Visual Testing of Graphical User Interfaces: an Exploratory Study Towards Systematic Definitions and Approaches

Ayman Issa[1], Jonathan Sillito[1], and Vahid Garousi[2]
[1]Computer Science Department, [2]Department of Electrical and Computer Engineering,
University of Calgary,
Calgary, Canada
{issaa, sillito, vgarousi}@ucalgary.ca

***Abstract*—Graphical User Interface (GUI) testing literature emphasizes testing a system's functionality through its GUI, rather than testing visual aspects of the GUI itself. In this paper we introduce the notion of visual testing as a subset of GUI testing. To explore visual testing, we have conducted a study of defects in four open source systems. We found that visual defects represent between 16% and 33% of reported defects in those systems. Two categories of visual defects are identified with six subcategories within each of them. Other findings are also reported that are aimed at motivating the importance and the need for systematically conducting visual testing among researchers and practitioners.**

Keywords-graphical user interface; visual testing; visual defect.

I. INTRODUCTION

Graphical User Interface (GUI)-based software systems represent more than 60% [5] of software systems being developed today. One of the main motivations behind the use of GUI front ends is that they promote easier and natural interaction between the system and its users.

Testing GUIs involves validating the GUI visual layout and the system's functionality accessible through it. The GUI testing literature [6,8,11,12] is rich in different techniques and approaches that are focused on GUI based functional testing. Although a GUI's visual layout highly affects the end user interaction and acceptance of a given system, it has not received due attention in the literature. In this paper, the term Visual Testing (VT) is used to refer to that part of GUI testing concerned with testing the visual properties of GUI elements and Visual Defect (VD) is the term that will be used to refer for those defects reported by this type of testing.

Practitioners seem to be doing good job in visual testing of successfully delivered GUI based software systems. However, the context of visual testing being practiced is not clearly specified. The role GUI plays in modern software systems and the black-box context of visual testing being practiced have motivated the authors to study the visual context of software systems as a first step towards systematic definitions and approaches for this type of testing. This has been achieved in three phases. In the first phase, we analyzed a number of GUI based software systems so as to define the visual context of GUI based software systems. The second phase was dedicated to validate the proposed definition in the first phase. The third phase of this research was devoted to collect and analyze visual defect reports and metrics so as to study the nature and implications of visual defects in software GUIs.

The remainder of this paper is structured as follows. Section II surveys the related literature. Visual testing and visual defects are defined in Section III. Section IV describes the setup of the case study performed to validate the proposed definitions. The findings of the case study are presented in Section V. Threats to validity of this research are discussed in Section VI. Finally, the conclusions and future works are outlined in Section VII.

II. RELATED WORK

Mahajan and Shneiderman [10] touched one aspect of visual testing: consistency. Consistency is one factor in the usability of an application. Their experiment showed that an inconsistent GUI could slow user performance between 10% and 25%. Their tool, SHERLOCK, covers consistency checking for dialog boxes and buttons GUI widgets only. SHERLOCK has extra subsystems to check spelling and term use consistency.

Tamm [13] tool that is specialized in capturing layout faults in a specific browser. The tool reads web pages DOM and visual information and compares multiple screenshot taken for a specific page to a visual baseline so as to spot GUI faults. Currently, Tamm's library covers five visual defects: *DetectInvalidImageUrls*, *DetectNeedsHorizontalS-crolling*, *DetectTextNearOrOverlappIngHorizontalEdge*, *D-etectText NearOrOverlappingVerticalEdge*, and *DetectText-WithToo- LowContrast*.

In their recent Cross Browser Incompatibilities (XBI) testing tool (CROSSCHECK), Choudhary et al. [6] identified two levels of differences: screen and trace. Screen level differences are further divided into two levels: visual and DOM. Five visual XBIs are considered in CROSSCHECK: size difference ratio, displacement, area, leaf DOM text difference, and image distance.

CROSSCHECK is useful in identifying visual differences between browsers but not in uncovering VDs. This is attributed to the fact that it tries to find XBIs for a web page displayed in two browsers by comparing the properties of the two web pages and their embodied objects. Thus, a VD will not be reported if it is identical in both browsers.

In our work, we aim to define a framework for visual testing that is missing from the related work reported above. This should pave the way for further research and investigation for this type of testing and defects.

III. DEFINITIONS

Cunha et al. [8] define GUI Testing as "the process of testing a software application with a graphical front-end to guarantee that it meets its specification". GUI system specifications specify both the GUI visual layout and how the system functions through this GUI. Thus, testing of GUI based systems consists of the validation of both the GUI visual layout and its functionality. Our focus in this paper is on the verification of the visual properties of a system.

Test cases for VT should potentially verify different visual properties such as foreground color, background color, alignment, and indentation. A mismatch between a visual system specification (e.g. prototype) and the actual GUI of the system is considered a VD.

Many testing practitioners use a checklist, such as that shown in Fig. 1, to manually inspect and verify the visual properties of a system. In many cases it is easier and faster to prepare a checklist than to create executable test cases that verify these properties. However, like any manual testing, testing using checklists of visual properties is time consuming and error prone. There can also be a certain amount of subjectivity around what constitutes a failure. The GUI of a software system consists of a hierarchy of GUI objects [8,11]. A GUI object has a set of properties. The values of these properties constitute the state of the object at any single point at runtime. An analysis of these properties has led us to identify two groups of them:

1. Presentation properties: properties belonging to this group (color) have direct noticeable impacts on the visual presentation of the corresponding GUI object.
2. Functional properties: properties belonging to this group are concerned with managing the state and identity of the GUI object (e.g. object ID).

Table I shows a list of the identified general common presentation and functional properties of GUI objects in addition to a list of specialized properties for some of the identified GUI objects. Verifying the visual checklist (e.g. Fig. 1), it was concluded that GUI object properties represent the basis on which visual test cases can be written to check the visual state of a system's GUI, at runtime. Consequently, we categorized visual test cases into two main categories with a number of subcategories as summarized in table II.

Given the above GUI object classification, properties, and visual test categories, we propose three definitions:

Definition 1: Visual testing is a testing activity that aims to verify the presentation properties of a GUI object under various conditions, likely based on a high-fidelity prototype. The particular properties of interest may depend to some degree on the particular application and particular GUI toolkit in use. However, Table I lists properties that are likely to be generally applicable.

Definition 2: A Visual Test Case (VTC) is a sequence of steps and expected outcomes in terms of the visual properties of the GUI objects. As shown in Table II, some verifications that are performed, might consider an individual GUI object (called single-object verification), while others require comparing values between multiple GUI objects (called multi-object verification). For example, checking that a given object has the correct font size can be conducted in isolation, while verifying that two objects are appropriately aligned will require comparing the position properties of both of them.

Definition 3: A Visual Defect (or failure) is an external visual anomaly that represents a mismatch between the actual and expected value of a visual property of one or more GUI objects. Regardless of the root cause of a defect, if it results in an anomalous visual property, we classify it as a visual defect.

In the next section we describe an exploratory study based on the above definitions. The goal of the study is to evaluate the applicability of the above definitions and the importance of VT generally.

1. Is the general screen background the correct color?
2. Are the field prompts the correct color?
3. Are the field backgrounds the correct color?
4. In read-only mode, are the field prompts the correct color?
5. Is the text in all fields specified in the correct screen font?
6. Are all the edit boxes aligned perfectly on the screen?

Figure 1: Visual Testing Checklist Example [2].

IV. SETUP OF THE CASE STUDY

TABLE I: General and Specialized Properties of GUI Objects.

GUI Object	Presentation Properties	Functional Properties
General Common Properties Among GUI Objects	direction, style, background color, foreground color, font type, font size, font color, width, height, resizable, disabled, tab index, xposition, yposition, visible, tooltip	class, id, name, type, language, title
Text field	max length, read only, size, default value, selected text color, selection color	Value, shortcut key
Button	caption, icon, disabledicon, captionalignment	Shortcutkey

TABLE II: Visual Test Case Categories.

Category	Subcategory	
single-object verification	1) Position.	2) Size
	3) Color	4) Font
	5) Content (e.g. default value, options, tooltip, caption, format)	
	6) Activation (e.g. disabled, visible)	
multi-objects verification	1) Alignment	2) Spacing
	3) Relative Sizing	4) Tab Order
	5) Consistency (e.g. color, font, size, relative sizing and spacing)	
	6) Ordering	

A. *GOALS, RESEARCH QUESTIONS, AND METRICS*

A Goal-Question-Metric (GQM) [1] approach is adopted in this research. Two goals are identified:

Goal1: Evaluate the precision of the proposed visual testing definitions.

Goal2: Identify the frequency and nature of visual defects in software GUIs.

Three Research Questions (RQs) are identified. One of them is associated with the first goal (Goal1-RQ1), whereas the other two are concerned with the second goal (Goal2-RQ2, and Goal2-RQ3):

Goal1-RQ1: Can the proposed definitions be used to drive visual testing design and execution?

Goal2-RQ2: How appropriate are the proposed VTC categories in describing the VDs being reported by practitioners?

Goal2-RQ3: How do software developers deal with visual vs. non-visual defects from the timing and urgency perspectives?

Inter Rater Reliability (IRR) is the metric calculated to address Goal1-RQ1. The total number of investigated defect reports, the number of VD reports, the number of non-VD reports, and the total number of reported VDs are the metrics used to address Goal2-RQ2. The metrics calculated to address Goal2-RQ3 are: defect age, number of comments, and number of participants.

B. *SUBJECTS AND OBJECTS*

To address the research questions, this study analyzes defects repositories of four GUI based systems in the light of the proposed definition of visual testing context. These are: WordPress[1], Moodle[2], Joomla[3], and Bugzilla[4]. These systems were selected for two reasons: 1) they have large development and user communities, and 2) their defect repositories are publically available. A snapshot of defect repositories of these systems was taken as of November 10th 2011. Then, three hundred (75 from each system) defect reports were randomly selected as the objects of this study whereas the authors were identified as its subjects.

C. *IRR Study: Addressing Goal1-RQ1*

The correctness of the defect classification into VD and non-VD is considered as a metric towards the precision of the proposed definitions; and thus, the achievement of the first goal of this study. An Inter Rater Reliability (IRR) [9] study is designed for this purpose. As the objects of the IRR study, a subset of 25 defect reports was randomly generated for each project. The selected defect reports were then manually analyzed by one of the authors to classify them into VDs and non-VDs based on the proposed definitions. If a VD is identified, its category(ies) and subcategory(ies) were determined. After that, the other two authors performed an independent classification using the same procedure. Each reliability rater handled two projects which were randomly assigned to him. Then, defect classifications were cross validated to inform the reliability of the underlying definitions. Three IRR iterations were conducted to get high reliability defects classification. After each IRR iteration, the proposed visual context definitions were refined and defects reclassified accordingly until the definitions presented in section III were reached with a final classification reliability rate of 95% which addresses Goal1-RQ1. For example, defects numbered 7482 and 3504 for WordPress classified as VD and non-VD, respectively. As a VD, defect number 7482 is further categorized as a single-object verification that belongs to two subcategories: Disabled and Position.

D. *DATA COLLECTION*

After validating the precision of the proposed definitions, we classified the remaining 50 defects in each project to address the second goal of the study. Different data attributes were collected about each analyzed defect report. Due to inconsistent defect reports of the subject systems, only common data attributes among the different systems were collected. Examples of collected raw data attributes include: bug ID, creation and closing dates, release, resolution, severity, priority, number of comments, and number of involved participants. Our analysis also considered one calculated attribute: defect age [14].

V. FINDINGS AND ANALYSIS

A. *Goal2-RQ2*

One of the problems encountered during the data collection process is reports reporting on multiple defects (e.g. Joomla's defect number 22477). This resulted in considering two metrics in counting VDs and non-VDs. The first macro metric counts physical defect reports regardless the number of defects reported in each report. The second micro metric counts the number of reported defects in each report. The macro metric is used to generally quantify VDs in software systems whereas the micro metric is used to study the distribution of VD categories and subcategories. Tables III and IV present the macro and micro VDs distributions, respectively.

As shown in tables III and IV, VDs represent between 16% and 33% of reported defects. 71% of reported VDs are single-object verifications, whereas 29% are multi-object verifications. Standard deviations of these findings are: 7%, 4%, and 4%, respectively. These low standard deviations indicate that the set of values is close to their average, giving us more confidence in the reported average percentages.

B. *Goal2-RQ3*

The age of the defect is a derived attribute that we calculated to study the developer's reaction to reported VDs. We compared the average age of VDs vs. non-VDs in the four systems being studied. We found that the average age

[1] http://core.trac.wordpress.org/report
[2] http://tracker.moodle.org/secure/IssueNavigator.jspa
[3] http://joomlacode.org/gf/project/joomla/tracker/
[4] https://bugzilla.mozilla.org/ query.cgi

TABLE III: Statistics of Visual Defects.

Project	Total Number of Investigated Defect Reports	Number of Visual Defect Reports	Number of Non-Visual Defect Reports	Percentage of Visual Defect Reports
Joomla	75	25	50	33%
Moodle	75	19	56	25%
Bugzilla	75	12	63	16%
WordPress	75	18	57	24%
			Average	25%

TABLE IV: Breakdown of single-object versus multi-objects verifications in Visual Defects.

Project	Total Number of Reported VDs	Single-object verification	Multi-objects verification
Joomla	44	34 (77%)	10 (23%)
Moodle	29	20 (69%)	9 (31%)
Bugzilla	26	18 (69%)	8 (31%)
WordPress	33	23 (70%)	10 (30%)
Average Percentage		71%	29%

of VDs is, on average, longer than that of non-VDs in three of the investigated four systems with WordPress being the exception. In other words, it is tends to take longer to close or resolve a VD than that of non-VD. In addition, we found that the average number of comments and participants of VDs are larger than those of non-VDs as summarized in table V.

To test if the above observations represent statistically significant differences between VDs and non-VDs, we used a t-test. We defined a null hypothesis (H_0: Average Number of Comments in VDs = Average Number of Comments in Non-VDs) and a corresponding alternative hypothesis (H_A: Average Number of Comments in VDs > Average Number of Comments in Non-VDs). The p-value of this test for the four systems being investigated is reported in table V. They are low enough to reject the null hypothesis in favor of the alternative one in three projects (again with the exception of the WordPress project). As a parametric statistical test, t-test assumes that the data being tested is normally distributed. Note that a Chi square goodness of fit test showed that there is no difference between the distribution of the collected data and the normal one, suggesting that the t-test is an appropriate test.

We investigated the WordPress project anomaly and found that this project is unique in having a "WordPress Theme Review Team" responsible for generating GUI theme design, development, presentation, and testing guidelines (http://codex.wordpress.org/Theme_Review). This resulted in having a large number of well-structured, well-presented, and easily imported WordPress GUI themes. In addition, support for several mobile environments is also available for this project. On the contrary, users are enjoying the functionality of Bugzilla but not the GUI as stated in the description of defect report number 259723: "Bugzilla is an awesome product, but the UI leaves a lot to be desired". However, GUI alternatives cannot be introduced due to the lack of GUI oriented community as stated in comment 35 of the same defect report. This indicates that GUI standardization and validation participates in less reported VDs and respective communication.

We conducted a simple qualitative analysis to help identify the factors leading to having a larger number of comments and participant in reports about VDs [7]. Specifically, we used open coding [7] to analyze the defect description and comments (i.e. narrative data) of VDs and non-VDs reports as an initial exploration of why some defects require more discussion than others. Our key finding is that VDs tend to be more subjective or based on personal preference as compared with non-VDs. Thus, the extra discussion of VDs is dedicated to expressing personal views and perspectives about the reported issues. An example of discussion around the nature of a VD arose in Joomla's defect number 26525 where the reporter said: "YOU might like ascending topics, "I" like descending. Is your choice more right than mine? Is mine more right than yours? Certainly not!". One source of subjectivity in this context is the absence of a system UI prototype that can be used as a baseline to judge VDs. It seems likely that building and agreeing on UI prototypes prior to development will save some of visual testing and discussion time.

Furthermore, developers tend to push back on some VDs in an attempt to avoid fixing them and they tend to be reported as a low priority defects. 88% of VDs were reported as a low priority defects whereas 80% of Non-VDs were reported as low priority. However, we found that the average fixing percentage of VDs (71%) is slightly more than that of non-VDs (66%). This shows that although VDs are being reported as a low priority defects, there is an implicit consensus on the importance of their being resolved as they negatively participate to user satisfaction being the first thing a user sees in a system.

C. DISCUSSION

Defect taxonomies are of special importance in software testing as they are used to drive different activities such as: (1) error based testing, and (2) testing time allocation. Error based testing uses a defect taxonomy to drive testing activity that aims to validate the clearance of the System Under Test (SUT) from defects similar to those identified in the baseline defect taxonomy. Also, testers tend to use a baseline defect taxonomy to allocate testing time to testing types based on their history of uncovering defects. The more frequent a given defect is, the more testing time is allocated to testing activities aimed at that type of defect. In both cases visual testing is sacrificed as currently available defect taxonomies [3,4] pay little or no attention to visual defects. Recently, Brook's et al. [5] proposed a modified version of Beizer's [3] taxonomy in their study that aimed at analyzing the defects of GUI based systems. They kept the original eight categories but added a number of subcategories to accommodate GUI defects and others. A "GUI Defect"

TABLE V: Average Age and Participants of Visual vs. Non-visual Defects.

	Joomla		Moodle		Bugzilla		WordPress	
	VD	Non-VD	VD	Non-VD	VD	Non-VD	VD	Non-VD
Average Age in Days	26.000	6.169	165.000	65.351	719.919	500.721	106.000	144.486
Average Number of Participants	6.000	2.755	3.000	2.054	5.300	3.558	3.077	3.666
Average Number of Comments	10.000	3.918	8.000	4.357	16.775	7.023	5.100	8.729
P-value for Average Number of Comments t-test		0.021		0.037		0.022		0.466

subcategory was added to accommodate defects that exist either in the graphical elements of the GUI or in the interaction between the GUI and the underlying API. However, these categories do not show how GUI defects could be identified, what their subcategories are, and how the GUI and visual testing community can benefit from them, Visual testing definitions from this research are suggested as a basis to drive new studies to better position GUI and VDs in defect taxonomies and their dependent software testing activities.

VI. THREATS TO VALIDITY

Threats to construct validity: The definition and categorization of VT are based on analyzing a limited number of open-source software systems that could not be considered as representative for all models of GUI.

Threats to conclusion validity: On the quantification of VDs, reported percentages are limited to the set of defect reports analyzed in each system. However, the randomization of defects selection should support the generalization of these findings in similar contexts.

Threats to external validity: Quantification and distribution of VDs may need to be re-assessed in other contexts such as commercial web-based systems that are based on UI prototypes.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, a new definition for the visual testing of GUI based software systems is introduced and validated with an inter rater reliability of 95%. In our approach to visual testing we consider that a GUI based software system consist of a collection of GUI objects that have two types of properties: functional and presentation. Two types of visual verifications are identified based on presentation properties: single-object and multi-objects. This equips researchers and practitioners with a systematic framework for the preparation, execution, and reporting of VT. The analysis of four issues repositories showed that visual defects represent between 16% and 33% of reported defects. Thus, practitioners are advised to use the proposed definitions and the reported findings to drive and improve their visual testing effort.

It was noticed that VDs involve more people, and thus, interactions than non-VDs. In particular, it seems that when a reported VD does not have a baseline in some form such as a prototype, more people and more discussion is involved in resolving the defect. However, the fixing percentage of VDs is slightly more than that of non-VDs showing the implicit common agreement between practitioners on the importance of delivering a GUI that is free from noticeable discrepancies. A revised defects classification is required to explicitly emphasize the position of VDs in reported defects. Finally, the efficiency of the proposed VDs categories in driving the VT effort needs to be assessed and reported.

ACKNOWLEDGMENT

This work was supported by the NSERC Discovery Grant program (no. 341511-07 and 341367-07) and the SurfNet NSERC Strategic Grant.

REFERENCES

[1] Basili, V., (1992). *Software Modeling and Measurement: the Goal/Question/Metric Paradigm*. Technical Report. University of Maryland, Institute for Advanced Computer Studies.

[2] Bazman, (1999). GUI Testing Checklist [online]. Web page: Tripod. Available from: http://bazman.tripod.com/checklist.html?button1 =GUI+Testing+Checklist [Accessed 09/30/2011].

[3] Beizer, B., (1990). *Software Testing Techniques*. Van Nostrand Reinhold Co.

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

[5] Brooks, P., Robinson, B., and Memon, A., (2009). An Initial Characterization of Industrial Graphical User Interface Systems *in Proceedings of International Conference on Software Testing Verification and Validation* Denver, CO, IEEE, pp.11-20.

[6] Choudhary, S., Prasad, M., and Orso, A., (2012). CROSSCHECK: Combining Crawling and Differencing To Better Detect Cross-Browser Incompatibilities in Web Applications *in Proceedings of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* Montreal, Canada IEEE, pp.171-180.

[7] Corbin, J. and Strauss, A., (2007). *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. 3rd edition. Sage Publications, Inc.

[8] Cunha, M., Paiva, A., Ferreira, H., and Abreu, R., (2010). PETTool: A Pattern-Based GUI Testing Tool *in Proceedings of 2nd International Conference on Software Technology and Engineering (ICSTE 2010)* San Juan, PR IEEE, p.V1202-V1206.

[9] Gwet, K., (2012). *Handbook of Inter-Rater Reliability* . 3rd Edition. Gaithersburg, MD, USA : Advanced Analytics, LLC.

[10] Mahajan, R. and Shneiderman, B., (1997). Visual and Textual Consistency Checking Tools for Graphical User Interfaces *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 23(11), pp. 722-735.

[11] Memon, A., Banerjee, I, and Nagarajan, A., (2003). GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing *in Proceeding of Tenth Working Conference on Reverse Engineering; Victoria, BC; 13 November 2003 through 16 November 2003* IEEE Computer Society , pp.260-269.

[12] Mesbah, A. and Prasad, M., (2011). Automated Cross-Browser Compatibility Testing *in Proceedings of 33rd International Conference on Software Engineering* Honolulu, HI, USA ACM, pp.561-570.

[13] Tamm, M., (2009). Fighting Layout Bugs [online]. Zurich: Google Test Automation Conference . Available from: http://code.google. com/p/fighting-layout-bugs/ [Accessed 09/21/2011].

[14] Tian, J., (2005). *Software Quality Engineering: Testing, Quality* Assurance, *and Quantifiable Improvement*. Wiley.