IEEE International Conference on Software Testing Verification and Validation Workshops

An Initial Study of Customer-Reported GUI Defects

Brian Robinson, Penelope Brooks
ABB Corporate Research
Raleigh, NC, USA
{brian.p.robinson, penelope.a.brooks}@us.abb.com

Abstract

End customers increasingly access the delivered functionality in software systems through a GUI. Unfortunately, limited data is currently available on how defects in these systems affect customers. This paper presents a study of customer-reported GUI defects from two different industrial software systems developed at ABB. This study includes data on defect impact, location, and resolution times. The results show that (1) 65% of the defects resulted in a loss of some functionality to the end customer, (2) the majority of the defects found (60%) were in the GUI, as opposed to the application itself, and (3) defects in the GUI took longer to fix, on average, than defects in the underlying application. The results are now being used to improve testing activities at ABB.

1. Introduction

Most software systems today provide a Graphical User Interface (GUI) as the main interface to the delivered functionality for the end customer. While many developers consider the GUI to be a less critical part of the system, it is essential to customers, who must use it whenever they need to interact with the system. Simple defects in the GUI, such as incorrect values in GUI properties, can result in the user experiencing a loss of delivered functionality in the system.

To test a GUI, the test designer develops test cases that are modeled as a sequence of user events and executes them on the software via the GUI, either manually or automatically. These events include clicking on buttons, selecting menus and menu items, and interacting with the functionality of the system provided through the GUI. Events are governed by context, where some events may not be executed in sequence with other events, while other events require the execution of one or more events before they are enabled. Legal event sequences are defined implicitly as part of the GUI development. Defects are manifested as failures observed through the GUI. Some of these failures are due to GUI defects (e.g., the text-label is incorrect, the OK button is missing a caption), or application defects (e.g., the result of the computation is incorrect). A *GUI defect*, for the purpose of this paper, is defined as a defect in the GUI itself, as opposed to application defects that are observed through the GUI. The GUI includes the code which makes up the GUI objects and the glue code that connects those objects to the underlying application. All other defects in the software are considered *application defects*.

Effective GUI testing is a difficult problem, as the large number of valid and invalid actions and states that exist inside GUIs leads to a combinatorially impractical number of tests. Since exhaustive testing is infeasible, many recent GUI testing techniques target functional defects in the system. These techniques include capture-replay tools [7], operational profile-[5] or user profile-based methods [3], structural testing [12][13], and *n*-way event testing [8].

Research in GUI testing has focused on university-developed applications and open source software. Due to the difficulty in creating test oracles for unfamiliar applications, the defects detected in evaluative studies were either seeded into the software or were detected through an application crash [8][12].

This paper presents a case study of customer-reported GUI defects on two large, industrial GUI systems that are used by customers around the world. The study focuses on defects from the user's perspective and aims to capture its impact to the customer, location in the system, and the duration the customer experienced the defect before a fix was released.

The contributions of this work include:

- A case study on two large, deployed, industrial applications
- Measures of the impact that GUI defects have on customers
- Measures of where in the system these customer GUI defects are
- Measures of how long customer GUI defects took to fix

This paper is organized as follows: Section 2 presents related work in the areas of GUI testing, GUI defects, and case studies on other types of customer defects. Section 3 provides detailed examples of three customer defects in the study. Section 4 describes the research design for this study, including a description of the systems used. Section 5 presents the results, and Section 6 presents the discussion and analysis. Conclusions and future work are presented in Section 7.

978-0-7695-3671-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICSTW.2009.22

2. Related Work

To our knowledge, previous research in testing has not studied the impact of GUI defects on customers. However, defect studies have been conducted on customer-reported defects, to assess the impact on customers; these studies did not report on GUI defects separately. This section will describe some of this previous work as it relates to our work.

Sullivan and Chillarege [11] studied the types of defects found in database management systems (DBMS) as compared to those found in operating systems (OS). Their work focused on field defects and their impact, as reported by the customer and the customer service representative for two DBMSs and one OS. The study compared the error type, trigger and defect type for the two types of systems studied. They found the OS and the older DBMS had about the same percentage of high-impact errors, though the errors detected in the DBMS were marked high-impact by the maintenance programmers while the errors in the OS were marked high-impact by the customer. The newer DBMS varied from the other two systems in high impact errors; maintenance programmers and customers both rated the errors as high-impact. Their study also supported the intuition that younger products have higher defect rates and that those defects have higher customer impact than older products.

Gittens *et al*. [6] performed a study of the effectiveness of in-house testing by investigating the breadth of the system and regression testing on factors such as code coverage, the number of defects detected in-house, and the number of defects detected in the field after release. They had several interesting findings. First, using a test suite with regression testing code coverage in the range of 61-70% and system test coverage in the range of 51-60%, very few defects are detected in the field. Second, the study showed that as in-house testing increases per module up to 61-70% coverage, the number of field defects also increases. This is counterintuitive, but supported by their data. Third, for in-house module testing that achieves greater than 70% coverage, field defects decrease. Therefore, their overall findings show that code coverage of about 70% is very effective at decreasing field defects. While the study reported here does not correlate code coverage with customer-reported defects, the findings from Gittens' study complement our findings in showing the importance of testing.

Musa, well-known for his work in software reliability, developed the Software Reliability-Engineered Testing (SRET) technique [9], and applied it to the Fone Follower, a system that implements telephone call forwarding. His SRET technique classifies a failure in one of four severity classes based on service impact. Subsequent regression testing then includes information on previous failures when planning the failure intensity objectives for new software.

Adams [1] studied five years of customer-reported defects for nine products in the hope of determining the cost-benefit tradeoff between preventive and corrective service. He found that most failures in the field were caused by defects found by customers shortly after the release of software products; these defects would have taken hundreds to thousands of months to detect if the product had been tested on a single machine. Therefore, he concluded that it would be almost impossible to have prevented many of the defects detected in the field.

The study reported here will further this body of knowledge by adding data on GUI defects detected in the field and the impact on customers from an objective viewpoint.

3. Study Design

Many recent defect studies have been performed on systems developed at ABB [4][10]. These studies have been very beneficial to ABB by identifying and motivating software test improvements throughout the development cycle. The results of these efforts have shown measureable improvement in the detection of early defects and a corresponding decrease in time to develop a software release.

The study presented in this paper examines approximately 200 customer-reported defects from two large, deployed, industrial systems. The applications were developed by ABB and are Human Machine Interfaces (HMI) for large industrial control systems. The defects studied represent four separate HMIs. The systems have been deployed for over 10 years and are used by customers around the world to monitor, configure, and control systems in their businesses. The HMIs are developed in C++ and run on the Windows operating system. The GUI objects, such as forms, buttons, and menus, are developed visually through Visual Studio templates. The glue code connecting the GUI objects to the underlying application is developed by hand.

A *customer*, for the purpose of this paper, is defined as any external receiver of the final released software. This includes the end users themselves, as well as any third party integrators or other ABB units that configure, sell, and deploy the software to the field.

The goal of this study is to *improve the overall quality of GUI testing by studying customer-reported GUI defects to assist testers and researchers in creating and evaluating effective GUI test techniques*. Using the Goal Question Metric (GQM) Paradigm [2] the goal for this research can be restated as follows:

Analyze **customer GUI defects**
for the purpose of **understanding**
with respect to **GUI systems**
from the point of view of the **customer / user**
in the context of **industry developed GUI software**.

This research goal is further refined into five research questions to be answered by this study:

RQ1: How do GUI defects impact the customers' use of the software product and its delivered functionality?

RQ2: How do customer GUI defects differ from GUI defects found in the testing phases?

RQ3: Where are customer-reported GUI defects found in the software system?

RQ4: How long do customer-reported GUI defects take to fix?

RQ5: Were these customer-reported GUI defects released in scheduled releases?

In order to answer these five research questions, this study is broken down into three areas: defect impact, defect location, and defect resolution, each of which is described in the following sections.

3.1 Defect Impact

There are many ways a defect can impact a customer's use of delivered software. The defect may cause a loss of functionality in the delivered system. This functionality loss may be total, evidenced by a system crash, or partial, evidenced by the loss of specific functionality. Other defects may not impact the overall functionality but instead are nuisances to the customer, such as a button requiring two clicks to activate or a misspelling of text in a box.

For this study, functionality loss is categorized as major, minor, or cosmetic. A *major* loss of functionality represents the loss of a core function, such as the ability to log in or access remote systems. A *minor* loss of functionality happens when the customer is unable to perform a non-critical action, such as printing a specific screen. Finally, a *cosmetic* defect is defined as a defect where no functionality is lost, but the customer experiences some incorrect behavior, such as an invalid message box, misspelled word, or an incorrect color on the screen.

Even if the functionality of the software is impacted, a workaround may exist to restore that functionality. These workarounds usually involve a set of actions the user can employ to restore the lost functionality. In this study, workarounds are divided into two categories: simple and complex. *Simple* workarounds do not require the user to spend more time accessing the functionality than the original method requires, such as using the print button on the toolbar as opposed to selecting 'File' and 'Print' from the menu system. *Complex* workarounds, on the other hand, do require the user to spend more time to access the original functionality, such as the user having to open a text file, edit values, and restart the system to change values instead of selecting the desired value from a list displayed in the GUI.

The impact of a defect can also be expressed in terms of the number of customers affected. For the purpose of this study, defects were annotated to impact either *one* customer or *many* customers. Due to the nature of the products in this study, customers represent large companies which may have many sites. Therefore, even defects marked as affecting one customer may impact many users at each customer site.

Finally, the impact of a defect discovered by a customer may vary from the impact of defects discovered in testing. Studying the difference between the type of GUI defects detected in-house and those discovered by customers provides more insight into the defects themselves, as well as their impact on the customer.

3.2 Defect Location

The location of the customer-found defects in the software is also examined. For this study, the location of a defect is categorized into widget, property, glue code, or application. A GUI *widget* is a unit of code representing a user control, such as a dialog box. These widgets can contain user-configurable *property* values that customize its appearance and behavior. Example properties include values that affect the text, color, size, and functionality of the widget. *Glue code* is used to connect the GUI widgets with the rest of the software *application*, which implements all of the underlying functionality.

Understanding the location of the customer-detected defects gives insight into the areas of the software that are not properly tested. If the defect caused the system to crash, that information allows testers to prioritize testing for those parts of the system. Because GUI code can make up a large portion of the overall system, it needs to be thoroughly tested. However, most testing efforts do not focus on testing the GUI itself, but rather test the application through the GUI. Our previous work [4] showed that only 20% of the test cases were designed to test the GUI, while the remaining 80% of test cases were designed to test the underlying application through the GUI.

3.3 Defect Resolution

The length of time a customer experiences a defect is also examined in this study. To better understand the impact of a defect on the customer, the total number of days between reporting the defect and receiving a fixed version of the software is calculated. The customer's ability to work around the defect and the number of defects which cause the system to crash are also studied.

Additionally, the type of release is studied and reported. Releases can be *major*, where new functionality is included in the release, *minor*, where only bug fixes are included in the release, or *customer specific*, where the release is only sent to a few customers.

3.4 Threats to Validity

Assessing the impact of each defect is somewhat subjective. We devised the scale discussed in the previous section to aid in consistently characterizing the defects. Where applicable, the scale is also consistent with the Problem Reporting and Correction (PRC) System in use at ABB.

Defect location can be difficult to pinpoint, even after examining changes in the source code. In particular, determining the difference between a defect in the glue code or in the application was sometimes difficult as many systems rely heavily on passing data to other systems.

To assist in consistency across the defects, the authors met and collaboratively characterized the first 5% of the defects. In addition, a random sampling of classified defects was collaboratively reviewed and, if any disagreements were found, any other defects with a similar classification were also reviewed.

The defects analyzed for this study are from large, currently deployed production systems. While they are applicable to a variety of domains, they are primarily control and monitoring systems and therefore the results may not be directly transferrable to systems in other domains, such as office automation applications.

4. Example Customer Defects

To better understand how GUI defects affect the customer's use of the software and its functionality, three example defects from this study are presented in detail. The three defects described here were actually detected by a customer while using the system in the field for its intended purpose.

The first failure occurs when the customer clicks a GUI button twice, causing numeric values to be changed in a text input field elsewhere on the page. The values affected by the extra click cause the software to send incorrect values out to the running control system, which could cause the control system to error and stop production. The underlying defect for this first failure involves the glue code for the GUI event handler. Specifically, the double click event is not handled properly and the second click event gets processed by the text input field, which sets its value to the screen coordinates of the mouse.

Another customer failure is observed when the user changes the default color for a GUI control. The control is displayed with the correct color until its blink functionality is triggered. At this point, the control is supposed to blink between its assigned color and the inverse of its assigned color. Instead, the control just blinks the default black and white and, when the blink functionality is complete, the user specified color is not restored. This defect exists in the GUI widget itself and only has a cosmetic impact on the software functionality.

The third customer failure occurs when customers want to rerun existing reports to repopulate their data. The initial report is created correctly, but when it is rerun, the custom formatting for the report is lost. This defect exists in the glue code for the update report GUI button. It results in a minor loss of functionality and has a complex workaround, which involves recreating the report each time the data changes.

5. Results

Each research question, listed in Section 3, has an associated set of metrics that were collected to provide insight into the problem. These metrics, and their values, are presented here, along with the research question to which they apply.

5.1 Defect Impact

RQ1: How do GUI defects impact the customers' use of the software product and its delivered functionality?

Metrics: *Functionality loss*, *workaround availability*, *size of impact, frequency of crash*

Table 1. Defect Impact

Impact	Percent of Total
Major	22.40%
Minor	43.23%
Cosmetic	34.38%

The defects were classified based on the impact to the customer's ability to use the delivered functionality of the system, which is shown in Table 1. The majority of the defects resulted in a minor impact on functionality (43%), while approximately 35% of the defects were cosmetic in nature and the remaining 22% had a major impact on the functionality.

Table 2. Workaround Availability for Defects

Workaround	Percent of Total
Complex	17.71%
Simple	23.44%
None	58.85%

The customer's ability to work around the defect is also of concern. Table 2 shows that almost 60% of the defects did not have a workaround, while 18% had a complex workaround and the remaining defects (23%) had a simple workaround.

Table 3. Relating Defect Impact and Workaround

Impact	Complex	Simple	None	Total
Major	32.56%	16.28%	51.16%	100%
Minor	19.28%	32.53%	48.19%	100%

For this study, we were also interested in determining if a correlation exists between the impact of the defect and the workaround. The data in Table 3 show that approximately half of the defects, whether a major or minor impact on the customer, did not have a workaround. For those defects with a workaround, those with a major impact had a complex workaround most often, while those with a minor impact frequently had a simple workaround.

Table 4. Breakdown of Defect Impact by Crash

System Crash	Impact			Total
	Major	Minor	Cosmetic	
Yes	70.83%	25.00%	4.17%	100%
No	15.48%	45.83%	38.69%	100%

Although the impact of the defect was determined separately from whether or not it caused the system to crash, the data in Table 4 show that the two are related: of the defects that caused the system to crash, 71% of them had major impact on the customer. Note that it is possible to have a cosmetic defect cause a system crash. An example of this involves a cosmetic defect where a GUI dialog box has an incorrect button enabled. This defect itself results in no loss of functionality, but if the user clicks on that button, the application crashes.

Table 5. Relating Defect Impact to Number of Customers Affected

Impact	Number of Customer Sites		Total
	One	Many	
Major	19.79%	2.60%	22.40%
Minor	33.33%	9.90%	43.23%
Cosmetic	28.65%	5.73%	34.38%

Finally, Table 5 shows that defects with minor impact on the customer also affected the most customers. 33% of the defects were minor and affected only one customer site, while ~10% of the minor defects affected more than one site. Almost 30% of the defects were cosmetic and affected only one site, while 6% were cosmetic and affected more than one. The least number of customers were affected by the defects with major impact, where ~20% of the defects affected one customer and 3% affected many customers.

RQ2: How do customer GUI defects differ from GUI defects found in the testing phases?

Metrics: *Classified defects*

In our previous work, defects were classified according to a modified version of the Beizer Defect Taxonomy. Details of the previous studies as well as the classification scheme can be found in [4] and [10]. Table 6 shows the difference between in-house defects reported for three of the system interfaces studied compared to customer-reported defects. The customer-reported defects make up ~40% of all reported defects. The categories shown are those with a difference of over 1%.

The category of Feature Completeness shows the largest difference in reported defects between in-house (~3%) and customer-reported (~8%) defects. The next category of defects with the largest difference between reporting sources GUI Defects, with a difference of 4.25%. Data Access and Handling showed a difference of 3.57%.

Table 6. Classification of Defects

	Classification	Cust. Reptd	In-house	Diff.
22	Feature Incomplete	7.91%	2.76%	5.15%
53	GUI Defect	30.22%	25.97%	4.25%
42	Data Access/Hndlg	10.79%	14.36%	3.57%
81	System Setup	5.76%	8.84%	3.08%
54	Software Doc	1.44%	3.87%	2.43%
24	Domains	2.88%	0.55%	2.33%
61	Internal Interface	4.32%	6.63%	2.31%
23	Case Complete	3.60%	1.66%	1.94%
62	External Interface	1.44%	3.31%	1.88%
25	User Msg/Diagnos	5.04%	3.31%	1.72%
73	Recovery	0.00%	1.66%	1.66%
71	OS	0.72%	2.21%	1.49%
21	Correctness	1.44%	0.00%	1.44%
72	Sw Architecture	3.60%	2.21%	1.39%
63	Config Interface	0.00%	1.10%	1.10%

5.2 Defect Location

RQ3: Where are customer-reported GUI defects found in the software system?

Metrics: *Location of defects in the system, frequency of crash*

Table 7. Defect Location

Location	Percent of Total
Application	40.22%
Glue Code	32.61%
Property	22.28%
Widget	4.89%

The data in Table 7 show that 40% of the total defects were detected in the application code itself. The glue code (33%) and properties of the GUI (22%) each contained almost as many defects, while only 5% of the defects were in the widget. In total, the GUI itself contained approximately 60% of the defects, while the underlying application contained the remaining 40%.

Table 8. Relating Defect Location to System Crash

	System Crash	
Location	Yes	No
Application	62.50%	36.88%
Glue Code	16.67%	35.00%
Property	16.67%	23.13%
Widget	4.17%	5.00%

Table 8 shows the relationship between defect location and system crashes. The majority of the defects in the application code also caused the system to crash (63%), while defects in the glue code and properties only caused crashes approximately 17% of the time and widget defects cause crashes in less than 5% of the cases.

5.3 Defect Resolution

RQ4: How long do customer-reported GUI defects take to fix?

Metrics: *Time to fix, frequency of crash, workaround availability*

Table 9. Relating Impact and Fix Time

Impact	Avg Days to Fix
Major	244
Minor	156
Cosmetic	139
Total Average:	**170**

Next, the average number of days to fix each type of defect (based on impact, location, crash and workaround) was computed by comparing the date the defect was reported and the date changed code was checked in to the code repository. The date of the actual release for the majority of these defects was preplanned, so the total elapsed time for the customer is less interesting. The defects with major impact took the most time to fix; on average, 244 days, compared to 156 days for minor impact defects and 139 days for cosmetic defects. Table 9 shows the data.

Table 10. Relating Location and Fix Time

Location	Avg Days to Fix
Application	114
Glue Code	239
Property	147
Widget	301
Total Average:	172

Table 10 breaks down the time to fix based on the location of the defect. The widget defects took the most time to fix – 301 days on average. Glue code defects took 239 days to fix, while defects located in property and application code took an average of 147 and 114 days, respectively.

Table 11. Relating System Crash and Fix Time

Crash	Avg Days to Fix
Yes	175
No	169
Total Average:	**170**

Data was also gathered on whether crash-causing defects take more or less time to fix. The average amount of time to fix any defect was 170 days. Table 11 shows that crash-causing defects took 175 days to fix on average, while non-crashing defects took 169 days to fix. There is no real difference between the time needed to fix crash-causing and non-crashing defects.

Table 12. Relating Workaround and Fix Time

Workaround	Avg Days to Fix
Complex	151
Simple	196
None	166
Total Average:	**170**

The average number of days to fix the defect, as it relates to workaround difficulty, was studied and the results are shown in Table 12. Defects with complex workarounds were fixed the fastest (average of 151 days), while the defects with simple workarounds took 45 days longer on average (196 days to fix). Those defects with no workaround took 166 days to fix.

RQ5: Were these customer-reported GUI defects released in scheduled releases?

Metrics: *Previously known or not, type of release*

In most cases, defects were corrected in the next major release of the product (88%). Table 13 also shows that 10% of the defects warranted a customer-specific release, while only 2% were fixed in a minor release.

Table 13. Type of Release to correct defects

Type of Release	Percent of Total
Major	88.02%
Minor	2.08%
Customer-specific	9.90%

6. Discussion and Analysis

The results presented in the previous section provide details of the defects that were studied, including impact of the defect from the perspective of the customer, defect location, and the resolution of the defect. The following sections provide further analysis of these results.

6.1 Defect Impact

Defects were examined on the basis of loss of functionality for the customer as well as available workarounds. Studying the combined impact and workaround of the defects (Table 3) revealed that approximately half of all major and minor impact defects have no workaround. This means that customers who experience functionality loss due to GUI defects often have no way to restore that functionality until a fix is released. Combining that with the fact that major defects take an average of 244 days to fix (Table 9) and are often released in scheduled yearly major releases (table 13), customers are often impacted by this functionality loss for between 8 and 12 months. Also, many ABB customers do not upgrade more than once per year anyway, since upgrades often require a shutdown of production.

Furthermore, relating defect impact to system crashes (Table 4), it can be seen that 71% of the defects that caused a crash had a major impact on the customer. However, the overall percentage of defects that were major and also crashed the system was fairly small, only 9% of the total defects.

Studying the number of customers affected (Table 5) shows an interesting phenomenon. Due to the nature of the software studied, the customer base is much more limited than that of other commercial software, such as office automation software. Previously, it was our assumption that the software is used in approximately the same manner at each site. However, the results in Table 5 disagree and show that less than 20% of all defects are observed at more than one customer site.

Finally, this study showed that the classified defects varied only slightly between those found in-house and those found by customers in the field. This shows that in-house testing is finding the same *types* of defects as customers find, therefore with an increase of targeted testing efforts, perhaps even more defects will be found in-house.

6.2 Defect Location

Examining defect location gave a good indication of how many defects observed through the GUI are actually defects in the underlying code. For the systems studied, the difference between application code, glue code, and property code is not statistically significant; however, these results show that 60% of the defects are located in the GUI while the remaining 40% are found in the underlying application. These results support previous findings that GUI testing reveals underlying application defects, as well as the intuition that defects in the glue code can be difficult to test for directly and can only be found by testing *through* the GUI. Furthermore, of the defects that caused the system to crash, almost 63% of them are due to errors in the application code. This supports the intuition that uncaught application defects can have a major impact on the reliability of the deployed application.

6.3 Defect Resolution

Due to ABB's software release schedule, the majority of the defects were not resolved for almost 6 months, and up to 8 months in some cases. Defects that are high in priority are scheduled for faster, customer specific releases. The products studied release a large percentage of bug fixes in scheduled major and minor releases. In three of the four interfaces studied, almost 90% of the defects were released in scheduled major product releases each year (Table 13).

Tables 9-12 show the impact to the customer, with several interesting findings. First, defects with major impact to the customer take the longest to fix (Table 9). Since higher priority defects are intended to release faster, this may be due to the difficulty of the resolution. Second, defects located in the widget and in the glue code take the most days to fix (Table 10). These two points suggest that either the defects in the glue code and widgets are not perceived to be as important by the developers or those defects are harder to pinpoint and, therefore, fix. Further examination of defect resolution start (instead of defect reported date) and end dates is needed to make any further conclusions. Third, it appears that defects with a simple workaround are also fixed slightly slower; Table 12 shows these defects take

an average of 196 days to fix, compared to the average time of 170 days. This was expected, as defects with simple workarounds are prioritized lower.

7. Conclusions and Future Work

Previous work has shown that GUI testing is important. The purpose of this study was to examine the impact of GUI defects on the customer. We found that the majority of customer-reported GUI defects had major impact on their day-to-day operations, but were not fixed until the next major release. On average, customers waited 170 days (almost 6 months) for these defects to be fixed. Further, previous research also shows that many defects of the underlying application can manifest themselves in the GUI. The findings of this study further support these previous studies, since 40% of the defects observed in the GUI were actually defects in the underlying application code.

These findings provide more detailed information on customer defects than was previously available. In the future, testers in ABB can leverage this information when planning the overall testing strategy of a product release. For example, learning that less than 20% of the defects are seen at more than one customer site shows that it is more important to tie in customer profiles as testing efforts are planned. By testing the system as it is used by several customers, even fewer defects will present themselves at multiple sites.

ABB's software development teams are in the process of making major changes, specifically in the quality assurance area. One of these changes may include a more frequent release schedule to address customer-reported defects sooner. After incorporating the findings of this study into the testing strategy at each development organization, the impact of defects on the customer can be studied again.

References

[1] Adams, E. N. *Optimizing preventive service of software products*. IBM Journal of Research, 28(1), January 1984.

[2] Basili, V. R. *Software Modeling and Measurement: the Goal/Question/Metric Paradigm*. Technical Report. University of Maryland at College Park, 1992.

[3] Brooks, P. A. and Memon, A. M. 2007. Automated GUI Testing Guided By Usage Profiles. In *Proc. of the 22nd IEEE/ACM Int'l Conference on Automated Software Engineering*, Atlanta, Georgia, USA, pp. 333-342, Nov 2007.

[4] Brooks, P., Robinson, B., and Memon, A. M. An Initial Characterization of Industrial Graphical User Interface Systems. To appear in *Proc. of the IEEE Int'l Conf on Software Testing, Verification, and Validation*, Apr 2009.

[5] Clarke, J. M. Automated test generation from a behavioral model. In *Proc. of 11th Int'l Software Quality Week*, May 1998.

[6] Gittens, M., Lutfiyya, H., Bauer, M., Godwin, D., Kim, Y. W., and Gupta, P. An Empirical Evaluation Of System And Regression Testing. In *Proc. of the Conf of the Centre For Advanced Studies on Collaborative Research*, pp. 3-15, 2002.

[7] Memon, A. M., Banerjee, I., and Nagarajan, A. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *Proc of 10th Working Conf. on Reverse Eng*, pp 260-269, 2003.

[8] Memon, A. M. and Xie, Q. *Studying the fault-detection effectiveness of GUI Test Cases for Rapidly Evolving Software.* IEEE Transactions on Software Engineering, 31(10):884–896, 2005.

[9] Musa, J. D., *Software Reliability-Engineered Testing*, IEEE Computer , vol.29, no.11, pp.61-68, Nov 1996.

[10] Robinson, B., Francis, P., and Ekdahl, F. A Defect-Driven Process for Software Quality Improvement. In *Proc. of the Int'l Symposium on Empirical Software Eng and Measurement*, Kaiserslautern, Germany, pp 333-335, Oct 2008.

[11] Sullivan, M. and Chillarege, R., A Comparison of Software Defects in Database Management Systems and Operating Systems, In *Proc. of Int'l Symposium on Fault-tolerant Computing*, pp. 475-484, July 1992.

[12] Xie, Q. and Memon, A. M. *Designing and comparing automated test oracles for GUI-based software applications*. ACM Transactions on Software Engineering Methodology, 16(1):4, 2007.

[13] Yuan, X. and Memon, A. M. Using GUI run-time state as feedback to generate test cases. In *Proc. of the 29th Int'l Conf on Software Eng*, pp. 396-405, May 23–25, 2007.