database tables has been created that stores task times, success rate, failure rate, ease of use scale, comments, suggestions and user details. In an n-tier application, the entire application can be implemented in such a way that it is independent of the actual Database. For instance, you could change the Database Location with minimal changes to Data Access Layer. The rest of the Application should remain unaffected.

IV. CASE STUDY SET-UP

Security Enhanced metric Taxonomy for SCRUM (SET4SCRUM) shown in figures 3 and 4 enables an organization to select significant metrics that are beneficial to its goals of providing secure software in an agile development lifecycle. It has role based login thus providing a appropriate list of metrics based on the role for a SCRUM framework to enhance an organization’s ability to make educated decisions in regards to secure agile software development. [16]
We perform a case study on SET4SCRUM using JRUTT to test the most important aspect of the system which is often neglected i.e., the “Usability”.

A remote usability testing administration is similar to the online survey administration, but with the additional steps of a traditional manual usability study, as follows [15]:

- **Define the study**: Decide the type of tasks that the participants will perform, decide the order of the tasks, and what follow-up questions you want to ask them about their experience.
- **Recruit participants**: Ideally five users are enough to discover 85% of the usability issues [3]. In this study, the author did the recruiting. However JRUTT also offers the options of intercepting users on a live Web site.
- **Launch test**: A remote test takes around 30 minutes comprising between 3 and 5 tasks, because the dropout rate tends to increase if a test takes longer. [12]
- **Analyze the results**: Both automated and manual results are gathered to reveal the critical usability issues of SET4SCRUM.

A. Scenario/ Task Description

The complete case study consists of four main scenarios where the user performs different tasks based on the scenario described in the interface. Figure 5 illustrates the first page of JRUTT that describes a scenario and task that has to be performed by the user. The details for the scenarios used in testing SET4SCRUM follow:

1) **Scenario I**

   a) **Scenario Description**: You have a new developer in your company and as a System Administrator, you should add the user onto the system and you should be able to re-instate the developer if he/she is accidentally removed from the system.

   b) **Task Description**: Your task is to add, remove and re-instate the user onto the system.

   c) **Instructions**: Click "Success" once you are done with your task or you can click "Quit" if you are not able to finish the task.

2) **Scenario II**

   a) **Scenario Description**: As a ScrumMaster, you want to add a new project to the system and set up all possible metric values for the project you just created.

   b) **Task Description**: Your task is to create a project and set up all the metric values for this new project.

3) **Scenario III**

   a) **Scenario Description**: As a ScrumMaster, you want to view and update the metrics of an existing project and generate a dynamic report for these metrics.

   b) **Task Description**: Your task is to view metrics of any existing project, update any metric and generate a dynamic report for these metrics.

4) **Scenario IV**

   a) **Scenario Description**: As a Developer, you want to change a number of security components in the Sprint for an already existing project.

   b) **Task Description**: Your task is to create a project and set up all the metric values for this new project.

Note that scenarios can be changed for other Java web applications just by changing the content enclosed in the <body> tag of the respective web pages.
We used the above scenarios customized according to SET4SCRUM to perform the whole case study. These four scenarios cover the main aspects of SET4SCRUM. Instructions provide the steps that a user has to follow while using the application to ensure JRUTT records the required usability data correctly. Scenarios, tasks and instructions descriptions are perfect instruments to automate to reduce the test moderator manual effort [12].

B. Live Participant Recruitment

In order to catch people in their native task environment, we implemented live online recruiting, one of the finest features of JRUTT. Live recruiting uses an online screener to intercept people in the middle of their real-life tasks, and watch them live as they complete those tasks in their native task environment. This feature of JRUTT lets any Java web application recruit an interested user to participate in the usability testing instead of providing incentives to the users who are not really interested in providing the feedback on an application. This lets a developer or tester collect invaluable data on participants’ true intent and motivation for visiting the SET4SCRUM application or any other web application. Figure 6 shows how the user is intercepted for live recruitment.

The JRUTT Live Recruitment page acts as the main entry point to collect user details if he/she is interested in participating in the usability testing. The importance of this feature can be clearly understood if it intercepts the actual web application so that the user can participate in performing usability testing with JRUTT. Our case study consists of five users and out of these five users, four users were interested in using the application and showed further interest in participating in future usability sessions. The users’ data (i.e., their e-mail and names) are collected if they are interested in participating. These participant details are visible at the administrator console of JRUTT. Note that the tool does not require the administrator to know SQL to process the data pushed into the database by the tool.

C. SET4SCRUM Evaluation

JRUTT is used to evaluate SET4SCRUM usability using two methods which follow:

- Each task is followed by a rating questionnaire shown in figure 7 to rate the subtasks associated with each task based on the ease of use.
- Comments and Suggestions are collected at the end of the usability study.

The user will get a chance to rate the ease of use of different subtasks after finishing each task. In addition the tool calculates the average ease of use of each task and displays the values at the administrator console.

Figure 8 collects the comments and suggestions input by the user and pushes into the database which can be viewed at the administrator console.

V. CASE STUDY RESULTS

A. Results Collected by JRUTT

Automated Results: A user clicks on either Success or Failure depending on whether he/she is able to finish up the tasks according to the instructions provided in the scenario.
JRUTT captures and records the time spent on each task, calculates the average times for all tasks, and displays these results in the Administrator console of JRUTT.

JRUTT also calculates both success and failure rates as is shown in figure 9. Furthermore, it captures the average failure time if users are not able to finish the tasks.

Our case study consisted of four basic tasks necessary to the use of SET4SCRUM. Nobody failed at finishing these tasks. According to the average time calculated by JRUTT, it seems that the second and third tasks took more time when compared to other tasks.

JRUTT calculates the average ease of use of each task. This information will be displayed at the administrator end and is shown in figure 10. The range of the scale used to measure the ease of use of each subtask is between 1 and 5. According to the results collected in this case study, the ease of use of both the second and third tasks was calculated as an average score around 3, whereas the first and third tasks were relatively easy with average scores above 4. The Administrator console also has a place to display the comments and suggestions collected using the JRUTT Comments and Suggestions form. Figure 11 depicts this form.

<table>
<thead>
<tr>
<th>Taskname</th>
<th>Average success time</th>
<th>Average failure time</th>
<th>Success #</th>
<th>Failure #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task1</td>
<td>2.49</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Task2</td>
<td>5.76</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Task3</td>
<td>5.54</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Task4</td>
<td>2.77</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 9. Success rate and time spent on tasks

<table>
<thead>
<tr>
<th>Taskname</th>
<th>Ease of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task1</td>
<td>4.67</td>
</tr>
<tr>
<td>Task2</td>
<td>4.13</td>
</tr>
<tr>
<td>Task3</td>
<td>3.93</td>
</tr>
<tr>
<td>Task4</td>
<td>4.20</td>
</tr>
</tbody>
</table>

Figure 10. Calculated ease of use for each tasks

While quantitative evaluation of SET4SCRUM is captured by the tool, qualitative evaluation and feedback was collected through the JRUTT comments and suggestions form. These comments appear below:

- Tree display structure present on the left side of the page does not work properly.
- There is no confirmation after updating the metrics.
- SET4SCRUM needs improvement in the navigation of links.
- I felt searching options needs time to find.
- The tree contracts for every step.
- There is no option to select all, update all metrics.
- Metric report management was not easy to find for a particular project.
- Links present on the left side of the page are not expandable at the first go.

Users suggested improvements based on the above comments. Though SET4SCRUM was a simple application, it suffered many design issues with very bad navigation. The first author moderated the usability test. Her observations follow:

- Tree Navigation does not work fine at all.
- There are no confirmations after updating and saving the metric values.
- Reaching metrics for a given project is difficult.
- Dynamic, static reports seem like they are disabled because of improper highlighting of the text.
- Tree structure does not change according to the page navigation. Instead, it contracts every time a user browses to other pages.

The developer of SET4SCRUM found these observations to reveal its critical design issues.

VI. JRUTT EVALUATION

A questionnaire to evaluate the effectiveness of JRUTT was also administered. On the questionnaires, users were asked to rate
the tool on a scale ranging from 1 to 5 about the usability experience. In addition, there was a section where users could provide feedback and suggestions to improve JRUTT. The summary of user experiences follows:

- The tool average rating for ease of use was 4.98 on the scale of 5.
- There was no difficulty with the usability of the tool.
- Comments and suggestions form should be integrated with the rating scale instead of having it in the end.
- Users often forgot the description of the tasks.

VII. CONCLUSIONS AND FUTURE WORK

Java Remote Usability Testing Tool (JRUTT) provides a framework to perform usability testing on any Java Web application to test one of its most important factors “Usability”. It provides both quantitative and qualitative data about an application through task elicitation. It has a simple interface to capture task times, success rates, ease of use for every task, comments and suggestions with live participant recruitment. JRUTT has many advantages that make it stand from many other tools:

- Administrative Console.
- Scenario Descriptions.
- Live Recruitment.
- Records failure time.
- Scale questionnaire with comments and suggestions.

JRUTT has basic features integrated to perform usability study on any Java web application. However, there is a lot of room for improvement as outlined in future work:

- The results displayed at administrator interface can be represented through graphs.
- Intelligent analysis of success and failure of the tasks can be analyzed more efficiently.
- A separate interface can be provided to the user for the customization of the scenario and task descriptions, as well as the scale questions.
- The tool is more useful if it is hosted on the server.
- The Task Description should be accessible while browsing the main application.
- The interface can be provided to the user to customize the rating scale for each subtask.

ACKNOWLEDGMENT

The authors are very thankful to all the participants for providing their valuable inputs on both SET4SCRUM and JRUTT.

REFERENCES

Towards Using Abstract Behavior Models to Evaluate Software System Performance Properties

James H. Hill
Indiana University-Purdue
University Indianapolis
Indianapolis, IN USA
Email: hillj@cs.iupui.edu

Harold Owens
Indiana University-Purdue
University Indianapolis
Indianapolis, IN USA
Email: owensh@cs.iupui.edu

Abstract—Evaluating software system performance properties requires constructing a test driver that exercises a test object (or system) under realistic conditions. In many cases, core intellect defined in the test driver is reinvented across different applications domains. This can cause software testers to expend much time and effort during the performance evaluation phase of the software lifecycle. This paper therefore presents initial results on using abstract behavior models to reduce reinvention of core-intellect in performance tests and facilitate automated performance test generation.

I. INTRODUCTION

Evaluating performance properties, e.g., throughput, service time, and response time, of a software system, both distributed and non-distributed, is traditionally an afterthought in the software lifecycle [7]. One major challenge that impedes validating such properties throughout the software lifecycle is the amount of time and effort required to perform the evaluation. For example, evaluating performance requires designing and implementing test drivers that exercises the test object (or system), not including the time required to execute the test. The test driver must also have correct behavior and workload to evaluate its corresponding performance property. This requirement, however, is challenging because (1) software testers may lack domain knowledge to construct the appropriate test driver and (2) implementing test drivers in parallel with the software system can be a heavy task. Moreover, test drivers are typically invented for each new system undergoing performance testing.

Model-based testing [5] and parameterized unit testing [8] are two approaches that help software testers reason about testing concerns (and objectives) at higher levels-of-abstraction (i.e., independent of a system’s concrete implementation). For example, in parameterized unit testing, software testers for create template-like behaviors that exercise the correctness of a test object, such as a container class. Traditional approaches for both model-based testing and parameterized unit test, however, focus on evaluating functional concerns of a software systems [2], [5], [9], such as verifying execution states. Applying such techniques to evaluating software system performance properties, however, has not been investigated.

This paper therefore presents initial work on a technique rooted in model-based testing and parameterized unit testing to evaluate software system performance properties. The main contributions of this paper are:

- It discusses an approach that uses abstract behavior models to capture core-intellect of a performance test;
- It discusses the technical challenges, and current solution approaches for using abstract models to generate complete performance tests; and
- It presents future research directions for realizing a complete solution that uses abstract models to evaluate software system performance properties.

Paper organization. The remainder of this paper is organized as follows: Section II presents examples to motivate the need for using abstract behavior models; Section III discusses the methodology of using abstract behavior models for performance testing; Section IV presents preliminary results; and Section V provides concluding remarks and future research directions.

II. MOTIVATING EXAMPLE

Listing 1 highlights three simple examples for evaluating service time of three different test objects from three different application domains. As shown in this listing, each test object contains implementation that is tied to its corresponding application domain. For example, Database is an object that provides methods for accessing a database. Likewise, Component is a reference to a remote object (i.e., resides on a different machine).

```java
// generic object
long TestServiceTime (Calculator c) {
  // generate input data
  // ...
  long startTime = Util.getTImeOfDay();
  c.calculate (*input data*);
  return Util.getTImeOfDay() − startTime;
}

// database object
long TestDBServiceTime (Database d) {
```
As shown in Listing 1, similar code is used to evaluate the service time of a method exposed by each object. For example, each test must (1) contain correct behavior that implements the testing goal, (2) generate valid input data, and (3) integrate with the test object. This can be considered “boilerplate” code, which is relatively similar for each test object that wants to evaluate service time in different application domains.

Although Listing 1 highlights examples that could be achieved by using well-defined interfaces and library routines, constructing valid (or realistic) tests that evaluate performance when dealing with complex behavior or systems is not always a trivial process. For example, asking software testers to construct a test that identifies worst-case service time or meets a testing objective that must coordinate with many other test objects is challenging. This is because it depends on understanding how to construct a test that achieves the identified goal. The remainder of this paper therefore discusses the idea of using abstract behavior model to facilitate performance testing such that cross-domain performance evaluation is not a costly process.

### III. Abstract Behavior Modeling for Performance Testing

This section discusses the methodology of using abstract behavior models to test software system performance properties.

#### A. Composition of Performance Tests

Before discussing the details of using abstract behavior models to evaluate performance properties, it is first necessary to understand the composition of a performance test. Based on experience, a performance test is composed of the following entities:

- **Test object.** The test object is the entity undergoing performance testing. It can be a single object, or it can be multiple objects (e.g., a distributed system). The test object is fixed (i.e., its structure and behavior does not change) during performance testing.

  - **Interface model.** The interface model is the portion of the test driver that binds it to the test object. The interface model therefore is derived from the interface of the test object. This also implies that the interface model changes for each test object undergoing performance testing.

  - **Behavior model.** The behavior model is the core intellect of a performance test. It dictates how the test driver exercises the test object. Moreover, the behavior model determines test validity. For example, if a behavior model is designed to evaluate throughput in a distributed system and does not try to send as many events as possible, then this is considered an invalid test. Finally, the behavior model for given performance test should remain (relatively) the same across different application domains.

  - **Data model.** The data model is the set of values sent from the test driver to the test object while it is undergoing performance testing. The data model is the solution space of all possible values that the test object can handle, which is usually based on the test object’s exposed attributes and methods, and testing objectives. The data model also determines test accuracy. For example, when evaluating response time, service time, or searching for an optimal value, the data model determines how close the testing effort is to its goal, and how long it takes to reach the testing goal. Finally, the data model can be static or dynamic [10]. A static data model is determined offline before the test is run, while a dynamic data model is determined at runtime. This paper focuses on the static data model.

  - **Test oracle.** The test oracle’s purpose, as in all testing efforts, is to validate performance test results. Currently, it is assumed that the test oracle is manually handcrafted when using abstract behavior models for performance testing.

  In this composition, there are three critical models: structure, interface, and data model. The interface and data model can be learned by examining the test object. For example, it is possible to learn the interface model for a test driver that evaluates response time by discovering the plausible methods to evaluate on the test object. Moreover, it is possible to generate a data model using code analysis techniques, such as dynamic symbolic execution [1].

  The behavior model, however, cannot be learned automatically from the test object. Instead, the behavior model must be manually constructed for different kinds of performance tests. Once it is created, the behavior model remains constant. This therefore raises the following challenges:

  1) What is the best way to represent (or define) the abstract behavior model of a performance test?
  2) How is an abstract behavior model used to evaluate a concrete test object?
B. Constructing Abstract Behavior Models

Abstract models define the generalized behavior of a test driver. In addition, such models are able to send and receive events from other test drivers and the test object. When using abstract behavior modeling to define a performance test, there are two classifications:

- **Simple behavior** – This is a performance test that consists of a single abstract behavior model such that performance evaluation can be accomplished using a single test driver object. Examples of performance tests based on simple behaviors include: latency, throughput, and service time.

- **Complex behavior** – This is a performance test that consists of two or more (i.e., more than one) abstract behavior models that must coordinate with each other to achieve the desired testing goal. Figure 2 illustrates the concept of a complex behavior. As shown in this figure, there are multiple behavior models that interact with the test object and each other.

As discussed above, the abstract behavior model must be able to communicate with the test object and other abstract behavior models. Its modeling language therefore must be reactive (or event-driven). Because of this requirement, Timed Input/Output Automata (TIOA) [6] is selected as the formal base language for defining abstract behavior models. The behavior model in TIOA is defined as

$$BM = (X, Q, \Theta, E, H, D, T)$$

where $X$ is the set of internal variables; $Q \subseteq \text{val}(X)$ is a set of states; $\Theta \subseteq Q$ is a set of start states; $E$ is a set of external actions and $H$ is a set of actions, which are disjoint from each other; $D \subseteq Q \times A \times Q$ is a set of discrete transitions; and $T \subseteq \text{traj}(Q)$ is a set of trajectories.

For the purpose of abstract behavior modeling, TIOA does not differentiate between external actions to other behavior models, which are called **control actions**, and external actions to the test object, which are called **test actions**. The external actions therefore are redefined as $E = E_{test} \cup E_{control}$ where $E_{test}$ is the set of external test actions and $E_{control}$ is the set of control actions. $E_{test}$ and $E_{control}$ are disjoint sets.

The semantics (i.e., concrete name and parameters) of test actions are defined by the test object. The control actions, however, are governed by the semantics of TIOA. This means that they (1) are always active, (2) are connected by their unique names, and (3) occur instantaneously when fired. For example, if one abstract behavior model has $E_{control} = \{\text{reset}, \text{increment}\}$ and another abstract behavior model fires the $\text{reset}$ action, then the event is received instantaneously on $\text{reset}$ external action of the former abstract behavior model. Finally, control actions can contain parameters for passing data, which is carried over from TIOA.

Figure 1 illustrates an example abstract behavior model for evaluating throughput using a graphical implementation of TIOA that supports sequencing of internal actions for deterministic execution [3]. The key elements in this example are: $V=\{\text{isActive}, \text{startTime}, \text{eventCount}\}$, $E_{control}=\{\text{activate}, \text{passivate}\}$, and $E_{test}=\{\text{Port1}\}$. Since $\text{Port1}$ variable is a test action, it has no concrete semantics. Instead, its concrete name and data is dependent on the test object. This data can be a simple integer, or a complex data type.

C. Instantiation and Integration of Abstract Behavior Models

Section III-B discussed the concept of using abstract behavior models to capture the core intellect of a performance test. The abstract model operates independent of the test object’s target programming language, architecture, and platform. To realize a complete performance test, the abstract

*Figure 1. Graphical version of a simple abstract behavior model for evaluating throughput.*

*Figure 2. Overview of a complex behavior model.*
behavior model must integrate with a interface model, data model, and test object. This process is called abstract model instantiation and integration.

```java
behavior AvgServiceTime (X) {
  // ...
  input activate () {  
    ca = 0; done = false; i = 1;
    while (!done) {
      long start = getTimestamp ();
      call (X, Port1);
      duration = getTimestamp () - start;
      ca += ((duration - ca) / (i++));
    }  
  }  
}
```

Listing 2. Portion of an abstract behavior model for evaluating average service time.

Listing 2 highlights the idea of an abstract behavior model (in text format) called AvgServiceTime that evaluates the average service time of a test object’s method. As previously stated, the abstract behavior model does not have concrete semantics. To give the example in Listing 2 concrete semantics for realizing a complete performance test, the abstract instantiation process must construct a test driver such that its behavior resembles that in Listing 2. More specifically, the test driver must (1) invoke a concrete method on the test object and (2) pass the test object valid data from the data model (line 7 in Listing 2).

```java
public class AvgServiceTime extends Behavior {
  private DataProcessor dp_; // init by setter
  // ...
  void activate () {
    ca = 0; done_ = false; i = 1;
    long start = getTimestamp ();
    dp_.processData (getData ("data@1").nextValue ());
    elapse = getTimestamp () - start;
    ca += ((elapse - ca) / (i++));
  }
}
```

Listing 3. Abstract model instantiation and integration of the previous example.

Listing 3 illustrates the abstract model instantiation and integration of the example presented in Listing 2 for a test object named DataProcessor. As shown in this listing, object \(X\) is replaced with the concrete object \(dp\) of type DataProcessor. Likewise, the `call()` method is replaced with a method invocation to `processData` on the `dp` object. Finally, the `processData` is invoked with data generated by the `data81` data generator. This implies that each variable in an test output action has its own data generator in the data model. Likewise, the naming convention for a data generator is `id@ordinal` where `id` a unique `id` and `ordinal` is the parameter’s ordinal in the concrete method.

IV. PRELIMINARY RESULTS

The initial implementation of the framework for using abstract behavior models to validate performance properties was implement using the open-source tool called The Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS) [4]. CUTS was selected as the prototype platform because it supports code generation of software objects from abstract behavior models constructed using the graphical version of TIOA (shown in Figure 1) that can target many different programming languages and architectures. CUTS was updated to support the abstract modeling approach presented in Section III for the Java programming language.

More specifically, CUTS was updated such that (1) developers use `javap` to decompile a Java test object and import it into CUTS. (2) Testers then select the abstract behavior model of a performance test for evaluating the test object. (3) Next, testers manually map the test actions to concrete methods on the test object. Finally, CUTS auto-generates an interface model and data model for the test driver, and integrates it with an instantiated model. The current data model defaults to randomly generated data, which can be overridden in the generated data model file.

Figure 3 shows results for evaluating the service time of for different implementations of a calculator test object using an abstract behavior model. In this test, the default data model (i.e., random data generators) was overridden with domain-specific data generators that increased the number of operands after each iteration. As shown in this figure, the service time of the calculator increases (as expected) as the number of operands increase. Although this result is for a simple behavior, it's evaluation was driven by an auto-generated performance test.

V. CONCLUDING REMARKS

This paper presented initial results on using abstract behavior models to facilitate performance testing. The overall goal of the approach is to reduce the amount of core intellect that is reinvented across different application domains. It is believed that doing so will help reduce cost associated with evaluating performance properties throughout the software.
lifecycle. Based on preliminary implementation and results, the following are future research directions:

- **Evaluate the feasibility of using TIOA as the foundation for abstract behavior models.** TIOA was selected because of the abstract behavior model’s reactive nature, and the code generation support provided by tools that support TIOA. The use of TIOA and its extensions, however, remain to be validated for modeling abstract behavior models. Future work will investigate this concern.

- **Evaluate complex abstract behavior models.** This is because the current results illustrate usage with simple abstract behavior models. Applying the approach on complex behavior models will help improve the theoretical aspects of defining the behaviors. Moreover, it will highlight many technical challenges when coordinating many abstract behavior models from an implementation point-of-view—especially when such models are distributed across many different hosts (i.e., distributed system testing).

- **Publishing results for the test oracle.** The current specification does not provide an easy method for publishing information so the test oracle can determine test success or failure. Future research will investigate how this need can be incorporated into the abstract behavior model specification.

The initial version of the abstract behavior model framework is freely available in open-source format at the following location: www.cs.iupui.edu/CUTS.

**REFERENCES**


APPENDIX

The following is the complete implementation of the abstract model instantiation and integration for evaluating the calculator's service time presented in Section IV.

```java
public class CalculatorServiceTime extends Behavior {
    private Calculator c;
    private boolean done;

    public CalculatorServiceTime () {
    }

    public void setCalculator (Calculator c) {
        c = c;
    }

    public Calculator getCalculator () {
        return c;
    }

    public void setDone (boolean done) {
        done = done;
    }

    public boolean getDone () {
        return done;
    }

    // external control action (non-blocking)
    public void activate () {
        done = false; i = 1;
        while (!done) {
            long start = getTimestamp ();
            c.evaluate (getData ("data@1").nextValue ());
            elapsed = getTimestamp () - start;
            System.out.println ((i++) + " " + elapsed);
        }
    }

    // external control action (non-blocking)
    public void deactivate () {
        done = true;
    }
}
```

Listing 4. Complete implementation of the instantiated model for evaluating the service time of the calculator.

The following is the data model that was overridden with a domain-specific data generator to produce the results presented in Section IV.

```xml
<datamodel>
    <generator id="data@1"
        class="CalculatorDataGenerator" />
</datamodel>
```

Listing 5. Data model that was overridden to evaluate the calculator.

The behavior in Listing 4 is initialized with this data model.
Applying Testability Concepts to Create Testability Guidelines

Robin Poston and Jignya Patel
Department of MIS
University of Memphis
Memphis, USA
rposton@memphis.edu, jmpatel@memphis.edu

Bindu Varriam and Fatima Qadri
Quality Assurance
Federal Express
Collierville, USA
bindu.varriam@fedex.com, fatima.qadri@fedex.com

Abstract—This article presents an initial attempt at creating guidelines for software testability. While the guidelines proposed in this study are currently limited to the interface testing domain, with minor modification they should be expandable to the entire testing lifecycle. We start by using an auditing framework and borrow from the concepts of substantive and evaluative procedures found in the financial auditing literature to inform our approach in creating testability guidelines. This approach ensures a more comprehensive and rigorous series if steps is followed. Specific to technology, we also use components of the widely-adopted Control Objectives for Information and related Technology (COBIT) framework. We follow a case study approach to gather data from expert informants. As a result, we create and present an initial set of testability guidelines. Future steps will include cross validating the guideline using actual software projects by measuring the testability attributes of each project per the guidelines proposed.

I. INTRODUCTION
As part of the Systems Testing Excellence Program (STEP) research effort in 2010, STEP researchers introduced the concept of software testability and its role to achieve testing goals when selecting an enterprise architecture solution. 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). Academic scholars have identified several heuristics of software testability (Bach, 2003) out of which three emanated to be of utmost importance: Controllability, Observability, and Simplicity. Controllability reflects the testers’ ability to control inputs to the software code being tested and how the code within it works. If the testers cannot control the input, they cannot be sure how a given input was processed by the code. The less controllable the software code is, the less testable it is. Observability reflects the testers’ inspection of the output of the software code being tested (i.e., how well the code performs its job). Lacking observability means that the tester can be tricked into seeing the correct output for a particular input when in reality it may not be the case. Simplicity refers to the fact that the software design should exhibit functional simplicity with minimal feature sets and structural simplicity with cohesive and loosely coupled code (Bach, 2003).

Building on these principles, we take the concept of testability one step further. How can one know how testable a project is? Can it be measured and quantified into a single number which represents the probability of the software’s testability? To answer these questions, we apply the concept of evaluative and substantive procedures from the auditing literature to develop a set of testability guidelines (i.e., a testability audit). Evaluative procedure asks, in the case of software testing testability, “Whether the right steps are being taken to test software and are there ways of ensuring testability?” (Anonymous, 2011). Substantive procedure, in the case of software testing testability, refers to gathering evidence that what a company is doing across the software development lifecycle follows the heuristics of testability of: controllable, observable, and simplicity. We create the testability guidelines by meeting with key informants (i.e., members of the interface testing team at a large transportation company and at another corporate entity) to solicit their expertise in exhaustively identifying issues of testability with each of the different components of an IT project, e.g., documentation, software, security, etc. Our aim was to unveil the indicators of testability and provide guidelines for enhancing testability throughout the development lifecycle, starting from the interface testing perspective. Our next step in this research is to gather evidence that our guidelines can be applied to actual
software projects and be modified to apply to alternative testing perspectives.

II. THEORETICAL BACKGROUND

We develop our theoretical background from the accounting literature in auditing. But first, we justify why we chose the auditing perspective. The justification lies in the very definition of Auditing. An audit enables management and other stakeholders to find out about potential problems and implement corrective actions, modifications, and improvements wherever needed. (Hinson, 2008) This definition is in sync with what we are trying to achieve, i.e., trying to find the problematic issues that decrease the testability of software products and suggesting measures that indicate the testability of software. Audits serve as opportunities for companies to improve performance based on an audit analysis. The general perception of an auditor is a person who looks for ineffective, inadequate, and inefficient strategies, policies, or practices that could jeopardize the integrity of the corporation. By using an auditing mindset in our case study approach, we support the credibility of our guidelines as measuring software testability. We borrow the concept of evaluative procedure from the auditing literature, which consists of a) identifying and understanding a particular process and then b) improving or redesigning the process or making minor changes in the process to result in an improved achievement (Anonymous, 2011; Lindeberg, 2003). Evaluative procedure, in the testing context, basically asks: Whether the right steps are taken to test software and are there better ways of conducting testability?

Although the concept of auditing was justified from the accounting literature, we also consulted the IT auditing literature to derive the initial list of IT issues. Specifically we looked into the COBIT and Val IT frameworks. COBIT and Val IT are registered trademarks of Information Systems Audit and Control Association, ISACA and IT Governance Institute, ITGI. Although the COBIT framework is used for IT governance, its first version, COBIT 1.0 was developed as an auditor’s tool for IT governance. It is a methodology that consists of standards and controls that are created to assist IT professionals in the implementation, review, administration and monitoring of an IT environment including the software development process (Lindeberg, 2003). COBIT is generally accepted in the information systems auditing community and has been commonly used for IT governance implementation and assessment. COBIT has also been used from the auditing perspective of agile software development (Gupta, 2008), for determining compliance of the projects with Sarbanes Oxley Act (SOX), a regulatory requirement for all public listed companies in United States (Mahnic, 2008). Thus the COBIT literature served as a starting point to identify the list of issues that auditors seek when evaluating a software project using COBIT.

Based on these perspectives, we created an initial list of the relevant components for generating a set of testability guidelines, as follows:

<table>
<thead>
<tr>
<th>Component</th>
<th>Sub-component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>critical applications</td>
</tr>
<tr>
<td></td>
<td>number of licensed concurrent users</td>
</tr>
<tr>
<td></td>
<td>version levels</td>
</tr>
<tr>
<td></td>
<td>interaction with other applications</td>
</tr>
<tr>
<td></td>
<td>where and how applications are executed</td>
</tr>
<tr>
<td></td>
<td>database structure</td>
</tr>
<tr>
<td></td>
<td>level and type of support by the software vendor</td>
</tr>
<tr>
<td></td>
<td>are users adequately segregated by functions?</td>
</tr>
<tr>
<td></td>
<td>are there appropriate number of software licenses?</td>
</tr>
<tr>
<td></td>
<td>is data stored in a logical, secure and easy access format?</td>
</tr>
<tr>
<td></td>
<td>policies against viruses</td>
</tr>
<tr>
<td></td>
<td>are patches are up-to-date?</td>
</tr>
<tr>
<td></td>
<td>input and output controls</td>
</tr>
<tr>
<td>Hardware</td>
<td>system file servers</td>
</tr>
<tr>
<td></td>
<td>workstations</td>
</tr>
<tr>
<td></td>
<td>network hubs</td>
</tr>
<tr>
<td></td>
<td>wiring</td>
</tr>
<tr>
<td></td>
<td>communication devices</td>
</tr>
<tr>
<td></td>
<td>laptops/desktops/pad</td>
</tr>
<tr>
<td></td>
<td>printers</td>
</tr>
<tr>
<td></td>
<td>peripherals</td>
</tr>
<tr>
<td></td>
<td>fileserver integrity</td>
</tr>
<tr>
<td></td>
<td>hard drive space</td>
</tr>
<tr>
<td></td>
<td>amount of RAM</td>
</tr>
<tr>
<td></td>
<td>processor speed</td>
</tr>
<tr>
<td></td>
<td>drive partition information</td>
</tr>
<tr>
<td></td>
<td>operating system upgrade versions</td>
</tr>
<tr>
<td></td>
<td>data storage optimized for speed and user ease of use</td>
</tr>
<tr>
<td>Documentation</td>
<td>log files</td>
</tr>
<tr>
<td></td>
<td>disaster recovery plans</td>
</tr>
</tbody>
</table>
This initial list of the relevant components serves as our starting point for generating a set of testability guidelines via brainstorming discussions with key informants at one large case study company.

III. RESEARCH METHODOLOGY

Given the lack of a systematic frameworks in the literature to investigate the creation of a set of testability guidelines that cover controllability, observability, and simplicity testability criteria, we take an exploratory rather than a confirmatory approach (Yin, 1989). When there exists limited understanding of a given phenomenon, the case study methodology is considered a suitable approach, in which the research process is tied directly to field data (Eisenhardt, 1989). The case study methodology allows us to incorporate the insights of experts in the field of study into the outcomes of this study (Whitley, 1995). Also when ‘how’ or ‘why’ questions are posed, a case study is more suitable (Yin, 1989) to examine the phenomena in a business-world setting. Our investigation of testability guidelines satisfied these criteria.

Case Study Site

We selected a Fortune 500 multinational service organization, TransCo (a pseudonym), headquartered in a major urban center in the U.S. TransCo is a global leader in transportation existing for over 40 years. As a technology driven company, the company has over 270,000 employees worldwide with $30 billion in revenues. TransCo has about 5,500 IT employees overall, with separate IS development and testing groups. The testing group has managing directors, project managers, and about 150 full-time employees total with much of the testing execution work performed by offshore vendors. Development teams are responsible for unit testing while the testing division performs the system and integration testing. TransCo has 3 major releases of hundreds of new and modified software at pre-designated times over a year.

Since our primary objective was to investigate the creation of a set of testability guidelines that cover the testability criteria of controllability, observability, and simplicity, we needed a case where the client organization was 1) routinely involved in system and integration testing within their IS function, 2) willing to share their perceptions regarding what makes software products more or less testable, and 3) willing to introduce us to the appropriate key informants throughout the organization. TransCo satisfied these criteria.

Data Collection

Qualitative data was collected over a 3-month period (Phase 1 data collection: January 2011 – March 2011, 4 interviews and we are currently planning Phase 2 data collection: April – June 2011) through unstructured and semi-structured interviews with various individuals from multiple groups in the organization. Our data gathering began with interviews of key informants to better understand the issues related to testability of software products and projects. Key informants included project managers with first-hand testing experience. Interviews initially focused on several dimensions of software testing across IS components: software, hardware, documentation, security, system environment, data, personnel, and facilities. Interviews lasted from 90 to 120 minutes each. Some key informants were interviewed more than once. All interviews were conducted in a face-to-face setting with interviewees and both researchers present. We took copious notes and transcribed all these face-to-face interviews, simultaneously building the testability guidelines drafts. We also examined software development and testing process documentation used by TransCo. We conducted follow-up interviews and emails as needed to get clarifications and to validate our interpretations.
Data Analysis

Our data collection and analyses progressed iteratively. As data was being collected, we analyzed it and identified which testability guidelines were relevant to interface testing, allowing us to refine/broaden and verify/disconfirm the draft guidelines being created (Pettigrew, 1990). We started with a broad definition of the testability guidelines that cover the testability criteria of controllability, observability, and simplicity, which was refined through the further interviews and data analysis and review of the literature. Phase 1 data collection resulted in a write-up of various facets of the testability guidelines and will be shared at an academic research colloquium and workshop. At this time, however, our analysis of the phenomenon of vendor silence is very preliminary, which is why Phase 2 data collection will focused on the implementation of the testability guidelines to assess several software products as a proof of concept activity, with the guidelines modified as deemed appropriate.

To establish that our findings have credibility (i.e., similar to internal validity in qualitative studies (Guba, 1981), we use triangulation in qualitative studies. Triangulation is accomplished through the use of multiple data sources and multiple researchers (Mason, 1996). Iterative comparison and contrasting and cross-examination of our work across multiple interviewees allow us to ensure that the outcomes of this research are well developed. Two researchers conducted the interviews, with one researcher asking the questions and the other listening and taking notes and asking follow-up questions. Presence of multiple researchers allows us to systematically recognize, discuss, and debate different interpretations and improve our understanding of the challenges faced by the interviewees in defining testability guidelines for interface testing. To further improve credibility we also employed member-checking and peer-debriefing (Strauss and Corbin, 1990). We will present drafts of our testability guidelines to the members of TransCo (member-checking) and other researchers (peer-debriefing) at the research colloquium and workshop discussed above to gather additional input. To enhance the internal validity and improve generalizability of the research, Orlikowski (1993), Pandit (1996), and Strauss and Corbin (1990) recommend combining the findings of qualitative research to existing literature.

IV. FINDINGS

We started with initial list of the relevant components from IT auditing. We then determined the testability guideline for each component based on expert informant input from TransCo, as well as another large multinational firm, as it related to the interface testing domain. We then assessed whether each component addressed any of the testability heuristics of controllability, observability, and simplicity. As a result, our initial testability guidelines incorporated items that were considered out of scope for interface testing. The figure below illustrates our intermediate guidelines prior to removing out-of-scope items.

We draw upon the literature on auditing, COBIT, and IT Val frameworks to improve the analytical generalizability of our findings. Out of these, the COBIT framework proved to be the most useful since it gave us most of the comprehensive list of IT components which served as the basis for analyzing the testability of software products in the interface testing domain. This list served as a thought provoking instrument to our interviewees when thinking about testability. After conducting three rounds of interviews, we realized that there were parts of the list that were not relevant to our context of testability in software development. Thus they were removed from the list. It was better to keep every component in the list at first (even the irrelevant ones) since we thought that its presence could potentially trigger a thought in the interviewee’s mind that could be worth noting. Once all the irrelevant items were removed, the document was developed further by assessing each response of the interviewees and determining whether they fit in the controllability, observability, and/or simplicity categories. We also cross validated the testability guidelines with several testing team professionals from another large multinational firm. As a result of the cross validation only minor changes were made. As a future step, we intend to use the guidelines in several of TransCo’s ongoing projects and test how the measures can be captured for those projects. If the guidelines withstand the test, we intend to propose them as general audit-type guidelines for the testability of any software development project.
The Appendix illustrates all the guidelines with the out-of-scope elements removed. Note the far right-hand column which will be used for future assessments of how projects measure for each guideline on a 7-point Likert scale, where 1 = does not exhibit that guideline and 7 = strongly exhibits that guideline.

The eventual goal of this research is to measure the guidelines across several testing projects to update and modify these guidelines and once stabilized, to create a baseline of testability benchmark measures for any corporate to use in assessing issues of testability for ongoing projects across the software development lifecycle.

V. IMPLICATIONS AND FUTURE RESEARCH

Several important implications result from this study. First, the software testability guidelines created in this study can be a useful audit technique that testing teams can use to determine the level of testability in their projects, as assessed across the stages of testing performed across the software development lifecycle. This will enable project managers to determine the appropriate resources needed to test certain features and functions of the software, and the appropriate level of resources to be assigned to components of the projects based on the measured level of testability. It will also help project managers highlight where areas of improvements in testability can be achieved giving testing leads ways to discuss with stakeholders project improvement ideas. While this study examines only interface testability guidelines, future research could expand this work to create guidelines for other types of testing (i.e., unit, system, user acceptance, interoperability, etc.) in order to support testing teams throughout the entire software development lifecycle.

Another important implication of this study is to use the software testability guidelines as a benchmark-type tool to determine whether projects are more or less testable. A database of projects can be gathered and used to determine patterns of the factors that drive testability. Factors could include project size, project manager style, whether offshoring was involved or not, criticality of the software to the user base, etc. As benchmark data builds, best practices in software testability can be derived and shared and future projects assessed to determine if improvements have been made. Future research should create the repository (i.e., database) of project demographic and testability measurements in order to assess what factors drive testability across multiple types of projects and testing teams.

Finally, the testability guidelines can be used as a checklist for project managers to ensure highly testable software projects. As project managers review the guidelines they may tweak their test plans and resource allocations to change coverage and issue management in order to ensure more testable software all along the development lifecycle. In our case study example, discussions about the guidelines led one expert informant to notice her team needed to improve how it tracked open issues (i.e., the many emails of issues that floated around without accountability) and how it handled the level of confidence of team members with respect to their understanding of the various requirements documents. Future research should continue to determine whether the guidelines trigger additional learning, understanding, and insightful comments from testing experts to continue to update and improve the testability guidelines making the most comprehensive and accurate list possible.
REFERENCES


APPENDIX

Below is the result of starting with categories from auditing and IT auditing, then through brainstorming sessions with key informants in the interface testing domain, we created testability guidelines for each component. Finally, we assessed whether each guideline related to the testability heuristics of controllability, observability, or simplicity, and provide assessment measures of each guideline.

<table>
<thead>
<tr>
<th>IT Components (Source: COBIT, etc.)</th>
<th>Testability Guideline</th>
<th>Controllability</th>
<th>Observability</th>
<th>Simplicity</th>
<th>Measure (1-7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical applications</td>
<td>Quality level of the original software application before the start of testing</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interaction with other applications</td>
<td>Number of changes in interfaces with no / little visibility</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Where and how applications are executed</td>
<td>Defined data mapping between systems</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Visibility to data mapping to input and output of interfacing systems</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Degree of ability to decouple the code between interfacing systems (more data/switch driven less code driven)</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ability to control the test parameters (e.g., modify data retention periods or test system outage scenarios)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Database structure</td>
<td>Comprehensiveness of documentation of changes in database structure</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Are patches are up-to-date?</td>
<td>All patches been applied within the test environment before the start of testing</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input and output controls</td>
<td>All input and output controls are completely documented</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data dependencies are completely documented</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changes that affect other systems are completely documented</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error messages</td>
<td>Error messages provide clear description of the problem</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Error handling processes are efficient</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System components</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business Requirements (BRS)</td>
<td>Level of involvement of testing representative in the document walkthrough</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Comprehensive assumptions and constraints have been included</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Detail business scenarios and</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Requirements (SRS)</td>
<td>Level of involvement of testing representative in the document walkthrough</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓  ✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Comprehensive assumptions and constraints have been included</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓  ✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detail business scenarios and examples have been included</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traceability to BRS has been documented</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stakeholder review and approvals exist</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Version control in place and followed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open issued should be tracked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration Control has been done</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Architecture Specification (SAS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Completed and provided with entire system flow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>De-coupling/ Back-out Plan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Document is complete and provided</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level of detail explanation as it related to features</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detail Test Plan Specification (DTPS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stakeholder review and approvals exist</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Confidence level in understanding of DTPS of testing team members</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Version control in place and followed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Known location of organized repository of project files</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mitigation and Contingency plan know for risk analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log files: Defect log files</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All defects and their remedies are logged in an easily accessible manner by the testing group</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access controls</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access rights to all impacted systems have been set up before the start of testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓  ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Description</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>Internal controls on key applications</td>
<td>Level of testing for balancing controls (e.g., HIPPA, SOX, PCI)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>Data encryption Ability to simulate sensitive data or encrypted data</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Level of complexity in decrypting encrypted data</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facilities</td>
<td>Test environment Separate testing facility from the remaining software development team</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
</tbody>
</table>