GUI Failure Analysis and Classification for the Development of In-Vehicle Infotainment

Daniel Mauser
Daimler AG
Ulm, Germany
daniel.mauser@daimler.com

Alexander Klaus
Fraunhofer IESE
Kaiserslautern, Germany
alexander.klaus@iese.fraunhofer.de

Ran Zhang
Robert Bosch GmbH
Leonberg, Germany
ran.zhang@de.bosch.com

Linshu Duan
AUDI AG
Ingolstadt, Germany
linshu.duan@audi.de

***Abstract*—Modern automotive infotainment systems have sophisticated graphical user interfaces, leading to various challenges in software testing. Due to the enormous amount of possible interactions, test engineers have to decide which test aspects to focus on. In this paper, we examine what types of failures can be found in graphical user interfaces for automotive infotainment systems and how frequently they occur. A hierarchical classification for failures has been developed based on common concepts in software engineering, such as Model-View-Controller and Screens. More than 3,000 failures, found and fixed during the development of automotive infotainment systems at Audi, Bosch and Mercedes-Benz, have been analyzed. Results show that 62% of reports describe failures related to high and low level behavior, 25% of reports describe failures related to contents and 6% of reports describe failures related to design.**

***Keywords*-failure reports; domain specific failures; GUI based software; in-vehicle infotainment system.**

I. INTRODUCTION

In modern automotive infotainment systems the graphical user interface (GUI) is an essential part of the software. The so-called HMIs (human machine interface) provide the system functionality to the user, whether it be the radio system, the navigation, or system functionality, such as the tire pressure monitoring system. According to Robinson and Brooks [1], a GUI "is essential to customers, who must use it whenever they need to interact with the system". Additionally, they "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" [1]. As automotive infotainment GUIs are built into a car, there is no easy possibility to upgrade the system or to buy a new release, which renders the situation for such systems even worse. Additionally, when the system does not work correctly, drivers may get distracted from driving. Therefore, special attention has to be drawn on finding and fixing defects during development.

Figure 1 shows an example of a screen in an HMI. It consists of a menu at the top of the screen, where all available applications, e.g., navigation or audio, can be accessed. Each application consists of an application area at the middle of the screen, where the actual content is displayed (here: information about the radio station and the song played) and a sub menu for content specific options at the bottom (here: "Radio", "Presets", "Info", etc.). The HMI is operated via a central control element (CCE) allowing the user to set the selection focus by rotating or pushing the CCE in a direction, and to activate options by pressing it down. This interaction concept is common in modern in-vehicle infotainment systems.

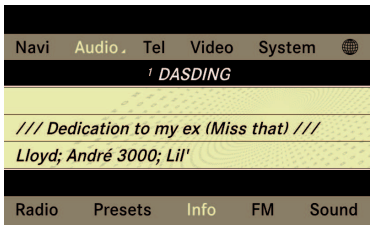

Figure 1. Example for a graphical user interface of the Mercedes-Benz infotainment system COMAND

GUI based software, especially in the automotive domain, is becoming more and more complicated - often, documents with more than 2,000 pages are written to describe all the functionality [2]. The reasons are the growing amount of functions, which form more and more complex systems, as well as the usage of more advanced graphical views and elements (e.g., complicated animations or 3D-elements).

When testing GUIs, sequences of system interactions are performed and the system reaction is compared to the specified reaction in each step. It is obvious that for such complicated systems, not all possible combinations can be tested, and thus it is necessary to focus testing activities on certain failure types. To be able to choose strategies accordingly, several questions need to be answered:

- What types of failures can be found in GUI based software today? Is it possible to build a classification of these types?
- What are frequent failures in current GUI software? Which are common, which are rare?

The article is structured as follows. In Section 2, we discuss related work and show why we need to create a new classification scheme. Section 3 describes our approach

 ISBN: 978-1-61208-233-2

for creating a scheme, which is then presented in Section 4. Section 5 discusses the results of our work. The article ends with a discussion of the approach and future work in Section 6.

II. Related Work

In the literature, various types of defect classifications can be found. However, many of them lack the practical usage and empirical data in the form of distributions of defects into the scheme, and thus it is hard to tell whether they are a valuable addition. Other schemes for classification are used frequently, or at least once. For our study, we concentrate on those latter ones, and discuss why they are not fully suited for our means. As described above, our context is black-box testing of a GUI for automotive infotainment systems.

IBM created the so-called Orthogonal Defect Classification (ODC) [3]. Since then, many companies have applied this approach. It consists of several attributes, such as triggers, defect types, impact, and others. A GUI-section is included in the ODC Extensions V5.11. It contains triggers, such as "Design Conformance", "Navigation", and "Widget/GUI Behavior". Compared to our classification (see Section 4), some of the triggers match to our categories, but not fully: "Navigation" is represented by "Screen Transition" and "Widget Behavior", which also maps to the "Widget Behavior" trigger. Our "Design" category maps to "Icon Appearance" and "Design Conformance", while "Input Devices" are not covered by us. All in all, the scheme would be spread across three levels of our classification.

Another scheme, which contains several categories for GUI-related issues, was made by Li et al. [4]. It consists of 300 categories, and is based on the ODC, but adapted for black-box testing. It contains, e.g., categories for GUIs in general, and for GUI control [4]. The GUI-related categories do not fully fit, for example, there is a "Title bar" category, but our systems do not have title bars, as desktop software does. This scheme is created for regular desktop software, as it also classifies keyboard or mouse related faults. Due to the differences between desktop software and our systems, we decided not to adapt this scheme.

Børretzen and Dyre-Hansen [5] created a scheme, which is also based on the ODC. They target industrial projects. A GUI fault category is included, but not further segmented. The rationale for this is that, although "function and GUI faults are the most common fault types", they are most often not severe, and thus, not as critical as other categories [5]. This seems to be a contradiction to what was stated in the introduction, but the criticalities of certain types of faults are subject to the application domain. As stated in the introduction, in our application domain they are very critical, and therefore, we focus on them to assure software quality.

Hewlett-Packard created a scheme based on three categories: origin, type, and mode [6]. Origin refers to where the defect was introduced, the type can, e.g., be logic, computation, or user interface. The mode refers to whether something was missing, unclear, wrong, changed, or done in a better way now. This scheme also does not differentiate the various types of GUI-related failures.

Another well-known scheme has been developed by Beizer [7]. The main categories are "requirements, features and functionality, structure, data, implementation and coding, integration, system and software architecture, and testing" [7, p. 33], each having three levels of subcategories. The scheme is very detailed, but there is no GUI-related category.

An adaption of this scheme for GUI contexts has been created by Brooks, Robinson and Memon [8]. The authors emphasize that "defining a GUI-fault classification scheme remains an open area for research" [8]. They simplified Beizer's scheme to create a two level classification and added a subcategory for GUI-related issues, "to categorize defects that exist either in the graphical elements of the GUI or in the interaction between the GUI and the underlying application" [8]. However, all of our failures would fit into that category, and thus, we cannot use this scheme.

There also exists a fault classification scheme for automotive infotainment systems [9], however, this scheme is based on the network communication, and thus, it cannot be used for our purposes of classifying software based GUI failures. This scheme differs between hardware and software, but does not differentiate the different possibilities of issues in the software enough: "software based system faults can be computational, data management and interface faults" [9]. This scheme again has many categories not usable by us, and does not include different GUI-related categories.

Ploski et al. [10] studied several schemes for classification, including approaches not presented here. Since there was no matching scheme, we did not present them here.

Another approach has been created by the IEEE [11]. However, this approach is not very detailed, and just lists a number of attributes to be filled out for each defect. But the standard includes a scheme for distinguishing between defects and failures. A defect is "an imperfection or deficiency in a work product that does not meet its requirements or specifications", while a failure is "an event, in which a system or system component does not perform a required function within specified limits" [11]. So, when a defect is present, and we perform GUI testing, we can observe failures. They are caused by defects in the code, but since we test by using the GUI, and not the code (i.e., black box), what we can observe is the behavior, and this is why we do not create a defect but a failure classification scheme.

The classification schemes available do not meet our requirements. Since we employ block-box testing of GUIs, we cannot use any code-related categories or schemes. We focus only on GUI-related failures. The schemes presented in [6][7] and [9] do not have GUI-related categories and because of this, they cannot be used by us. Others ([3][5][8])

 ISBN: 978-1-61208-233-2

have GUI-related categories, but still do not meet very well to our purposes. The scheme presented in [4] has many GUI-related categories, but for desktop software. Due to the differences of desktop and automotive infotainment GUIs, we did not adapt it because we would then have to either delete or change most of the categories. Therefore, we created our own failure classification scheme. After describing the approach we used, the categories of our scheme are explained in Section 4.

III. METHODOLOGY

For this research, we analyzed databases of existing failure reports. The data was collected during the development of state of the art automotive infotainment systems. Testers executed the System Under Test (SUT) manually, based on specification documents, and used failure reporting tools to keep records of anomalies. The reports were handed over to developers who then rechecked and fixed the software. In this context, failures are defined as mismatch between the SUT and an explicit GUI specification, which can be observed while operating the system. Any implicit requirements, such as general standards or guidelines, are not subject of the study. Only reports that were accepted as failures by both testers and developers were accounted. Failures that are not referring to the GUI were sorted out.

For this study, Audi, Bosch and Mercedes-Benz provided failure data. Hence the analyzed reports represent a broad variety of contexts as they stand for different infotainment systems (Audi MMI, Mercedes-Benz COMAND and several projects, developed at Bosch), different steps in the development process as well as different test strategies, test personnel and test environments. In total, more than 3,000 reports were analyzed. One third of the reports have been used as training data to construct the failure classification which then was fine-tuned using the remaining reports as test data.

As preparation of the analysis, the reports were exported to an Excel document with one line for each report. Furthermore, reports that describe more than one failure have been split up in one line for each failure. Redundant reports that describe exactly the same failure as already considered ones were removed. The following information per report was relevant for the analysis:

A **Report ID** identifies the reports uniquely. In the **Title** testers describe the essence of the report. The **Problem description** is a detailed statement on (a) the required setup of the system under test, (b) the actions that lead to the failure, (c) the behavior or result that has been observed, (d) a description, what should have been displayed instead and (e) how this failure could be bypassed. If failures were ambiguous or hard to describe, screen shots were added. Table I shows simple examples of reports.

We analyzed this data iteratively by hand to develop a classification by clustering similar failures. To determine the similarity of failures, the classification is based on concepts and patterns used in software engineering. For example, the top level failure classes are *behavior*, *contents*, and *design*, according to the well-established Model-View-Controller [12] design pattern. The structure of the classification and related separation criteria are presented in Section 4.

Table I
EXAMPLES OF THE ANALYZED GIVEN FAILURE REPORTS

ID	Title	Problem description
4711	Inserted music CDs are not played automatically	*Setup*: Any state *Actions*: Insert music CD *Observed result*: Nothing happens *Expected result*: System should display CD play screen *Reference*: R0026679 *Workaround*: Navigate to CD play screen manually
4712	Cell phone icon on call screen obsolete	*Setup*: Connect cell phone *Actions*: Navigate to Call screen *Observed result*: Placeholder icon for cell phones is displayed *Expected result*: Correct icon is displayed *Reference*: R0026672 *Workaround*: —

A classification is needed that gives a good overview and is flexible to extend for comprehensiveness. This should be achieved by a hierarchical structure. As indication, how many hierarchy levels have to be applied and whether one category could be subdivided reasonably or several categories should be combined, we defined the following requirements for the failure classes: To scale the scope of each classification level, an initial analysis of the data indicates the necessity to limit the percentage of the lowest level to 10% of the total numbers of failures. To develop a clear and easy to use structure, the number of categories on every level has to be 2 in minimum and 5 in maximum.

IV. FAILURE CLASSIFICATION

In this section, the GUI failure report classification is described. Table II gives an overview of the entire classification, including the failure distribution. As mentioned above, the top level follows the Model-View-Controller pattern [12], as this pattern proved to be an adequate abstraction for GUI based software. **Controllers** (here: *behavior*) abstract the observable behavior, indicating how input is processed. **Models** (here: *contents*) define all contents that are displayed by the system. **Views** (here: *design*) describe layout and appearance of the contents to be displayed. As the SUT was tested as a black box, the MVC pattern is not intended to represent the actual software structure or to relate any failures to implemented software modules.

In order to avoid enforced classifications of reports to existing classes, one category “to be categorized” (TBC) has been created. As for other categories, on the lowest level the TBC failure class is limited to 10% of the total number of

 ISBN: 978-1-61208-233-2

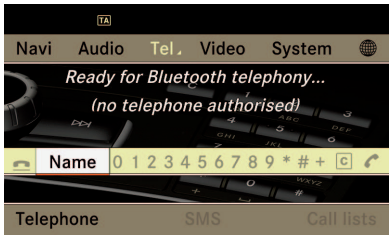

Figure 2. Screen example: Telephone application

failures. Classifying more failures than that limit as TBC would indicate, that the definition of an additional failure class is necessary.

A. Behavior

The top level failure class *behavior* contains all failure reports describing that stimuli to the SUT do not result in the specified output. In order to subdivide this failure class, common abstractions in GUI development were applied:

Screens [13][14] represent the current state of the GUI displayed. This state is defined by the options available to the user. Figure 1 shows the radio screen, where the current radio station and the song playing are displayed. The options provided allow users to change waveband (FM option) or adjust the sound setting (Sound option). Screens are structured based on elementary GUI elements, so called **widgets**. Widgets are either primitive (label, rectangle, etc.) or complex, meaning that they are a composition of primitive or again complex widgets. An example for widgets in Figure 1 would be the horizontal list in the top end that contains button widgets for all available applications, such as “Navi”, “Audio” or “Tel” (i.e., phone). In this classification, the concepts of screens and widgets are used to differentiate micro behavior that affects single elements on the display (e.g., iterating list entries) and macro behavior that changes the entire context of use.

1) Widget Failures: The GUIs of the automotive infotainment systems analyzed mainly use various types of lists to present options to the user. To activate an option, those lists set a focus by turning or pushing the CCE and pressing the CCE once the option wanted is focused. Potential failures might be that the wrong option is focused on start or that the focus changes not as specified. An example would be that every time the main menu is entered, the element in the middle should be focused automatically. A failure would exist, if the first element would be focused instead. Those failures are considered as deficient *widgets focus* logic. Subcategories are *initial focus* (the wrong option is focused when a list is entered), *implicit focus* (the focus has to be reset due to changing system conditions) or *explicit focus* (the user resets the focus by turning or pushing the CCE).

For widgets, often additional behavior is specified. One example might be alphabetic scrolling to allow the user to jump to a subgroup of list entries starting with one specific letter. Reports describing that such behavior is either *missing* (specified behavior is not implemented), *wrong* (instead of specified behavior, behavior not specified is implemented) or *extra* (behavior not specified is implemented), are considered as deficient *widget behavior*.

2) Screen Structure Failures: In this failure class, reports are clustered describing the logic to determine the widget objects the screens contain and what data they hold. In automotive infotainment systems, the availability of options depends

Table II
THE DISTRIBUTION OF FAILURES

1. level	2. level	3. level	4. level	distr.
TBC	-	-	-	*7.6 %*
Behavior (Σ: 61.5%)	Screen Transition (Σ: 17.9%)	missing	-	*5.8 %*
		extra	-	*2.9 %*
		wrong	-	*9.2 %*
	Pop-up Behavior (Σ: 11.7%)	missing	-	*3.6 %*
		extra	-	*3.2 %*
		priority	-	*0.5 %*
		wrong	-	*4.4 %*
	Screen Structure (Σ: 13.8%)	screen composition (Σ: 5.4%)	missing	*2.4 %*
			extra	*0.9 %*
			wrong	*2.1 %*
		options offer (Σ: 5.4%)	missing	*2.2 %*
			extra	*1.3 %*
			wrong	*1.0 %*
			order	*0.9 %*
		option gray-out (Σ: 3.0%)	missing	*1.6 %*
			extra	*1.0 %*
			wrong	*0.4 %*
	Widget (Σ: 18.1%)	Behavior (Σ: 14.7%)	missing	*5.1 %*
			extra	*0.9 %*
			wrong	*8.7 %*
		focus (Σ: 3.4%)	initial	*0.9 %*
			implicit	*1.5 %*
			explicit	*1.0 %*
Contents (Σ: 25.1%)	Text (Σ: 15.1%)	design time (Σ: 5.9%)	missing	*1.2 %*
			incomplete	*0.3 %*
			extra	*0.5 %*
			wrong	*3.9 %*
		run time (Σ: 9.2%)	missing	*2.2 %*
			incomplete	*1.1 %*
			extra	*1.0 %*
			wrong	*4.9 %*
	Animation (Σ: 1.8%)	design time (Σ: 0.8%)	missing	*0.4 %*
			extra	*0.1 %*
			wrong	*0.2 %*
			others	*0.1 %*
		run time (Σ: 1.0%)	missing	*0.4 %*
			extra	*0.1 %*
			wrong	*0.3 %*
			others	*0.1 %*
	Symbols & Icons (Σ: 8.2%)	design time (Σ: 2.9%)	missing	*1.5 %*
			extra	*0.2 %*
			wrong	*1.2 %*
		run time (Σ: 5.3%)	missing	*2.2 %*
			extra	*1.0 %*
			wrong	*2.1 %*
Design (Σ: 5.8%)	color	-	-	*1.0 %*
	font	-	-	*0.4 %*
	dimension	-	-	*0.7 %*
	shape	-	-	*0.4 %*
	position	-	-	*2.7 %*
	other	-	-	*0.6 %*

 ISBN: 978-1-61208-233-2

on numerous conditions, such as available devices (e.g., radio tuner available, connected mobile phones, etc.), the current environmental conditions (e.g., car is moving faster than 6 km/h) or even previous interactions (e.g., activating route guidance). These conditions affect, whether options are displayed but cannot be selected (gray-out mechanism) or whether options are listed at all. Failure reports describing that the options are displayed incorrectly are considered as deficient *option offer* or *option gray-out*. The subclass *screen composition* clusters failures related to deficient setup of widgets on screen. Subclasses of this category are *wrong widget* (the wrong widget is displayed), *extra widget* (an unspecified widget is displayed) or *missing widget* (widgets that are specified are absent). *Screen structure* failures are distinguished from the *widget behavior* category as follows: the former represents erroneous selection of widgets such as horizontal or vertical lists, whereas the latter clusters failures of widget behavior itself, such as the scrolling logic or widget state change.

3) Screen Transition Failures: As described above, screens represent one special usage context. The failure class *screen transition* clusters failures occurring when those usage contexts change. One indication for a screen transition is that the widget composition and the displayed options are replaced. With Figure 1 and Figure 2, a screen transition is demonstrated: first, the Radio screen is shown; with activating the option “Tel”, the context changes to the telephone screen of the infotainment system. Subclasses of this category are *missing transitions* (a specified transition does not take place), *extra transitions* (a transition that is not specified takes place) or *wrong transitions* (instead of screen A, screen B is displayed).

4) Pop-up Behavior Failures: With automotive infotainment systems, messages are often overlaid over the regular screen (Pop-up mechanism). Those messages inform users about relevant events or change of conditions. For example, those messages might state that the car has reached the destination of an active route guidance or that hardware has heated up critically. Subcategories are *missing* (the pop-up is not displayed although the respective conditions are active), *extra* (pop-up appears although the respective conditions are not active) and *wrong* (instead of pop-up A, pop-up B is displayed). Additionally, with the pop-up mechanism the priority system is important: a pop-up with higher priority always has to be displayed on top of pop-ups with lower priority. Those failures are clustered in the subclass *priority*.

B. Contents Failures

The next top level category is related to contents. The separation criterion is the type of the content: *symbols & icons*, *animations*, or *text*. In Figure 1, a contents failure would be, if the button for the “Audio” application would have been labeled incorrectly with “Adio” or the globe symbol in the upper right corner of the screen would be a placeholder. In this classification, we distinguish content that is known at *design time* (e.g., the labels of available applications) and content that cannot be defined until *run time* (e.g., displaying the names of available Bluetooth devices). For each of those content types, subclasses for *wrong*, *missing* and *extra content* have been defined.

This category might be confused with the screen structure failure class in the behavior sub tree. For example, a failure report describing that the second button in the main menu is “Blind Text” instead of “Audio” could be categorized as *contents* or *option provision* failure. If pressing the button still leads to a screen transition to the Audio context, the report is considered as deficient contents. If another context is displayed, for example the telephone screen, it would be a deficient option provision.

C. Design Failures

The last top level category clusters reports, which describe design failures. This includes *color* (e.g., focus color is red instead of orange), *font* (e.g., text font is Times New Roman instead of Arial), *dimension* (e.g., a button is higher or broader than specified), *shape* (e.g., a button should be displayed with rounded instead of sharp edges) and *position* (e.g., a label of a button is centered instead of left-aligned). As design failures often were described vaguely, a subcategory for *other* design failures was defined. Ambiguous descriptions were, for example, that wrong arrows, wrong Cyrillic letters or a wrong clock were observed. As it became obvious early, that a low percentage of reports were categorized as design failures, no additional work has been done to clarify this category.

V. Discussion

The requirements defined in Section 3 were met for most failure classes. We intended to cover at least 90% of all defect reports analyzed. Only 7.6% of the reported failures had to be classified as “to be categorized”. Furthermore, we intended to limit the percentage of the classes on the lowest level of the hierarchy to 10%. This could be achieved as well: with 9.2%, the largest category was *behavior - screen transition - wrong*. We intended to allow only 2-5 categories on each hierarchy level. This could not be realized for the design category (6 subclasses). However, due to a very small number of failures classified as design related (5.8%), we did not consider it necessary to restructure this category.

Further, we answered the question, what types of failures are frequent in current GUI software. The results show that the majority (61.5%) are failures related to behavior. This points out the complicated macro and micro behavior in modern infotainment systems. Most of the failure reports are related to missing or wrong individual widget behavior (13.8%), as well as missing or wrong screen transitions (15.0%). The content category is the second biggest top level failure class (25.1%), with erroneous text being the biggest

 ISBN: 978-1-61208-233-2

subcategory (15.1%). The majority (9.2%) is not known until run time. Explanations are (a) that in most infotainment systems information is mainly displayed textually and (b) that testing texts is easier for human testers than comparing symbols or animations in detail. Very few failures (5.8%) describe erroneous design. One explanation might be, that design is hard to test manually. For example, it is a problem to differentiate shades of colors by eye. In addition, most design errors are less critical and might even not be recognized by users. Therefore, testing design might not be of high priority to test planners.

VI. Conclusion and Future Work

In this paper, we answered the question what types of failures can be found in GUI based infotainment systems in the automotive domain today. A failure classification has been developed and applied to more than 3,000 failure reports created during the development of modern automotive infotainment systems at AUDI, Bosch and Mercedes-Benz. 62% of the reports describe failures related to high and low level behavior, 25% of the reports describe failures related to contents and 6% of the reports describe failures related to design. We support not only testers, but the entire GUI development process by pointing out pitfalls leading to gaps between the specification and the implementation. The classification indicates, what aspects need special attention in specification documents and might need to be described more explicitly than is usual today. For roles responsible for the implementation of GUI concepts, this work points out aspects that might be ambiguous and need clarification.

In future research, the suggested classification might be scaled by reducing the maximum percentages of lowest level categories. Thus, some categories have to be differentiated further and additional failure classes have to be defined. Moreover, additional parameters such as “failure criticality”, “predicted number of affected users” or “costs for testing” could be added to the classification. Those aspects are not in focus at the current stage and might influence the choice of test strategies significantly. One could then focus or prioritize testing on those types of failures, which are most critical, based on the frequency and these additional parameters. For this, coverage criteria and prioritization techniques are currently examined, to check, which of them, if any, may be used for our purposes. This classification could be applied to future automotive infotainment systems to analyze change of the failure focus.

Acknowledgment

The authors would like to thank Krishna Murthy Murlidhar, Sven Neuendorf and Jasmin Zieger for their contributions. The research described in this paper was conducted within the project automotiveHMI. The project automotiveHMI is funded by the German Federal Ministry of Economics and Technology under grant number 01MS11007.

References

[1] B. Robinson and P. Brooks, “An initial study of customer-reported gui defects,” in *Software Testing, Verification and Validation Workshops, 2009. ICSTW’09. International Conference on*. IEEE, 2009, pp. 267–274.

[2] C. Bock, “Model-driven hmi development: Can meta-case tools do the job?” in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. IEEE, 2007, pp. 287b–287b.

[3] R. Chillarege, “Orthogonal defect classification,” *Handbook of Software Reliability Engineering*, pp. 359–399, 1999.

[4] N. Li, Z. Li, and X. Sun, “Classification of software defect detected by black-box testing: An empirical study,” in *Software Engineering (WCSE), 2010 Second World Congress on*, vol. 2. IEEE, 2010, pp. 234–240.

[5] J. Børretzen and R. Conradi, “Results and experiences from an empirical study of fault reports in industrial projects,” *Product-Focused Software Process Improvement*, pp. 389–394, 2006.

[6] R. Grady, *Practical software metrics for project management and process improvement*. Prentice-Hall, Inc., 1992.

[7] B. Beizer, “Software system testing techniques,” *New York: Van Norstrand Reinhold*, 1990.

[8] P. Brooks, B. Robinson, and A. Memon, “An initial characterization of industrial graphical user interface systems,” in *Software Testing Verification and Validation, 2009. ICST’09. International Conference on*. IEEE, 2009, pp. 11–20.

[9] M. Kabir, “A fault classification model of modern automotive infotainment system,” in *Applied Electronics, 2009. AE 2009*. IEEE, 2009, pp. 145–148.

[10] J. Ploski, M. Rohr, P. Schwenkenberg, and W. Hasselbring, “Research issues in software fault categorization,” *SIGSOFT Software Engineering Notes*, vol. 32, no. 6, pp. 1–8, November 2007.

[11] “Standard classification for software anomalies,” *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pp. C1 –15, 7 2010.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Reading, MA: Addison Wesley, 1995.

[13] S. Stoecklin and C. Allen, “Creating a reusable gui component,” *Softw. Pract. Exper.*, vol. 32, no. 5, pp. 403–416, Apr. 2002.

[14] J. Chen and S. Subramaniam, “Specification-based testing for gui-based applications,” *Software Quality Journal*, vol. 10, pp. 205–224, 2002.

 ISBN: 978-1-61208-233-2