An Initial Characterization of Industrial Graphical User Interface Systems

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

Atif M. Memon
University of Maryland
College Park, MD
atif@cs.umd.edu

Abstract

To date we have developed and applied numerous model-based GUI testing techniques; however, we are unable to provide definitive improvement schemes to real-world GUI test planners, as our data was derived from open source applications, small compared to industrial systems. This paper presents a study of three industrial GUI-based software systems developed at ABB, including data on classified defects detected during late-phase testing and customer usage, test suites, and source code change metrics. The results show that (1) 50% of the defects found through the GUI are categorized as data access and handling, control flow and sequencing, correctness, and processing defects, (2) system crashes exposed defects 12-19% of the time, and (3) GUI and non-GUI components are constructed differently, in terms of source code metrics.

1. Introduction

Almost exclusively, software today provides a *Graphical User Interface* (GUI) as the only method for users to access the functionality of the software. Hence the GUI is an important part of the software; it can account for as much as 60% of the overall code [19]. Overall system and integration testing of *software under test* (SUT) involves testing *via the* GUI (we'll call this type of testing *GUI testing*).

For GUI testing, the test designer develops test cases, each modeled as a sequence of user events, and executes them on the SUT via the GUI, either manually or automatically. Events include clicking on buttons, selecting menus and menu items, and interacting with the functionality of the system provided in the GUI. 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 "business logic defects" (e.g., the computation result values are incorrect). In order to exhaustively test the GUI, it is necessary to exercise every combination of sequences of the events present in the GUI, which would require an exponential number of tests, based on the test case length (*i.e.,* the number of GUI events) [28].

Due to its importance, GUI testing has received considerable attention by researchers in the last decade. Work on GUI testing has included using capture-replay tools [15], operational profile- [8] or user profile-based methods [5], structural testing [27],[28], and *n*-way event testing [18].

Several researchers have developed new techniques and have demonstrated the effectiveness of these techniques on experimentation subjects, either using toy examples, larger subjects developed *in-house*, or open-source applications. For example, the work by the GUI testing group at the University of Maryland has also undergone an evolution in terms of experimentation subjects. The earliest techniques were demonstrated on a small word-processor consisting of 1 KLOC of C++ code [16],[17]; no "defect studies" were conducted. Later, larger subjects (> 25 KLOC) were developed in-house; a technique known as *fault seeding* was used to purposely seed predetermined classes of faults in these subjects. Although the researchers were careful to create faults that represent mistakes actually made by programmers, by carefully selecting portions of the source code with a considerable amount of test coverage, it remains unclear whether the seeded faults are relevant to GUI testing – the fault/defect classifications were borrowed from non-GUI work. Defining a GUI-fault classification scheme remains an open area for research.

In order to further reduce threats to external validity, the experimentation subjects were enhanced yet again; larger (approx. 100 KLOC) Java applications were taken from sourceforge.net, and adapted as subjects [5],[18],[27],[28]. Because of the difficulties associated with creating test oracles for unfamiliar applications, the test oracles were restricted to looking for *software crashes* only, i.e., a test case failed if it caused the software to crash; otherwise it passed.

Although the previous work has been extremely successful at moving forward the state-of-the-art in model-based GUI testing, it has several weaknesses. First, there is no classification for defects that are observable via the GUI. It is important to understand real GUI failures and create such a classification so that better test oracles may be designed to detect such failures; better test-case generation algorithms may be designed to target the faults that result in these failures; better models for fault seeding may be developed to

improve experimentation. Second, the GUI testing techniques have not been applied to large Industry applications – most have been applied to desktop applications. It remains unclear whether the existing algorithms will scale; whether the code written for the "GUI part" contains serious defects or whether most defects lie in the "non-GUI part"; whether Industry programmers write GUI code differently from non-GUI code; whether testers in Industry write test cases to specifically test the GUI. Finally, although crashes have provided a useful test oracle mechanism for academic experimentation, it is unclear whether such serious failures occur in well-used Industry applications; even if they do, then what percentage of actual failures are crashes?

This paper is our first attempt at making the leap into developing the next generation of GUI testing algorithms that are relevant and applicable to large Industry applications. We first want to enhance our experiments; in this paper, our goal is to better understand fielded Industry applications. We attempt to study the nature of GUI bugs in fielded Industry programs and understand the nature of GUI/non-GUI code. This is the first such study to look into GUI Industry programs.

The bulk of the work reported in this paper has been done at ABB, which has a long history of conducting successful defects studies [22]. These studies have had practical implications – they have been beneficial to developers, testers and managers; they have helped to associate defects with particular phases in the software development process. Test teams have seen dramatic increases in the number of defects they are now able to detect in early phase of testing (within only a few weeks), which has resulted in significant decreases in low-level bug fixes, which were previously common in late-phase test.

The study presented in this paper provides a thorough analysis of the defects discovered in three applications at ABB, focusing on those detected in or through the GUI front-end. In particular, this study was conducted to characterize GUI systems based on artifacts from testing and development, including source code measures, change metrics, and test suite characterization. A modified version of Beizer's taxonomy [1][7] is used to classify the defects.

The contributions of this work include:

- An empirical study of three large, deployed, industrial applications
- A profile of 1,215 defects detected in these applications, classified using Beizer's defect taxonomy
- A characterization of GUI systems, involving a study of test cases, faults found, source code metrics, and source change metrics

This paper is organized as follows: Section 2 describes the research design for this study, including a description of the systems used, the defect classification scheme, and the method of defect classification. Section 3 presents the results, and Section 4 presents discussion and analysis. Section 5 presents related work in the areas of GUI testing, defect classification schemes, and other defect classification studies. Finally, Section 6 concludes with a discussion of future work.

2. Study Design

The goal of this study is to *improve the overall quality of GUI testing by characterizing GUI systems using data collected from defects, test cases and source code to assist testers and researchers in developing more effective test strategies*. Using the Goal Question Metric (GQM) Paradigm [3], the goal for this research is restated as follows:

Analyze the **defects, test cases, and source metrics**
for the purpose of **understanding**
with respect to **GUI systems**
from the point of view of the **tester / researcher**
in the context of **industry-developed GUI software**.

This research goal is broken into four research questions to be answered by the study:

RQ1: How do defects in the GUI differ as compared to overall defects in the SUT? What kinds of defects are commonly found through the GUI?

RQ2: How many defects in GUI applications are detected through crashes, as compared to observed program deviations?

RQ3: How do GUI components compare to non-GUI components with respect to source metrics? How do source changes in GUI components compare to changes in non-GUI components?

RQ4: What are the characteristics of the test suites? How many of the tests are testing the GUI compared to testing the application through the GUI?

In determining the study setting, several key factors were decided, such as the defect taxonomy that would be used, how construction metrics could be used to characterize GUI systems, and how test results could be leveraged for the GUI characterization. Each of these factors is described in the following sections.

2.1. Choosing and Tailoring a Taxonomy

Beizer's taxonomy [1] divides defects into eight main categories, each describing a specific set of defects. Each main category is further refined into three levels of subcategories. A defect is then assigned a four digit

number with each digit representing the selected category or subcategory. For example, Processing Bugs would be 32xx, where the 3 designates a structural defect and the 2 places this defect into the processing subcategory. The last two numbers, shown as x here, refine the defect to more levels of detail. Beizer's taxonomy includes four levels of categories for each defect.

The first category of the taxonomy is for **Functional defects**, *i.e.*, errors in the requirements, including defects caused by incomplete, illogical, unverifiable, or incorrect requirements. The second category, **Functionality as Implemented**, deals with defects where the requirements are known to be correct but the implementation of these requirements was incorrect, incomplete, or otherwise wrong. The next two categories, **Structural Defects** and **Data Defects**, are used for low-level developer defects in the code, such as problems with control flow predicates, loop iteration and termination, initialization of variables, incorrect types, and incorrect manipulation of data structures. Another category of defects classifies **Implementation** errors. These are errors dealing with simple typographical errors, standards, or documentation. The next category is for **Integration defects**, representing errors in the internal and external interfaces in the software. Finally, the last two categories of defects deal with **System** and **Test defects**. System defects comprise errors in the architecture, OS, compiler, and failure recovery of the system under test. Test defects represent errors found in the test descriptions, configurations, and test programs used to validate the system.

Beizer's taxonomy was chosen for this study, based primarily on the categories themselves and the fit within the ABB environment. Currently, testing at ABB is not based on a test strategy, and therefore ODC was not chosen since it relies on a test strategy and process being in place. Other object oriented taxonomies were not chosen since the development of these applications is not strictly object oriented, although the language has the capability.

After selecting Beizer's taxonomy, all of the categories and subcategories were analyzed. A two level approach was selected, with only the main category and one subcategory used, due to the needs of ABB. Because this taxonomy was tailored for initial work with developers and testers within ABB, a few subcategories were renamed, giving them names more similar to those used inside ABB [22].

In addition to the existing categories in the taxonomy, a few additional subcategories were added for specific defect types due to their importance to ABB. The first additional subcategory was named "GUI defects," and assigned to the Implementation main category as 53, to categorize defects that exist either in the graphical elements of the GUI or in the interaction between the GUI and the underlying application API. These defects were given their own defect type since code involved in the GUI is treated differently than the underlying application code in many companies, and require different testing steps to validate.

Table 1. Beizer's Taxonomy (Modified)

1xxx	*Functional Bugs: Requirements and Features* 11xx Requirements Incorrect 12xx Logic 13xx Completeness 14xx Verifiability 15xx Presentation 16xx Requirements Changes
2xxx	*Functionality As Implemented* 21xx Correctness 22xx Completeness – Features 23xx Completeness – Cases 24xx Domains 25xx User Messages and Diagnostics 26xx Exception Conditions Mishandled
3xxx	*Structural Defect* 31xx Control Flow and Sequencing 32xx Processing
4xxx	*Data Defect* 41xx Data Definition, Struc, Declaration 42xx Data Access and Handling
5xxx	*Implementation Defect* 51xx Coding and Typrgraphical 52xx Standards Violations 53xx GUI Defects 54xx Software Documentation 55xx User Documentation
6xxx	*Integration Defect* 61xx Internal Interfaces 62xx External Interfaces 63xx Configuration Interfaces
7xxx	*System and Software Architecture Defect* 71xx OS 72xx Software Architecture 73xx Recovery and Accountability 74xx Performance 75xx Incorrect diagnostic 76xx Partitions and overlays 77xx Environment 78xx 3rd Party Software
8xxx	*Test Definition or Execution Bugs* 81xx System Setup 82xx Test Design 83xx Test Execution 84xx Test Documentation 85xx Test Case Completeness

The next change to the taxonomy involved splitting the documentation subcategory into two categories, one for classifying in-software documentation errors and one for user documentation errors. These were labeled 54 and 55, respectively. In-software documentation defects cover missing or incorrect developer documents, such as design

documents, *i.e.*, software models like UML, or internal code documentation, *i.e.*, comments in the code. User documentation deals purely with defects that exist in documents that are released to the customer with the software, such as product installation and user manuals.

The taxonomy was further modified to include a subcategory to classify defects in system setup. This category allows classification of defects dealing with configuring the system correctly for its intended use. Since all of ABB's systems are highly configurable, these defects are important enough to track separately. This subcategory was added to the Test Definition or Execution Bugs category, and labeled as 81.

Finally, a subcategory was added to categorize defects in the configuration interfaces that are available in the system. Since these systems have so many possible executable configurations, each of which highly impact how the system executes, the interfaces which allow this configuration to occur require their own classification. This new defect type was added to the taxonomy as 63. The modified version of Beizer's taxonomy is shown in Table 1.

2.2. Gathering Construction Metrics

In order to compare the construction of GUI and non-GUI components of the system, the source code files for one system were split into two groups: files implementing the interface of the system and files implementing the remaining functionality of the system. To aid in characterizing the systems, each of these groups was analyzed separately. Files were determined to be part of the GUI if they contained code that implemented a GUI action, *i.e.*, a button click, menu click, window open or window hide event.

Source code metrics were collected using Source Monitor[1] for one SUT, and the results were divided into the GUI and non-GUI groups, based on the label of the corresponding file or class. Five source metrics were selected to represent measures of size, complexity, coupling, and developer documentation. These include lines of code (LOC), LOC per method, percentage of lines with comments, cyclomatic complexity (CC), and call depth. In addition to source code metrics, source code changes were derived by computing the difference for each metric between versions of the system. Although source code was only available for one SUT, it is the largest in the study, containing over ~1.6 million lines of code.

[1] http://www.campwoodsw.com/sourcemonitor.html

2.3. Collecting Test Suite Data

The ABB product group defect repositories that were mined for this study do not uniformly contain information on whether or not the defect was detected through a crash. Therefore, to gather information on whether the system crashed to expose each defect, natural language queries were run on the text fields of the repositories, such as Title, Description, Evaluation, and Implementation Notes. These queries were stored in Perl and shell scripts to support repeatability. Words used in the query include "access violation," "ACCESS_VIOLATION," "crash," "hang," "freeze," and "froze." This natural language query was reinforced by human data checking of the defect reports. Determining how many of the reported defects were detected through crashes will greatly assist in the characterization of GUI systems.

Metrics pertaining to the test suites were also collected to determine characteristics of industrial test suites used for GUI systems. All three systems studied use manually executed tests, and one product augments this manual testing with a large suite of automated GUI tests. The manual test suite was only available for one of the SUTs and the automated tests were not available to include in this study. From the manual test suite and its results, metrics were gathered for the size of the test suite, the number of test cases used for crash testing, the method used to generate the test suite (functional, logical, or state), and the number of tests with validation points. Validation points, points between test steps where the state of the application is checked, are often implied in manual testing, as the human tester can visually check the state of the system. For this study, validation points were determined from the test case and the results of the test case execution.

2.4. Applications

The three applications chosen for this study were developed by ABB and are all Human Machine Interfaces (HMI) for large industrial control systems. They allow the user to monitor, configure, and control various aspects of the running system. These systems 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 and is the only portion of the GUI studied here. These applications were selected because they are large, deployed applications that have been running in the field for over 10 years by hundreds of customers around the world.

2.5. Collecting GUI System Data

Defect data is contained in several ABB data repositories containing Software Problem Reports (SPRs). Specifically, the defects of interest are those found in late-phase testing and by customers after release. Each SUT has a separate repository and one of the systems in this study implemented their repository in a completely different method than the other two systems. However, all three repositories contain roughly the same data items, and all of the data needed for this study was available for all of the systems.

Each SPR indicates when the defect was found, what version of the software was running, and the severity of the defect. Defect severity is assigned on a 5-point ordinal scale ranging from Low to Project Stopper. For this study, defects in the top three points of the severity scale were classified, since the management team has determined that the cost of discovering lower severity defects in the field can be tolerated.

Due to limited data query support in these repositories, the SPRs were saved in text files and parsed using a combination of manual effort (*i.e.*, members of the research team read the documents) and Perl scripts. After gathering the data into a format that could be reasoned about, the analysis was conducted. The SPR data used for this study represents three years of development and two major versions of the products.

2.6. Threats to Validity

The defect classification was performed by several people over the course of one year. Due to this, it is possible that the same type of defect was categorized differently by different people. However, to mitigate this risk, the team classifying the defects met each week and selected random groups to reclassify together. If issues were found, others with similar classifications were also discussed.

Crash data was mined from the defect repositories using natural language queries (see Section 2.3). These queries pose a risk of missing data due to the imprecise nature of natural language. To decrease the number of missed crashes, the queries included the synonyms and several misspellings of each search term. In addition, a set of random groups of defects were selected and one of the authors manually determined if the defect was a crash or not. If an uncaught crash was detected, additional keywords were added to the query and it was rerun. The crash results were also checked for false positives, but none were found.

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 desktop or productivity applications.

3. Results

From the original goal presented in Section 1, a set of research questions was developed for this study. Each research question 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.

3.1. Overall Defects

RQ1: How do defects in the GUI differ as compared to overall defects in the SUT? What kinds of defects are commonly found through the GUI?

Metrics: *Defect classification by type, software lifecycle phase of defect detection*

Overall, 1,215 defects from three different GUI systems were studied and classified into the taxonomy. This list only includes defects with a severity of High, Critical, and Project Stopper (on a 5-point scale) that were found in late testing phases or by customers in the field. These represent the defects that are most often fixed and included in later releases. Table 2 shows the number of the GUI and non-GUI defects found in the systems. The combined classification data for these defects is shown in Table 3, ordered by defect rank. The most common defects were in data access and handling (15.47%) and control flow and sequencing (12.67%). Out of the 27 defect classes, the top 4 classes accounted for approximately 50% of the defects. GUI defects ranked fifth overall.

Table 2. GUI and Non-GUI Defects per SUT

	Number of Defects Found	
System	*GUI*	*Non-GUI*
SUT1	7	154
SUT2	11	727
SUT3	90	226

RQ2: How many defects in the GUI applications were detected through crashes, as compared to observed program deviations?

Metrics: *Number of defects detected by software crash, number of defects detected by observed program deviations*

Due to our ability to use a primarily automated method for determining crashes, rather than the manually

intensive method used for classifying defects in RQ1, we analyzed a total of 3,869 defects, encompassing all five severity types and all three systems under test. Processing the natural language query described in Section 2.3 produced the following crash results: SUT1 had 248 crashes out of 1,661 defect reports; SUT2 had 372 crashes out of 1,892 defect reports; and SUT3 had 37 crashes out of 316 defect reports. Therefore, crashes accounted for 15%, 19% and 12%, respectively, of the defects detected**.**

Table 3. Defect type across all systems

Defect Class	Fault Type	% Defects
42	Data Access, Handling	15.47%
31	Control Flow, Sequencing	12.67%
21	Correctness	11.69%
32	Processing	10.37%
53	GUI	8.89%
81	System Setup	5.76%
23	Part. Implemented Features	4.77%
41	Data Def., Struc, Decl.	4.53%
72	Software Architecture	3.95%
22	Unimplemented Features	3.62%
26	Error Handl., Missing, Incorr.	3.37%
25	User Messages and Errors	2.63%
61	Internal Interfaces	2.63%
55	User Documentation	1.98%
75	Third Party Software	1.48%
71	OS and Compiler	1.23%
54	In-Software Documentation	0.82%
74	Performance	0.82%
62	External Interfaces	0.74%
24	Domains	0.66%
51	Coding and Typological	0.58%
83	Test Execution	0.33%
63	Configuration Interfaces	0.25%
73	Recovery	0.25%
82	Test Design	0.25%
16	Requirements Changes	0.16%
52	Standards Violation	0.08%

3.2. Construction

RQ3: How do GUI components compare to non-GUI components with respect to source metrics? How do source changes in GUI components compare to changes in non-GUI components?

Metrics: *File changes, average statements per method, number of statements changed, number of lines changed, percentage of commented lines, average complexity, average block depth*

Construction was investigated for one of the SUTs, due to the availability of its code. After dividing the source code into two groups, the GUI and non-GUI portions of the system (Section 2.2), their source metrics were calculated. The two groups of measures were compared using a *two sample student t-test* assuming unequal variances to compare the means of the two groups. The hypothesized difference in means was zero. This test was selected since the number of observations to compare was ~15000. The results of the source metrics analysis shows that there are statistically significant differences (at α=0.05) between the GUI and non-GUI components for all five of the metrics selected ($p < 0.05$). Table 4 contains the computed metrics for the system used in this part of the study.

Table 4. Source code metrics

Metric	*GUI*		*Non-GUI*	
	Mean	**StdDev**	**Mean**	**StdDev**
LOC	388.27	516.49	248.33	14542.59
LOC / Method	13.10	6.77	3.57	8.93
% Comments	11.54	11.92	20.40	17.09
Complexity	3.58	1.63	2.05	5.01
Call Tree Depth	1.47	0.45	0.82	0.72

Table 5 shows the code changes in the GUI and non-GUI portions for five versions of one SUT. The table shows that the mean of the number of statement changes between versions for GUI and non-GUI are similar, but the standard deviation is significantly larger for the non-GUI parts of the system.

Table 5. Statement changes

Version	*GUI*		*Non-GUI*	
	Mean	**StdDev**	**Mean**	**StdDev**
V1 - V2	77.17	119.27	67.32	1016.74
V2 - V3	77.92	150.01	94.70	2251.50
V3 - V4	38.88	76.54	61.57	948.73
V4 - V5	15.36	73.33	16.28	170.40

Figure 1 shows the percentage of change to the GUI and non-GUI portions of the SUT for the five versions studied. On average, 8% of the changes were to GUI portions and 92% were to non-GUI portions of the system. The overall size of the system is ~1.6 MLOC, of which the GUI portion of the system contains ~200 KLOC (14%) and the non-GUI portion contains ~1.4 MLOC (86%).

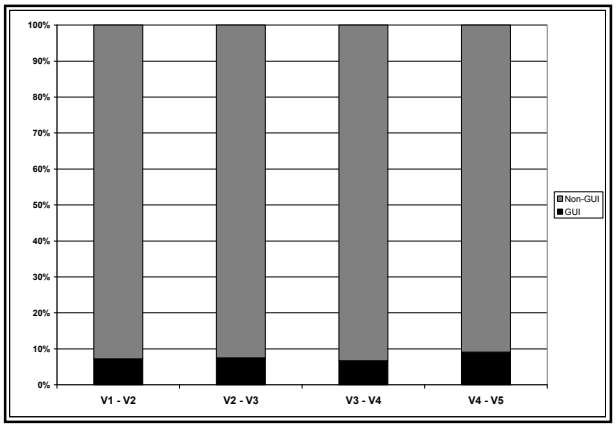

Figure 1. Changes to GUI and non-GUI code

3.3. Test Suite

RQ4: What are the characteristics of the test suites? How many of the tests are testing the GUI compared to testing the application through the GUI?

Metrics: *Number of test cases with validation points, number of test cases used for crash testing, size of test suite, method used to generate test suite (functional, logical, state)*

For this study, the current product testing suite for one of the SUTs was analyzed. This suite is executed manually and takes approximately three man weeks to complete. It contains 800 test cases in total. Of these, 42% contain specific verification points, 50% are only looking for crashes, and the remaining 8% contain general statements of what the correct behavior should be. 20% of the tests were designed to test the GUI itself and the remaining 80% were designed to test the application through the GUI. The main testing methods used in the suite include creating tests for the general cases (80%), error cases (8%), boundary values (10%), and state or combinatorial testing (2%).

4. Discussion and Analysis

This section presents further analysis of the metrics data presented in Section 3, providing some insight into the data gathered to answer the research questions.

4.1. Overall defects

The defect data presented in Section 3.1 provides an industry defect profile of three large deployed GUI applications. Table 2 provides a distribution of defect types that were found through the GUI during late phase testing and after deployment. A large portion of the defects found were categorized as data access and handling, control flow and sequencing, correctness, and processing, representing 50% of the total defects.

Defects in the GUI itself represented only ~6% of the total defects found across the three systems. Looking at the individual systems, Table 2 shows that less than 5% of the defects for two systems were GUI defects, while the third system had almost 30%. Due to the fact that all of these defects were detected through the GUI, it is surprising that more of the defects are not related to the GUI. Many of the problem reports studied during data collection described incorrect system behavior that was observed through the GUI. However, the actual defect often resided in the underlying system components rather than the GUI itself. The assumption that the problem would be rooted in the GUI was primarily due to the limited observability into the system that the GUI provides.

Upon examination of the defects detected in the systems, most required observability of program deviations and a knowledge of the expected behavior of the systems. Conversely, few defects resulted in a crash of the running systems. This also highlights the need for good observability into the system when it is tested through its GUI.

4.2. Construction

The results shown in Section 3.2 indicate that there is a significant difference in the source code metrics when the GUI and non-GUI components of the system are compared. On average, GUI components are larger than their non-GUI counterparts. The percentage of the code that is commented in the GUI components of the system is much lower than that of the non-GUI components. The GUI code is nearly two times more complex than the non-GUI code. Finally, the depth of the call tree is much larger in the GUI portions of the code.

These four measures, taken together, may indicate that developers do not create GUI components with the same discipline that they use when creating the rest of the system. GUI components also contain glue code which links the GUI events to their respective API calls in the underlying system. This extra step may cause some of the extra size, complexity, and call depth.

Section 3.2 also investigates the difference in source code changes between the GUI and non-GUI portions of the system. The results show that the average number of changes is similar between GUI and non-GUI components. However, there are significantly more non-GUI components, leading to a much larger overall number of changes to the non-GUI portion of the system, as shown in Figure 1.

4.3. Test Suites

The data presented in Section 3.3 shows that the majority of the test cases (80%) are intended to test the application through the GUI, while the remaining 20% are intended to test the GUI itself. Characterizing the test suite based on how often verification points are included in test cases, 50% of the written tests only involve verification that the system doesn't crash when the test case is executed. 42% of the tests contained specific criteria for the tester to verify through the GUI that the system performed as expected. Finally, upon examining characteristics of the test suite, it was determined that most of the test suite was executing test cases solely based on the general case of the system (80%), seldom applying additional test methodologies such as boundary checking (10%), state-based testing (2%) and checking error conditions (8%), methods seen as good testing practices.

These findings could be due to employing the strategy of testing the application through the GUI rather than testing the business logic of the application separately from the GUI. As testers are focused on the GUI, and the observability into the system that it provides, the ability to understand and verify the behavior of the underlying system is compromised. This lack of observability into the system often inhibits the testers from using additional test design methodologies, such as testing boundary conditions and checking error conditions, since that level of observability is not available.

5. Related Work

We have been unable to find any research papers specifically characterizing industrial GUI applications through studying their defect profiles and source code metrics. There are, however, several other areas of research related to the classification of GUI defects, including research on GUI testing, defect classification schemes, and case studies of defect classification.

5.1. GUI Testing

Research in GUI testing has focused on structural and model-based methods [5],[27],[28]. Research related to structural methods has focused on testing the GUI based on interactions between events. Test cases were generated based on knowledge of neighboring GUI events that could create a legal event sequence [27],[28]. Initial research involved determining how to glean the structure of the GUI from an existing application [15], which then allowed later work to test the system based on known relationships between events. First, test cases based on the structure of the GUI, called *smoke tests*, were exhaustively generated of length *n*, with *n* from 1 to 5, and they detected some faults [27]. This work also showed that as test case length increased, so did fault detection. Next, intervening events in the structure of the GUI were hidden for test case generation, and test cases were generated using *n* neighboring events, with *n* from 1 to 3. The hidden events were replaced after the test cases were generated, resulting in longer test cases which detected more faults than the smoke tests [28]. However, the problem of the rapid increase in the number of test cases generated as the length increases was still a problem.

Model-based methods for GUI testing have been based on operational models [7],[30], behavioral models [9], and graph models [5],[26]. Operational models provide insight into how the SUT will be used in the field, and guide testers in developing test cases that mimic actual usage and in determining criteria for test completeness [30]. Graph models can be traversed in a variety of ways to generate test cases, but cannot be traversed exhaustively. Therefore, most test suites generated from graph models build test cases by traversing paths of a particular length or combining several paths through the graph [26]. Finally, a study was performed to synthesize several approaches and assist in determining the most appropriate testing strategy based on the type of faults detected [25].

Because previous research has shown that GUI testing can reveal faults in the underlying code [18], the research presented here is focused on studying artifacts from GUI system development and testing to facilitate a characterization of GUI systems leading to better testing schemes.

5.2. Defect Classification Schemes

Several taxonomies exist for classifying software defects, including those described in [10], [21] and [23]. One of the best known taxonomies is presented by Boris Beizer [1]. Beizer's taxonomy has eight categories of defects: *requirements*, *implemented functionality*, *structural*, *data*, *implementation*, *integration*, *system and software architecture*, and *setup and test*. Each defect category is further refined into three levels of subcategories, allowing defect classification to be very precise.

While these taxonomies are designed to be generally applicable, other taxonomies are more specialized. Binder [2] describes one that has been specifically tailored for object-oriented programs, whereas Vijayaraghavan and Kaner [29] focus on eCommerce applications. Knuth describes a more course-grained schema based on the errors found in the TEX typesetting system [12]. Another taxonomy was developed for faults

in user requirements documents [23], and still others discuss hierarchies of faults present in Boolean specifications [11], [14].

The IEEE Standard Classification for Software Anomalies [10] presents a process for handling and resolving software defects as well as a taxonomy for classifying them. This classifies both the source of the defect (i.e. Specifications, Code, etc.) and the type of defect (e.g. Logic Problem). However, it is not as detailed as the taxonomies mentioned above.

In contrast to the taxonomies described above, Orthogonal Defect Classification (ODC) [7] allows practitioners to categorize defects according to their type and tie each one to a phase in the software development cycle where the defect could have been caught, generally with less impact on the software product. ODC has eight defect types: function, interface, checking, assignment, timing/serialization, build/package/merge, documentation and algorithm. These are associated with nine stages of software development: *design, low-level design, code, high-level design inspection, low-level design inspection, code inspection, unit test, functional test,* and *system test*. ODC uses fewer, more general defect categories than other schemes and is primarily focused on process improvement, rather than statistical defect modeling.

5.3. Defect Classification Studies

This study is not the first to classify defects. Three case studies were presented by IBM in 2002, which illustrated the success of characterizing defects in improving software testing strategies for large, deployed projects [6]. As part of the case studies, periodic assessments became part of the software process, which allowed the organizations to better see the cause of defects, thereby allowing them to change their processes early and prevent late-phase defects.

Other research has involved manual examination of code post-development. Using code inspections and the associated change history on software developed for an undergraduate course in high performance computing, another study relied on manual efforts to document defects, classify the defects based on a six-category classification scheme, and then develop hypotheses on how each defect type could be avoided [19].

None of the previous studies have examined and classified only defects found in or through the GUI, nor have they classified defects on user-driven applications with a GUI front-end. Therefore, by performing this study in which the focus is GUI applications, more knowledge can be attained for this particular class of systems.

6. Conclusions and Future Work

This paper presented the results of a study into characterizing GUI systems and their test suites to add to the knowledge base for testers and researchers alike as they determine better methods to test GUI systems. Traditionally, GUI testing has relied on crash testing, due to several factors, including difficulty in developing test suites that adequately cover the breadth and depth of the system as well as the need to observe the underlying system's behavior [27]. The results of this study further show the correlation between the test suite design and the defects detected by crashes. The study also reinforced the idea that the underlying code is often tested *through* the GUI due to the window it provides into the system's behavior. However, the results of this study also show that more visibility is needed so that more of the defects currently detected only by user observation can be detected before the system is released to the field.

The results also suggest that developers treat GUI components differently than other code, applying less formal software engineering practices to the development of GUI code. Stemming from this difference, GUI components are often larger, contain more methods, and contain less comments than the non-GUI components of the systems. Despite the best efforts of researchers to stress the importance of well-developed and well-tested GUI components, it seems developers do not apply the same rules to their development.

Possibly the most interesting finding in this study is the high percentage of defects found through the GUI that are actually defects in the underlying business logic of the system. While this has been shown in open source systems in past research [5],[18],[27],[28], it is interesting that this finding holds in testing industry systems as well. It is another indication that it is important to perform GUI testing – both to test the system through the GUI and to test the GUI itself.

This study is an initial characterization of GUI systems, and several steps succeed it. First, this study looked at three systems driven by system events. In future work, another class of systems could be studied, such as those driven by user interactions. After further characterizing GUI systems based on the same criteria presented here, it may then be possible to develop a methodology for generating test strategies based on the characterization. Additionally, the findings presented here can be applied to the software development process for each of the product groups to determine the impact of GUI system characterization on the effectiveness of testing for future releases.

Acknowledgements

Two graduate students from NC State University, JeeHyun Hwang and Dright Ho, assisted with mining and classifying the defects used for this study. ABB product teams provided access to defects, source code, and test suites.

This work was partially supported by the US National Science Foundation under NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421.

References

[1] Beizer, B. Software Testing Techniques. Van Nostrand Reinhold Co., 1990.

[2] Binder, R. 2000. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, Reading, MA.

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

[4] Broeckers, A., Differding, C., Threin, G., and Bosch, R. 1996. The Role of Software Process Modeling in Planning Industrial Measurement Programs. In *Proc. of the 3rd Int'l Symposium on Software Metrics: From Measurement To Empirical Results* (March 25 - 26, 1996). METRICS. IEEE Computer Society, Washington, DC, 31.

[5] Brooks, P. A. and Memon, A. M. 2007. Automated GUI Testing Guided By Usage Profiles. In *Proc. of the Twenty-Second IEEE/ACM Int'l Conference on Automated Software Engineering* (Atlanta, Georgia, USA, Nov 5-9, 2007), 333-342.

[6] Butcher, M. Munro, H. and Kratschmer, T. "Improving Software Testing via ODC: Three Case Studies," *IBM Systems Journal*, 41(1):31-44, 2002.

[7] Chillarege, R. 1996. Orthogonal defect classification. In *Handbook of Software Reliability and System Reliability*, M. R. Lyu, Ed. McGraw-Hill, Hightstown, NJ, 359-400.

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

[9] Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., Horowitz, B. M.. Model-based testing in practice. In *Proc. of 21st Int'l Conf on Software Engineering*, 1999, 285-294.

[10] IEEE. IEEE standard classification for software anomalies. *IEEE Std 1044-1993*. 2 Jun 1994.

[11] Kaner, C., Falk, J. and Nguyen, H.Q. 1993. *Testing Computer Software (2nd Ed.)*. Van Nostrand Reinhold.

[12] Knuth, D. E. 1989. The errors of TEX. *Software-Practice and Experience.* (19)7:607-685, 1989.

[13] Kuhn, D. R. 1999. Fault classes and error detection capability of specification-based testing. *ACM Trans. Software Engineering Methodology* (8)4:411-424, 1999.

[14] Lau, M. F. and Yu, Y. T. 2005. An extended fault class hierarchy for specification-based testing. *ACM Trans. Software Engineering Methodology* 14(3):247-276, 2005.

[15] 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*, 2003, 260-269.

[16] Memon, A. M., Pollack, M. E., and Soffa, M. L. "Using a goal-driven approach to generate test cases for GUIs" In *ICSE '99: Proceedings of the 21st Int'l Conf. on Software engineering*, 1999, pp. 257-266.

[17] Memon, A. M., Soffa M. L., and Pollack, M. E. "Coverage criteria for GUI testing" In *ESEC/FSE-9: Proc. of the 8th European Software Engineering Conf. held jointly with 9th ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering*, 2001, pp. 256-267.

[18] 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.

[19] Myers, B.A. *User interface software tools*. ACM Trans. on Comput.-Hum. Interact., 2(1):64–103, 1995.

[20] Nakamura, T., Hochstein, L., and Basili, V. R. 2006. Identifying domain-specific defect classes using inspections and change history. In *Proc. of the 2006 Int'l Symposium on Empirical Software Eng.*, Rio de Janeiro, Brazil, 346-355.

[21] Ostrand, T. J. and Weyuker, E. J. 1984. Collecting and categorizing Software error data in an industrial environment. *Journal of Sys. Software.* (4)4: 289-300, 1984.

[22] Robinson, B., Francis, P., Ekdahl, F. A Defect-Driven Process for Software Quality Improvement. In *Proc. of the 2nd Int'l Symposium on Empirical Software Eng and Measurement*, Kaiserslautern, Germany, October 9-10, 2008, 333-335.

[23] Schneider, G. M., Martin, J., and Tsai, W. T. An experimental study of fault detection in user requirements documents. *ACM Trans. on Software Engineering Methodology* (1)2:188-204, 1992.

[24] Schneidewind, N. F. and Hoffmann, H. An Experiment in Software Error Data Collection and Analysis. *IEEE Trans. on Software Engineering* (5,)3:276-286, 1979.

[25] Strecker, J. and Memon, A. Relationships between Test Suites, Faults, and Fault Detection in GUI Testing. In *Proc. of 2008 Int'l Conf on Software Testing, Verification, and Validation*, 12-21.

[26] Whittaker, J.A. and Thomason, M.G. A Markov chain model for statistical Software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, 1994.

[27] 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.

[28] Yuan, X. and Memon, A. M. Using GUI run-time state as feedback to generate test cases. In *ICSE'07, Proc. of the 29th Int'l Conf on Software Eng*, May 23–25, 2007, 396-405.

[29] Vijayaraghavan, G. and Kaner, C. "Bugs In Your Shopping Cart". http://www.testingeducation.org/a/bsct.pdf

[30] Weyuker, E. J. 2003. Using operational distributions to judge testing progress. In *Proc. of the 2003 ACM Symposium on Applied Computing* (Melbourne, Florida, March 9-12, 2003), 1118-1122.