IMPROVING GRAPHICAL USER INTERFACE (GUI) DESIGN USING THE COMPLETE INTERACTION SEQUENCE (CIS) TESTING METHOD

by

BENEDICT JOHN JAKUBEN

Submitted in partial fulfillment of the requirements For the degree of Master of Science

Thesis Advisor: Dr. Guo-Qiang Zhang

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

January, 2011
Committee Signature Sheet

CASE WESTERN RESERVE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Benedict John Jakuben

Candidate for the Master of Science degree *.

(signed)  Dr. G. Q. Zhange (chair of the committee)
          Dr. Andy Podgurski
          Dr. Lee White

(date)    October 27, 2010

*We also certify that written approval has been obtained for any proprietary material contained therein.
Table of Contents

List of Figures .................................................................................................................. 4
List of Tables ..................................................................................................................... 5
Abstract ............................................................................................................................... 6
Chapter 1 Introduction ....................................................................................................... 7
Chapter 2 Background .................................................................................................... 19
  2.1. Previous Work ......................................................................................................... 19
  2.2 CIS Testing Method ................................................................................................ 20
    2.2.1 GUI Objects ...................................................................................................... 21
    2.2.2 Responsibilities ............................................................................................... 22
    2.2.3 Complete Interaction Sequences (CIS) ......................................................... 22
    2.2.4 FSMs ................................................................................................................. 23
    2.2.5 Design Tests ..................................................................................................... 25
    2.2.6 Implementation Tests ...................................................................................... 26
    2.2.7 Failures ............................................................................................................. 28
  2.3 Memory Tools .......................................................................................................... 29
Chapter 3 Method ............................................................................................................ 31
  3.1 Configuration of the Software/Hardware Used .................................................... 31
  3.2 Testing with Source Code Available .................................................................... 31
  3.3 Use of Memory Tools ............................................................................................. 32
  3.4 Description of the Empirical Study ....................................................................... 33
  3.5 Threats to the Empirical Study .............................................................................. 39
## Chapter 4 GUI Design and Implementation

### 4.1 Data

- **4.1.1 Users** .................................................................43
- **4.1.2 Classes and Classrooms** .......................................44
- **4.1.3 Students** .............................................................45
- **4.1.4 Parents/Emergency Contacts** ..................................46
- **4.1.5 Assignments and Grades** .......................................46
- **4.1.6 Lessons** ...............................................................47

### 4.2 System Interaction

- **4.2.1 Program Initialization** ............................................48
- **4.2.2 Log-in** .................................................................49
- **4.2.3 Main Class Screen** ................................................49
- **4.2.4 Class Information** ................................................49
- **4.2.5 Creating a Class List** .............................................50
- **4.2.6 Editing Student Information** ....................................51
- **4.2.7 Editing/Viewing Parent Information** .........................52
- **4.2.8 Grades** ...............................................................53
- **4.2.9 Add/Edit Assignments** ..........................................53
- **4.2.10 Assignment Groups** .............................................53
- **4.2.11 Viewing/Editing Grades** .......................................54
- **4.2.12 Classroom Map** ..................................................55
- **4.2.13 Adding Objects** ..................................................56
- **4.2.14 Deleting Objects** ................................................56
List of Figures

Figure 1-1: Drag-and-Drop GUI Development .................................................................9
Figure 1-2: Generated code for the button in Fig. 1-1 .......................................................10
Figure 1-3: Open File FSM .............................................................................................13
Figure 1-4: Open File FSM with Additional Transition .....................................................15
Figure 2-1: Example FSM ...............................................................................................26
Figure 2-2: Example FSM with Additional Transitions ....................................................27
Figure 3-1: UI Diagram..................................................................................................34
Figure 4-1: Database Entity-Relationship Diagram ..........................................................44
Figure 4-2: Main Startup Dialog ......................................................................................48
Figure 4-3: Main Class Screen ........................................................................................50
Figure 4-4: Class Information Screen ..............................................................................52
Figure 4-5: Classroom Map .............................................................................................55
Figure 4-6: Lesson Planner .............................................................................................58
Figure 5-1: “Edit Student Information and Save” FSM with New Transitions ..............64
List of Tables

Table 1-1: GUI Objects in the Open File CIS .................................................................17
Table 3-1: UI Diagram Path Descriptions .................................................................35
Table 3-2: Rating Scale for Defects ........................................................................38
Table 5-1: Summary of Results .................................................................................63
Table 5-2: Additional Transitions ..............................................................................66
Table 5-3: Summary of Additional Transitions that Led to Failures .................67
Improving Graphical User Interface (GUI) Design Using the Complete Interaction Sequence (CIS) Testing Method

Abstract

by

BENEDICT JOHN JAKUBEN

This thesis involves designing and implementing a complex GUI system using a drag-and-drop GUI tool, followed by modeling the system as a set of finite state models (FSMs) to be used as the basis for developing a suite of design and implementation tests using the concept of complete interaction sequences (CIS). The next step is identify new transitions within the FSMs corresponding to implementation tests that were not also design tests, together with defects and surprises only detected by this subset of implementation tests. These additional transitions and their associated faults (sum of defects and surprises) are then analyzed to determine the root cause of how and when they were introduced into the system and whether or not they are due to the GUI tool used to produce the GUI code. Suggestions are given on how to prevent this unintended behavior introduced by the GUI tool, not the GUI design.
Chapter 1 Introduction

The Graphical User Interface (GUI) testing problem, as demonstrated by White et al. in [1] and [2], is a difficult problem for a number of reasons. The primary hurdles are that GUIs often have a very large number of states, inputs, and events, and that complex dependencies can exist within even the most simple of GUI systems. As GUIs are used for more and more systems, it becomes increasingly important to develop reliable, repeatable, scalable testing methods for testing GUIs. The percentage of code that developers write for GUIs is also increasing [9], further magnifying the need for improved GUI testing methods.

To obtain complete test coverage of all possible inputs and transitions between states of the GUI requires a very large number of tests as test growth is potentially combinatorial [7] based on multiple inputs and multiple paths through the system. This combinatorial growth occurs because each state of the GUI has a distinct number of choices and, when combining GUI objects into a path through the system, the user can select any one of the choices of each subsequent object in the path. Consider, for example, a simple GUI system with three radio buttons that allow the user to select one of 2, 3, and 4 choices, respectively. If the order of these inputs matters (i.e., the user must choose 1 of 2 choices, then 1 of 3, and finally 1 of 4), then the total number of paths required to fully test the GUI would be 24 (2 x 3 x 4). However, in most GUIs the user can select inputs in any order, and this ordering of the paths to test gives rise to factorial growth.
It is very important to reliably test GUIs because they are so important to present
day software [5]. GUIs are increasingly being used in consumer, business, and safety-
critical systems, which makes it even more important that they are reliable and free of
defects [5, 12]. This reliability can be improved through design, development, and testing,
and for this thesis the focus is on how testing can help improve the quality of a system
through the development and testing phases.

A more recent development in the implementation of GUIs is the significant
enhancement of GUI design and development tools, such as Microsoft Visual Studio
.NET. Because GUIs are increasingly being used in all types of systems, the Integrated
Development Environments (IDEs) used to write the code for the GUI have become more
powerful with much more functionality available to the programmer to assist in the
development of the GUI objects and their interactions. Many programming languages and
frameworks now support drag-and-drop design interfaces that provide controls and
templates that the developer can drag onto specific GUI screens. This makes it easy to
add GUI objects like buttons and textboxes or easily establish connections to data sources
to populate fields and displays in the program. The developer simply selects a GUI object
from a menu or toolbar in the IDE, drags it to a place on a GUI page where he or she
wants it to appear, and then drops it onto the page, instantly making it a new part of the
system. A detailed discussion of how drag-and-drop development tools “shield the
programmer from much of the intricate complexity associated with events, listeners, and
handlers” is given by Bishop and Horspool in [6].

An example of this type of functionality is the ability for the developer to drag a
button object onto a GUI page (Figure 1-1). All the code to initialize and display the
button (such as size, color, label, etc.) is automatically added by the IDE to the appropriate place in the code line (Figure 1-2). The developer can then modify the object using the tool or, in most cases, by working directly with the code. Note that both Figures 1-1 and 1-2 show attributes of the button that can be changed by the developer. The IDE generates all the necessary code for each GUI object with little or no input from the developer and creates function stubs, where the developer can insert his or her own code to add event handling code for when and how the GUI object is used in the system. Except for these places where the developer can insert his or her own code, he or she often does not see the rest of the generated code and sometimes does not even have the authority to change it.

![Figure 1-1: Drag-and-Drop GUI Development](image)

Very little research has been done on how much testing should be done on this type of automatically generated code from the GUI design tool. It is reasonable to assume that this type of generated code is extensively unit tested in a commercial IDE such as
public class Form1 : System.Windows.Forms.Form
{
    private System.Windows.Forms.Button button1;
    /* */
    private System.ComponentModel.Container components = null;

    public Form1()
    {
        /* */
        // region Windows Form Designer generated code
        /* */
        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();
            this.SuspendLayout();
            // button1
            //
            this.button1.Location = new System.Drawing.Point(200, 64);
            this.button1.Name = "button1";
            this.button1.TabIndex = 0;
            this.button1.Text = "button1";
            this.button1.Click += new System.EventHandler(this.button1_Click);
        }
    }
    protected override void Dispose(bool disposing)
    {
        /* */
        if (disposing)
        {
            components.Dispose();
        }
    }
}

Figure 1-2: Generated code for the button in Fig. 1-1

Visual Studio, but it may interact in unpredictable ways when multiple objects are combined in new ways by a developer. Thus it becomes very important to fully test these interactions because it is quite possible that the behavior of the system will be different than anticipated if new, unintended functions or choices have been added by the GUI design tool. Stürmer, Weinberg, and Conrad [11] examined how the reliability of generated code pertains to the automotive sector, which has seen large growth in the amount of generated code used in embedded safety-critical system like brake systems. These embedded systems do not have a user interface, but the way they are being developed using visual modeling tools and code generators is similar to how software developers are using GUI design and development tools to create GUIs that include large amounts of generated code. The authors found that the code generators used for these
systems must be safeguarded as part of the development process so that faults can be
detected and avoided as early as possible. They also found that most errors were based on
incorrect design models fed as input to the code generators, which again underlines the
importance of adequate testing to increase confidence in the generated code.

The hypothesis presented in this thesis is that utilizing the complete interaction
sequence (CIS) GUI testing method as described in [2] would uncover faults and
unintended behaviors that were introduced into the system by the GUI development tools
used to create the system. This thesis will investigate the additional behavior that was
unintentionally added to the system and show that it causes additional faults that will be
observed using the CIS GUI testing method. A fault is described here as the underlying
root cause of one or more defects or surprises, where a defect is defined as an error in the
system that produces unintended and incorrect results compared to how the system was
specified to work, and a surprise is defined as an unpredictable deviation from expected
behavior of the system as documented in the system specifications. Surprises are totally
new behaviors that are not addressed at all in the specifications. The specified behavior
can be determined by technical documentation such as the software specifications or user
manuals, and the severity of the defect is usually determined objectively based on how
much the behavior of the system deviates from how it was specified. The expected
behavior is more subjective based on what a typical user would intuitively expect a GUI
object to do just by how it is labeled and arranged, and surprises may often be considered
more serious than defects because of how they can affect the usability of the GUI system.
Surprises may not even produce an observable effect in the GUI. When the GUI produces
an observable effect of a surprise, such as opening a new, unintended window as the result
of some action, the user can tell right away that something is not right and can often take action to fix it or prevent further damage. However, if the surprise causes effects invisible to the user, the user will never know that something is amiss and the effects could potentially be severe enough to cause the entire computer to crash.

Consider the following examples of defects and surprises: if the user clicked a button that is labeled “Save” but no changes were made to the database or file being edited, then this would be considered a defect. If clicking that button did save the edits, but also caused the file to be printed, this could be considered a surprise since the user would not expect the file to be printed, nor would it be documented in the software specifications or other technical documentation. Further, if the button click also updated some area in memory that is supposed to be reserved for a critical operating system process, then the stability of the entire computer could be affected. Specific examples of defects and surprises as related to this thesis will be given in Chapter 5.

For the purposes of this thesis, the GUI system under test was modeled as a collection of finite-state models (FSMs) that represented the inputs and outputs of all the states of the system as well as the transitions between states, such as clicking a button to take the user to a new page. The foundation for using FSMs to test GUI systems was built by Chow [8] and has become a very common practice in many forms of GUI testing. An FSM is a model composed of one or more states and transitions between the states. There is always at least one initial state and at least one end state (multiples are possible), though the initial state may also be the end state for circular models. In terms of GUI testing, a state in the FSM can represent a specific state in the execution of the GUI with regard to the properties of each GUI object in the system. For example, an initial state of many
FSMs for GUIs may be the initialized program that displays the first screen in the GUI in which the user can take some sort of action, such as entering data or clicking a button. The transitions in the model represent the selections of GUI objects within the system that can take the user from one state to the next. For example, clicking a Save button transitions the GUI system from an edited state to a saved state. It is important to note, however, that the FSM does not capture the concept of an effect produced by the GUI, observable or not.

Figure 1-3: Open File FSM
Figure 1-3 shows an example of how an FSM can be used to model the “Open File” function typical of many GUI systems. The initial state is represented by the single circle at the top of the model. The end states are represented with double circles, and note that this FSM has two distinct end states: the first (1) shows that no change has been made to the system (no file has been opened), and the second (2) shows that the selected file has been opened by the GUI system. The rectangles represent the different states of the GUI, and the directed arrows show how the user transfers from state to state based on the descriptive actions within the rectangles.

The focus of this thesis is on the additional transitions between states that can be introduced into a system during the development phase. By using a GUI development tools such as those described earlier, it is possible that the developer would unknowingly add to the system additional transitions between states that were not intended as part of the original design or specifications. It is possible, then, that these unintended additional transitions could cause unexpected behavior that could manifest itself as a defect or surprise. The concern with these types of transitions is that they represent new behaviors and either affect the functionality or usability of the system or, even if no effects are produced, could cause problems in the future as changes are made or the system is used in new ways.

Consider Figure 1-4, which shows the same Open File model as Figure 1-3 with additional transitions that were added for this example. Additional transitions are represented as dashed arrows, and the additional states they transfer to (if applicable) are shown as rectangles with a dashed outline. In this example, imagine that the developer has unknowingly allowed the user to click on the “Save File” choice in the menu instead
of the “Open File”. Following this unintended transition could have an effect on the system ranging from the benign: the user clicks on the menu choice but absolutely nothing happens; or the very serious: the user clicks on the menu choice and an unhandled exception occurs causing the entire system to crash.

![Open File FSM with Additional Transition](image)

**Figure 1-4: Open File FSM with Additional Transition**

In this thesis, finding, testing, and analyzing these additional transitions is accomplished using the CIS GUI testing method first presented by White and Almezen in
The CIS GUI testing method involves first cataloging all GUI objects in the system such as buttons, textboxes, menu choices, etc. For each GUI object, then identify all possible states, links to other GUI objects, and even links to non-GUI components of the system, such as a database connection. This can be accomplished by using whatever information about the system is available such as user manuals, technical specifications, informal interactions with the system, or the code itself. Other studies that have developed and utilized the CIS GUI testing method have involved testing systems where the source code is unavailable, but this thesis involves using the actual source code of the system to design tests so that more insightful conclusions can be drawn specifically about where faults were introduced into the system.

The CIS GUI testing method is primarily concerned with user sequences of GUI objects and selections that collaborate to produce a desired response. The collection of objects and selections are defined as a complete interaction sequence (CIS), meaning that it covers a user’s interaction with the system from start to finish to produce the desired response, or responsibility of the system. A responsibility can be described as a GUI activity that involves one or more GUI objects and has some sort of observable effect within the system or within the environment in which the system is running. The effect may not even be observable from within the GUI, for example, the effect could be saving a file to a hard disk or updating some area of memory within the program. Using the previously defined Open File FSM in Figure 1-3, one responsibility of the system would be to open a file for viewing or editing. The CIS for this specific responsibility is then defined as the objects and transitions between them necessary for the user to open a file. Table 1-1 shows all the GUI objects that make up the Open File CIS in Figure 1-3.
achieve the desired responsibility (opening a file), the user must interact with this specific set of GUI objects in the order specified in the FSM.

<table>
<thead>
<tr>
<th>GUI Object</th>
<th>Object Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Menu</td>
<td>menu</td>
</tr>
<tr>
<td>Open File Choice</td>
<td>button</td>
</tr>
<tr>
<td>File Name Textbox</td>
<td>textbox</td>
</tr>
<tr>
<td>File Browser</td>
<td>icon</td>
</tr>
<tr>
<td>Cancel Button</td>
<td>button</td>
</tr>
<tr>
<td>Close Dialog Button</td>
<td>button</td>
</tr>
<tr>
<td>Open Button</td>
<td>button</td>
</tr>
</tbody>
</table>

**Table 1-1: GUI Objects in the Open File CIS**

As was described earlier, testing GUI systems can have combinatorial growth of test cases, and it should be fairly straightforward to see how additional transitions can quickly increase the number of tests required to fully test a GUI system. The CIS GUI testing method as described by White and Almezen in [1] and White, Almezen, and Alzeidi in [2] has been shown to reduce the number of tests needed to discover defects and surprises in a GUI system, and in fact, results in numbers of tests that grow linearly with the size of the GUI system under test. An important characteristic of the CIS GUI testing method is the use of both design tests and implementation tests to discover the defects and surprises. Design tests are developed based on how the system is designed and aim to test all the functionality as described in the system specifications. Implementation tests, on the other hand, are developed based on how the system actually works and usually contain additional tests to test additional transitions that were not included in the system design. This thesis will show that the CIS GUI testing method is very useful for testing additional
transitions that are unintentionally created in the system by using drag-and-drop GUI development tools.
Chapter 2 Background

2.1. Previous Work

The difficulty of testing GUI systems has led to a large body of research about how to improve GUI testing methods. GUI testing is particularly difficult because both the inner workings of the underlying system and the intricate relationships of the interface itself must be tested. Chen and Subramaniam discuss the difficulty of testing complicated GUI systems in [15] and describe a method to automate testing of a GUI based on its FSM-based specifications. A set of tests derived from this specification-based testing would be similar to the set of design tests derived from the CIS testing method (described in more detail below).

Another test method that has been studied extensively is the goal-driven approach first proposed by Memon, Pollack, and Soffa in [16]. In this method the test designer defines initial states and goal states of the GUI and then uses a hierarchical planner to produce a set of test cases that will reach the desired goal states. The planning techniques used in the goal-driven approach had previously been studied extensively in the realm of artificial intelligence. Memon, Pollack and Soffa [17] and Alspaugh, Richardson, Standish and Ziv [18] continued to build on this goal-driven method in later research.

Much of the research on GUI testing utilizes FSMs to model the GUI system to assist with the design and development of tests using both automated and manual methods. As mentioned earlier, the foundational work on this topic was done by Chow in [8]. Shehady and Siewiorek [14] utilized a different type of FSM called a variable finite state model (VFSM) and devised a test case generation technique that greatly reduced the
number of states (and hence the number of tests required) by introducing global variables into the FSMs to more effectively communicate the function of the FSM by making it intuitively match how the user interacts with the system.

Current testing methods for GUI systems often involve capture/replay tools [13], whereby testers can capture a specific test and replay it with the same inputs and interactions over and over. This type of testing is very useful for regression testing, particularly in systems where code is changed frequently in an iterative manner. Automating as many tests as possible allows the testers to quickly check to make sure that no new faults have been introduced. However, testers must be careful to not assume that automated testing is a “silver bullet” that will solve all of their testing problems. As Berner, Weber and Keller of [10] point out, automated testing cannot replace manual testing because most faults are found during the development of the tests.

2.2 CIS Testing Method

The CIS testing method, as first described by White and Almezen in [1], is another testing method that attacks the problem of test case generation. The method focuses on detecting faults that would be of the most significance to the user. Usability is very important in GUI systems [21, 22], and a poorly designed and implemented GUI can render a system unusable if the users do not understand how to operate it, or if their interactions with the system are so unpleasant as to dissuade them from using the system [21]. Thus, it is very useful to have a method that targets the most important aspects of how the user interacts with the system. The authors show in [1] that the CIS testing
method can be used to dramatically decrease the number of tests required to test a GUI
system such that the user’s highest priority defects will be detected.

Also in [1], the authors demonstrate that the CIS testing method is scalable to more
complex GUI systems with more GUI objects and a larger input and output base. The
authors show that the CIS GUI testing method grows linearly with the number of GUI
objects in the system. As mentioned in the first chapter, this is a dramatic improvement
over the combinatorial growth seen in the number of test cases required to test every
possible path through a GUI system.

The CIS testing method, first described by White and Almezen in [1] is described
below in more detail in sections 2.2.1 through 2.2.7. In each section below, the person
responsible for following the CIS testing method will be referred to as the “producer” to
avoid confusion with other more specific titles like “user”, “developer” and “tester” that
have a different meaning based on the different phases of software development.

2.2.1 GUI Objects

The first task of the CIS testing method is to catalog all the objects within the GUI
(such as buttons, text boxes, radio buttons, etc.). In addition, the producer must identify
all possible actions the objects can perform and document the state changes that occur as a
result of each action. In doing so, links to additional non-GUI components of the system
must be identified, such as peripheral devices like printers and alternative input devices.
The producer should use whatever resources are available to develop this catalog,
including informal interactions with the SUT, user documentation, software specifications,
or even the source code itself. Developing a comprehensive catalog of objects and actions
allows the producer to achieve a comprehensive understanding of the SUT and is a necessary building block for a more accurate and complete set of tests.

2.2.2 Responsibilities

The next step in the CIS testing method is to make sense of the system from a user’s perspective. The goal is to identify all the responsibilities of the system so that the scope of the testing effort can be defined. Responsibilities, as defined in Chapter 1, are activities in the SUT that involve one or more GUI object and perform some sort of observable effect for the user in the GUI, like changing the display, or elsewhere in the system, like saving a file to disk. In other words, the system has a responsibility to the user to perform a certain action based on the steps the user takes to achieve a desired result.

2.2.3 Complete Interaction Sequences (CIS)

Each responsibility is broken down into a corresponding complete interaction sequence (CIS). First defined and discussed in Chapter 1, a CIS is the sequence of GUI objects and actions that the user must perform to achieve the desired response, a.k.a. one of the responsibilities defined in the previous step. The CIS can be documented as if describing to the user step by step instructions on how to achieve a desired outcome. An example CIS from the SUT is as follows:

CIS #42: Delete an Assignment Group

1. (On the assignment page) Click the “Manage Groups” button.

2. Click the “Delete Group” button.
3. Select an assignment group from the drop-down list.

4. Click the Delete button.

2.2.4 FSMs

Perhaps the most difficult step in the CIS testing method is developing FSM representations of each CIS. The use of FSMs, as described earlier, shows how one can model the different states in a GUI system using states and transitions. When using this approach to model the CIS, the producer of the FSMs must be aware that a given FSM is one of potentially many FSMs, depending on both what he or she is trying to show and his or her experience developing FSMs. The right model for a CIS might even be different from the same CIS in another scenario, system, or test plan. The producer must ensure that the correct model is chosen to achieve the most effective test plan.

An example of a common GUI responsibility is opening a file for editing. Figure 1-3 illustrates how such a responsibility could be represented using an FSM. The initial state represents the point at which a user desires to open a file from some specific location.

To fully test this FSM would require 18 tests (see Appendix B for details about these 18 tests). This number is found by tracing all distinct paths of the FSM from the initial state to either final state as well as including the “File Not Found” circular path once on every distinct path where it is reached. In stark contrast, if the loop from the “File Not Found” state back to the “Click Open File” state were removed from this FSM, then only 6 tests would be required to fully test this FSM. Loops such as this one, which are prevalent in GUI systems, cause the number of test cases required to fully test a GUI to explode [19] and test design must be careful to avoid infinite loops (especially for
automated testing tools). As Marick points out in [19], complete loop coverage requires that a loop be tested zero, one, and multiple times. Even a simple loop that is used on multiple paths can greatly increase the number of tests required to fully test a GUI. However, this explosion of test cases can be avoided by excluding any path where a loop is traversed more than one time at a specific point in that path [1]. The loop may be traversed multiple times if it appears at different points in the path, but what matters most is the point on a specific path at which the given loop is traversed because different effects might be caused based on the point in the path at which the loop is traversed. Although there is no guarantee that the method of loop traversal described in [1] will always find every possible fault due to the loop, no fault has ever been missed that should have been caught.

Reaching the final state of the FSM allows the producer to observe if the responsibility of the CIS is met. In the case of the Open File FSM from Figure 1-3, the final states of the FSM represent “no change” (1) or the specified file being opened (2). These correspond to the two possible outcomes of the Open File responsibility (either the file is opened or it is not).

Chow [8] shows how FSMs can be applied to a GUI system to define the tests required to test the functionality of specific GUI objects and states and the transitions between them. White and Almezen [1] have applied this methodology to the CIS testing model to assist in the generation of both design and implementation test cases. The difficulty in converting a CIS to an FSM is that there is no direct formula to follow since each CIS is different and the possibilities for representation are numerous. The CIS testing method focuses on each individual CIS to avoid state explosion of the entire GUI.
FSM. Doing so allows the producer to clearly define the scope of the FSM: the input and output states and all the GUI objects and transitions included in the sequence. When possible, the producer may also perform a series of transformations described in [1] to develop a reduced FSM that includes fewer states than the original FSM based on properties of strongly connected components and structural symmetry.

At the highest level, the goal of using FSMs like this is to look for dependencies between objects and then test them. In this thesis the source code of the SUT was available, meaning that any suspected dependencies could be verified or disproved by looking at how the objects interact in the code. This is different from past studies using the CIS testing method where the SUTs were closed-source applications, and it helped to ensure that the right FSMs were being produced for the CIS of this system.

2.2.5 Design Tests

The initial set of tests derived from the CIS and FSMs of the previous steps is the set of design tests. The design tests are documented for each FSM using the assumption that the FSM is actually implemented in the SUT as designed by the software design team. It is important to highlight this assumption because errors in the implementation of the design are quite common and will be addressed in the next section.

Design tests for a specific FSM are developed by documenting all possible paths through the FSM from the initial state to the final state or states. A diagram from [1] is reproduced below (Figure 2-1) to assist with the explanation of this process:
The FSM in Figure 2-1 requires two design tests to fully test it. The first starts with I1: 
\{I1, B, C, D, A, B, C, O1\}. Note that this test executes the loop once which is required 
because of the discussion on loops in section 2.2.4. The second design test starts with I2: 
\{I2, A, B, C, D, A, B, C, O1\}. This test also executes the loop exactly once, and it is 
important to include the loop in this test as well as the first because the results of the loop 
may be different based on the different input state. Again, it is assumed that repeated 
extections of the loop will not produce a different result than just one execution of the 
entire loop.

2.2.6 Implementation Tests

*Implementation tests*, in contrast to design tests, are based on how the SUT is 
actually implemented and include any additional transitions that must be added to the 
FSM based on states and transitions discovered by actually using the SUT. 
Implementation tests will include all the original design tests as a subset, but also include
new tests derived from revised FSMs. A description and an example of a revised FSM are given below.

The first step in developing the set of implementation tests is to compare the SUT with the original CIS-based FSMs and add any transitions that are also available within the CIS. For example, the SUT was designed for many dialogs to have both “Cancel” and “Save” buttons, but the implementation of the system included Windows dialog “close” buttons in the upper right corner of each dialog, which performed the same action as the “Cancel” button. Each FSM that included a path to the “cancelled” state needed to be revised to include an additional path representing the user clicking on the “close” button. All the changes found during the implementation test development phase are added to the original FSMs and the result is a new set of revised FSMs.

The actual implementation tests are then derived from the revised FSMs the same way as the design tests were derived from the original FSMs. The example reproduced from [1] is continued below to illustrate this process:

![Figure 2-2: Example FSM with Additional Transitions](image-url)
Figure 2-2 contains three additional transitions: \{I3, D\}, \{D, O2\}, \{D, B\} and \{D, A^*\}. The transitions \{D, A^*\} is just like the Cancel/Close example given above for the SUT where a different GUI selection transitions between the same two states. The FSM in Figure 2-2 requires six implementation tests to fully test it:

\[
\begin{align*}
&\{(I1, B, C, D, B, C, D, A, B, C, D, A^*, B, C, O1), \\
&(I1, B, C, D, B, C, D, A, B, C, D, A^*, B, C, D, O2), \\
&(I2, A, B, C, D, B, C, D, A, B, C, D, A^*, B, C, O1), \\
&(I2, A, B, C, D, B, C, D, A, B, C, D, A^*, B, C, D, O2), \\
&(I3, D, A, B, C, D, B, C, D, A^*, B, C, O1), \\
&(I3, D, A, B, C, D, B, C, D, A^*, B, C, D, O2)\}
\end{align*}
\]

### 2.2.7 Failures

The final step of the CIS testing method is to execute both design and implementation tests and document and analyze the results. Each test failure, the observable incorrect output, can be classified as either a defect or a surprise, first defined in Chapter 1. **Defects** are faults (the root causes of failures) where the system deviates from the specified behavior in a way that causes an error. **Surprises** are faults that manifest as unintended effects based on the expected behavior of the system. Surprises are totally new behaviors that are not addressed at all in the specifications, and they may not even produce an observable effect in the GUI. Even though surprises may be unexpected for the user, they may or may not have a negative impact on the task the user is trying to perform.
As mentioned earlier, some surprises may even be more detrimental to the user experience than defects, even though many surprises may not even be noticeable by the user, such as the memory faults White and Alzeidi found in [2]. An example of a surprise that is undetectable by the user is where the system updates a block of memory in the program but never displays the updates to the user, nor has any effect on the processing that is occurring. Defects and surprises like this can only be found by using memory tools in conjunction with the execution of the design and implementation tests.

2.3 Memory Tools

Research has shown that many software defects cannot be detected without the use of memory tools because oftentimes these defects are unobservable to a user or tester [2, 20]. In [2], White, Almezen, and Alzeidi have shown that the CIS testing model can be even further enhanced by the use of memory tools to assist in detecting failures that the user may not even be aware of. These types of tools allow the tester to monitor how blocks of memory and other hardware resources (CPU, external devices, etc.) are utilized by a running program. As Fei points out in [20], even the configuration of the system on which the SUT is running may cause defects that could never be caught during system testing because of the nearly infinite number of possible configurations for most users (hardware, operating systems, privacy settings, etc.).

Consider an example of a program that stores session information about a user in memory, and the program unintentionally updates the block of memory used for the session information, but does not have anything to do with the current responsibility being performed by the system. The altered memory most likely will not result in any
observable effect for the user, but could cause a downstream defect or surprise. Perhaps the user times out of the system earlier than expected or is taken to the wrong personalized page within the system. Or in a worst-case scenario, a behind-the-scenes memory change could result in the user gaining access to another user’s information which, depending on the data stored in the system, could be a significant security breach.

Just as everything else in the software industry, memory tools are constantly evolving to give testers more information about memory usage or a better understanding of how memory is being used by the SUT. In earlier studies using the CIS testing method, the testers used memory tools like Memory Doctor, Hurricane 98, and even the Task Manager memory tools available in the Windows operating system. Most of these did not work for the SUT in this system, so two new tools were chosen: GlowCode by Electric Software, Inc. and a modified version of an open source tool called Memory Dump. GlowCode was selected because it provided detailed information about the objects being used in the SUT and also the messages passed between them, and Memory Dump allowed the tester to capture snapshots of memory usage. A small modification to the program caused these snapshots to be stored as text files instead of only existing in the context of the running Memory Dump instance. Further details about these specific memory tools will be provided in Chapter 3.
Chapter 3 Method

3.1 Configuration of the Software/Hardware Used

The hardware and software used for this study are as follows:

- All design, coding, testing, and analysis were done on a Dell Inspiron 5100 laptop:
  - Windows XP Home Edition SP3
  - 512 MB of RAM
  - 2.6 GHz Intel Pentium 4
- The development environment was Microsoft Visual Studio .NET (2002).
- The database management system was Microsoft SQL Server 2000.
- The memory tools used were
  - GlowCode
  - A modified version of Memory Dump

3.2 Testing with Source Code Available

For this thesis it was decided to test a system where all the source code was available so that a more detailed analysis could be performed about where faults were introduced. This was of course also very helpful in investigating how additional transitions that were not part of the original design were introduced into the system. This was also incredibly beneficial in designing the GUI tests so that all dependencies could be tested. Without the source code these types of dependencies must be inferred by interactions with the system, and important dependencies could be left out.
3.3 Use of Memory Tools

All tests were performed while also running two memory tools to watch system memory during the tests. The first tool, called GlowCode, is available from Electric Software, Inc. and captures a very large range of events and statistics around how memory is being utilized in the system for a specific program. There is an abundance of information available, but for this thesis only a few key features were utilized: tracking the number of objects in use as the user navigates through the system, and tracking and investigating some of the messages passed back and forth between objects.

The second memory tool used was a modified version of free source code found on the Internet for a program called Memory Dump. The program was written in C++ and worked like a camera taking a snapshot of the memory in use behind the scenes of a specific program. The original code allowed the user to select a running program and capture the memory contents and display them in a screen. The user could drill down on a specific block of memory to see a detailed byte-by-byte map of the memory values. The modifications did not affect the core functionality of this program and simply wrote the contents of the memory snapshot to a file that could be analyzed later. This was extremely helpful since analysis could be done after the fact rather than taking the time to do memory analysis of each individual test case.

Using these two tools in conjunction provided the tester with a wealth of information about how memory was being utilized in the SUT. GlowCode provided a real-time peek at all the objects being used at a given time as well as a tally of the messages being passed back and forth between these objects. Memory Dump was then used to supplement this information with snapshots of memory blocks being used by the
system at any one point. Thus the tester could monitor how memory was being used before, during, and after each test.

As the tests were executed and the memory tools were used for analysis, it quickly became clear that it would be very helpful to know which block of memory in the snapshot from Memory Dump was used for specific GUI objects in the program. This object-level mapping is not available, but it would be useful to see if this feature is available for future testing, or if not, perhaps a future study could analyze the benefits of having such detailed data about how memory is specifically used by a GUI system.

3.4 Description of the Empirical Study

One of the first steps for this study was to design a GUI system that would be complex enough to utilize different aspects of the GUI development tool which would, in theory, introduce faults that could then be discovered during testing. It may sound strange to set out designing and developing a system that is expected to include faults, but the reality of software engineering is that even the simplest of GUI systems may be prone to faults. Even simple GUIs increase in complexity very quickly as objects are added to it that can be used in different ways and sequences. The probability for finding a fault in a system increases dramatically as the GUI complexity increases because the probability directly correlates with the number of possible states or permutations of inputs and events of the GUI. Since test growth for GUIs is combinatorial, as discussed in the introduction, even a simple GUI with only a few objects will have a large number of possible states and transitions which greatly increases the probability of finding a fault somewhere in the system.
The initial proposal for the GUI system to be tested required at least 30 distinct GUI objects (such as a button or a text input box) involved in at least 20 responsibilities, which was estimated to be sufficiently complex based on the empirical findings in [1]. A moderately complex system was desired so that a significant amount of tests would be required and a number of faults would be detected. A student information system for grade-school level teachers was selected because such a system would require at least 30 GUI objects and have at least 20 responsibilities. A high-level user interface diagram showing the main responsibilities of the system is shown in Figure 3-1. Descriptions of the main responsibilities are also presented in Table 3-1. A complete and detailed description of the GUI system under test will be given in Chapter 4.

Figure 3-1: UI Diagram
<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>Path p1 corresponds to the program initialization that occurs when the user double clicks on the executable file to begin execution of the program.</td>
</tr>
<tr>
<td>p2</td>
<td>This path occurs after a successful login (or when the user selects a default user to be loaded when the program begins).</td>
</tr>
<tr>
<td>p3</td>
<td>A user can choose to logout and switch to another user from the main GUI screen.</td>
</tr>
<tr>
<td>e1</td>
<td>If the user enters an invalid username and/or password, an error message will appear asking the user to re-enter the username and password.</td>
</tr>
<tr>
<td>e2</td>
<td>This path just shows that control returns to the log-in screen.</td>
</tr>
<tr>
<td>i1</td>
<td>From the main screen, the user selects to go to the Classroom Information section. This also represents the return path to the main screen.</td>
</tr>
<tr>
<td>i2</td>
<td>This path illustrates when a user chooses to View/Edit grades.</td>
</tr>
<tr>
<td>i3</td>
<td>This path illustrates when a user chooses to View/Edit assignments.</td>
</tr>
<tr>
<td>i4</td>
<td>The user selects to import a class list from a text file.</td>
</tr>
<tr>
<td>i5</td>
<td>Control returns to the main classroom information screen after a list has been imported (or aborted). Any changes will be reflected in the main classroom information window.</td>
</tr>
<tr>
<td>i6</td>
<td>In the grades section, the user chooses to View/Edit report card grades for a class.</td>
</tr>
<tr>
<td>i7</td>
<td>Control returns to the main grades screen.</td>
</tr>
<tr>
<td>i8</td>
<td>A user chooses to add a new assignment.</td>
</tr>
<tr>
<td>i9</td>
<td>The user finishes adding a new assignment and returns to the main assignment area.</td>
</tr>
<tr>
<td>m1</td>
<td>From the main screen, the user selects to go to the Classroom Map section. This also represents the return path to the main screen.</td>
</tr>
<tr>
<td>m2</td>
<td>This path only becomes available after at least one room layout has been entered into the system. This path illustrates when the user chooses to assign seats based on a specific layout.</td>
</tr>
<tr>
<td>m3</td>
<td>When first designing a classroom map, the user must enter the dimensions of the room or choose to draw the walls manually.</td>
</tr>
<tr>
<td>m4</td>
<td>This path is executed when the user finishes entering the dimensions or chooses to draw the walls manually.</td>
</tr>
<tr>
<td>l1</td>
<td>From the main screen, the user selects to go to the Lesson Planner section. This also represents the return path to the main screen.</td>
</tr>
<tr>
<td>l2</td>
<td>In the Lesson Planner section, the user chooses to view a specific day in the daily organizer.</td>
</tr>
<tr>
<td>l3</td>
<td>The user chooses to add a new lesson/activity to the lesson planner.</td>
</tr>
<tr>
<td>l4</td>
<td>First the user must select (or create, see path l6) a template to use for adding a new lesson/activity.</td>
</tr>
<tr>
<td>l5</td>
<td>Once a template has been selected the user must then add the information to the template to create the new lesson.</td>
</tr>
<tr>
<td>l6</td>
<td>When choosing a template, the user chooses to create a new one.</td>
</tr>
<tr>
<td>l7</td>
<td>Once a new template is created, it is added to the list of templates so that the user can now use it for new lessons.</td>
</tr>
<tr>
<td>l8</td>
<td>Upon successfully adding a new lesson, the user is returned to the main Lesson Planner screen.</td>
</tr>
</tbody>
</table>

Table 3-1: UI Diagram Path Descriptions
After documenting use cases, requirements, and design specifications, it was estimated that the system would include 73 distinct responsibilities that used approximately 141 distinct GUI objects. A complete list of responsibilities is available in Appendix A. The main responsibilities listed in Table 3-1 serve as a high-level generalization of the complete list (e.g., logging in, adding a student, etc.).

Satisfied with the projected complexity, the next step was to select a development framework. The two frameworks that received primary consideration were Microsoft Foundation Classes and C# because both are widely used with extensive libraries and powerful IDEs. C# was selected because the programmer was more familiar with the C# development environment and Microsoft’s Visual Studio IDE for C#/NET. Visual Studio for C# contains a very powerful drag-and-drop editor that provided the right GUI design and development functionality that would be tested by this thesis.

The development phase of the student information system was completed by one programmer (the author). Unit testing was performed throughout the development phase as components were coded, but no unit test cases or results were documented. This would prove to be important later because failures were detected that should have been detected during the unit testing phase and it would be useful to track whether or not the particular functionality was tested and if faults were detected. For example, it would be useful to know if a specific responsibility was thoroughly unit tested and did not produce any defects, but does produce a defect in conjunction with the rest of the system.

The design of the student information system called for a database backend, and SQL Server 2000 was chosen because it integrates well with C# and the Visual Studio IDE since all are Microsoft products. This was particularly useful because the GUI
development tool contained drag-and-drop features related to database connections and query and result set objects. The programmer could drag a database adapter object onto the screen being worked on and, after supplying a few configuration parameters like the username and password to connect to the database, could utilize query and result set objects to interact with the database from controls on the screen or in the code behind the scenes.

Once the code for the program was completed, the next step was to identify all the GUI objects that were actually used in the system. Using the list of responsibilities that was generated during the design phase, the complete interaction sequences that equated to each responsibility could be modeled using finite state models. As described in [1] and [2], the FSMs could then be used to create a set of design tests to test the behavior of the system as designed.

The next step in the CIS testing method was to investigate and uncover additional transitions in the FSMs that could be used to generate a set of implementation tests. This required an extensive overview of the system as programmed to see what options and paths the user is allowed to take from any given point in the interaction with the system. For example, when adding a new student it was found that the user could also click on the "Add Parent" button that was hidden behind the "Add Student" dialog if the user resized, minimized, or moved the "Add Student" dialog. It was important to find all choices that were available from any particular step in a CIS, even if they do not make any sense, because the additional transitions that were expected would probably not be logical choices that a user would make.
These additional transitions were then incorporated into the original finite state models to produce a set of revised FSMs. Oftentimes the existing FSMs could be utilized and the new transitions could be added in a different color to set them apart, but at times it was necessary to recreate the entire FSM because the new transitions completely changed the FSM.

Testing was again performed by one person (the author) to ensure that the same methodology was used across all test cases and so that failures and other results would be documented consistently. The test cases and their corresponding results were tabulated in an Excel spreadsheet (included as Appendix C).

For testing and analysis purposes, the uncovered defects were further categorized with a rating scale that corresponded to the severity of the defect. For example, some defects were purely cosmetic in nature and did not undermine any functionality of the system, whereas others were quite severe and may have even caused the entire program or system to crash. Table 3-2 illustrates the rating system used:

<table>
<thead>
<tr>
<th>Rating</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cosmetic – no impact to functionality</td>
</tr>
<tr>
<td>2</td>
<td>A workaround is available to achieve the functionality that is broken</td>
</tr>
<tr>
<td>3</td>
<td>No workaround is available, but a simple solution could be designed to fix the defect</td>
</tr>
<tr>
<td>4</td>
<td>Program needs major rework – the program becomes unusable or suffers a major loss of functionality</td>
</tr>
</tbody>
</table>

Table 3-2: Rating Scale for Defects
Finally, the results were reviewed and additional analysis was performed to determine the original faults behind each defect and surprise. Each fault was then investigated to determine where it was introduced and how it could have been avoided. Some example questions that were used for the analysis phase are shown below, with full analysis available in Chapter 5:

- Were any faults introduced in the design phase where the design itself was incorrect?
- What faults/behaviors were introduced that have nothing to do with the original design?

### 3.5 Threats to the Empirical Study

There were a number of minor threats to this empirical study, though none were deemed too significant to endanger the findings. First of all, for basic security and maintenance reasons, periodic Windows updates were installed on the development/testing system that were flagged as "Critical" updates by Microsoft. This means that the underlying system was changed slightly during testing, but the risk was considered negligible.

Another potential threat was the length of time it took to define the design and implementation tests and then execute all of them. With any testing effort the goal is to complete the testing in as little time as possible for very practical reasons such as minimizing the cost or the manual effort of the testing, or meeting release dates. It is also beneficial to work on the system as much as possible so that one can have the most comprehensive understanding of the system, which helps to better identify less obvious
defects and surprises and also makes it easier to determine the fault behind the failure. In this study the testing was spread over a number of semesters, and as such, additional time was required at different periods to review the system and work that had been accomplished up until that point.

As mentioned earlier, all programming and testing was performed by one person: the author of this thesis. The benefit of this approach is that the author has the most thorough understanding of the system and the testing methodology. There are risks with this approach, however. In the programming phase, having one person responsible for all development and unit testing increases the potential for certain errors to slip through the cracks. A common business practice is to have others review code and unit testing so that someone with a different perspective than the developer can try to catch some errors that would otherwise have been overlooked. Also, system testing is often performed by someone other than the developer to prevent any sort of bias in the test approach. The developer may unknowingly avoid or workaround areas of known complexity or trouble spots that have already been reviewed extensively. However, the CIS testing methodology is a very objective approach to testing that reduces the risk that faults would be overlooked or missed, even with some sort of unintended developer bias. The risk that any one area would be overlooked or under-tested is minimal because the CIS method tests all aspects of the SUT as implemented.

Even though the SUT was designed as a multi-user system, most tests were performed as the same user, and no testing was done while having multiple users logged into the system and using it at the same time. There is a risk that some faults would only manifest as failures during this type of concurrent testing, but in practice this type of
testing would be done as part of some sort of load testing initiative that would simulate multiple users to test for exactly these types of faults.

A more significant risk that plagues any FSM-based testing method is the question of whether or not the FSMs representing the SUT completely define all possible states and transitions in the SUT. There is a slight risk that some obscure state or transition was missed in the development of these FSMs, but the risk is minimal because the system was exhaustively explored to develop the FSMs, and if nothing else was discovered, then it is very unlikely that regular users would uncover any undiscovered transitions or states in regular usage. Because the source code was analyzed in this particular study, it was also determined that no sections of code exist that were not represented by the final FSMs.

The effects of incomplete FSMs on the CIS testing method would be that the implementation test suite is not complete and additional faults could potentially exist that are never tested for. However, the cost in time, money, and effort to discover these types of faults is prohibitive for most software systems.

A related question is that, given adequate FSM models, how does one know that each FSM has been sufficiently tested? This is again a small risk since the method for developing these FSMs was to thoroughly explore the SUT as well as the source code behind it. Although it is possible that some FSMs may not have been sufficiently tested, the risk is minimal that additional faults would be discovered, and as mentioned above, the cost to guarantee that all FSMs are completely tested is prohibitive for most software systems. Also, many tests produced the same failures, and analysis showed that they traced back to the same fault. This indicates that if any tests were missing, the failures discovered by them may very well have already been detected by existing tests.
Chapter 4 GUI Design and Implementation

The student information system under test for this thesis was designed as a computerized aid to early education teachers. Its four main functions are to record grades, record student-specific information, supply a diagramming tool to design classroom layouts, and provide a calendar-like lesson planner to assist in scheduling lessons and activities.

The system was designed such that multiple users could utilize one relational database through individual instances of a thick client that could be installed on any Windows-based personal workstation that also has the .NET framework installed. Details about the data structure of the shared database are given below.

4.1 Data

Figure 4-1 shows the entity-relationship diagram for the system database. The key entities are users, students, classes, assignments, and lessons. Each entity and relationship in the diagram is described in the sections that follow.

4.1.1 Users

Each user (teacher) requires a user name and password and a unique user ID is assigned upon creation of a user.
4.1.2 Classes and Classrooms

A user is related to a class through the TeacherClass relationship that pairs a class ID with a user ID. Each user can have 0 to \( n \) class entities. The unique class ID is generated when a user creates a new class in the system. A class can be associated with only one classroom by storing the classroom ID as a foreign key on the class entity.

Classrooms are created by users and are unique to each user by storing that user’s ID as a foreign key on the classroom entity.

Classes:

- Title
- StartTime
- EndTime
- ClassSize
- Notes

Classrooms:
- RoomNum
- Length
- Width

4.1.3 Students

Each student has a unique identifier (student ID) that is generated at the creation of a student. Students are related to class entities by storing the student ID and the class ID as a key pair on the Roster relation entity. The student entity stores the following personal information attributes for each student:

- Name (First, Middle, Last)
- Home address
- Phone number (Home, Work, Cell/Pager)
- Bus numbers (AM and PM)
- Special needs
- Notes
- Scanned picture
4.1.4 Parents/Emergency Contacts

Each student can have 0 to \( n \) parent entities, and the relationship between the two is set up with the student ID as a foreign key stored on the parent entity. An emergency contact entity is exactly the same as a parent entity with the exception of an additional attribute called “Relation” that is used to describe the relationship between the contact and the student; otherwise the same data and rules apply to emergency contacts. Just as with parent entities, an emergency contact is related to a student by storing the student ID as a foreign key. The following data elements are stored for each entity:

- Name (First, Middle, Last)
- Address
- Phone Number (Home, Work, Cell/Pager)
- Email address
- Notes
- *Emergency Contacts only: Relation to student (Aunt, Grandmother, etc.)*

4.1.5 Assignments and Grades

Assignment entities are linked to users and classes. Each assignment ID has a unique identifier that is generated at creation of the assignment. The user and class IDs are stored as foreign keys on the assignment entity so that a user can easily work with assignments in a specific class. Assignments can be grouped together in an entity called AssignmentGroup where the AssignmentGroup has a primary key pair made up of its own unique ID and the assignment ID. Assignment groups can be weighted toward the overall grade for a class by using the Percentage value (all assignment group percentages should
Individual assignments are also related to students through grade entities. The grade entity uses the student ID and assignment ID as its own primary key pair.

Assignments:

- Title
- Type
- MaxScore

Assignment Groups:

- GroupName
- Percentage

Grades:

- Grade
- Notes

4.1.6 Lessons

The lesson planner is organized in a calendar format and individual lessons may be added, edited, or deleted in much the same manner as many calendaring tools currently available, such as the calendar product inside Microsoft Outlook. Individual lesson entities are identified by a unique ID generated when they are created and are linked to users and classes by storing both user and class IDs as foreign keys. Lessons contain the following attributes:

- LessonDate
- Title
4.2 System Interaction

Details about how a user interacts with the system’s main functions are provided below.

4.2.1 Program Initialization

The user starts the program by double-clicking on an executable file which brings up the main login screen. An example screenshot for the main startup dialog is shown in Figure 4-2.

![Login dialog]

Figure 4-2: Main Startup Dialog
4.2.2 Log-in

A user logs into the system using his or her unique user name and password. This, combined with the centralized data source, allows multiple users to use the system simultaneously. If the user chooses, the username and password can be saved as a default, thus bypassing the login screen for future sessions.

4.2.3 Main Class Screen

Once logged in, the user must choose a specific class to use. Each user may create a class in the system corresponding to each class he or she teaches. The user selects a specific class from the list of classes shown on the main screen and the class is set so that other functions and data appear for that specific class, such as the daily lesson planner. The user also has the option of setting a specific class as the default class to be selected automatically for future sessions. Every time the user uses the system, he or she will automatically be working with the default class (i.e., the class he or she is currently teaching).

An example screen shot of the main class screen is shown below in Figure 4-3. From this screen the user can choose to view the class roster, work with the Classroom Map function, or open up the Lesson Planner for that class. These threads of control are described below.

4.2.4 Class Information

This section of the program is used to record information in the database about students in a specific class. Student data is displayed in an editable table format (like a
spreadsheet) and additional functions may be selected by two rows of buttons, such as adding a parent for a student or importing a class list from a data file. An example screenshot of the main class information page is shown in Figure 4-4.

![Main Class Screen](image)

**Figure 4-3: Main Class Screen**

### 4.2.5 Creating a Class List

The user may create a class list for a specific class by either importing an entire list from a data file or adding each student manually. To import a student list from a file, the user must click on the “Import Student List” button on the main class information page. Upon doing so, the user must browse for the text file to be imported using a standard open file dialog such as that found in Microsoft Office products. The file must be in comma-
delimited format for the records to be inserted properly. The class information screen immediately reflects the changes once the file is imported and the database is successfully updated.

The user may choose to add information for each student manually. First the user must click the “Add New User” button, which will open the Add Student dialog. The user can then type in the information for each field until all the necessary information is entered. The user must then save the new student, which will insert the data into a row in the student information table (Figure 4-4) and into the database, provided it meets all the necessary constraints. If it does not, a warning will be issued to the user, and the user must correct the errors before submitting the data to the database again.

Students can be completely removed from the system by highlighting a student in the student information table (Figure 4-4) and pressing the Delete key.

4.2.6 Editing Student Information

Each student’s information is directly editable in the display window. The user simply has to select a cell in the table that requires editing and then edit the information as necessary. The user can complete changes in an individual cell by pressing the “Enter” or “Tab” key or by selecting something else in the window with the mouse. Changes will not be finalized in the database, however, until the user clicks on the “Save Changes to Database” button. This prevents inadvertent editing of data in case a user accidentally selected a field and made unintended changes.
4.2.7 Editing/Viewing Parent Information

The user may view information about a particular student’s parents and/or emergency contacts by highlighting a student in the student information table (Figure 4-4) and then clicking on the button labeled “View Parent/Emergency Contact Information”. This spreadsheet-like display will allow adding parents or emergency contacts and editing their information in the same fashion as adding/editing information for a particular student.
4.2.8 Grades

The other main use of the class information section of the program will be for the user to record grades for each student. The user must first choose a specific assignment to record grades for individual students. Interim, quarter, midterm, semester, and final grades can all be recorded as individual “assignments”.

4.2.9 Add/Edit Assignments

To add an assignment, the user must click on the “View/Edit Assignments” button from the class information screen. Next the user must click on the “Add New Assignment”, which brings up a dialog where the user is required to enter a Title, Description, and Max Score. The user clicks on “Add Assignment” to save and add the new one.

4.2.10 Assignment Groups

The user can choose to organize assignments into groups. These groups can be used for organizational purposes or to assist in calculations of final weighted grades. For example, all spelling test scores could be combined at the end of a term and factored into the final grade as 20% of it.

To create or edit assignment groups, the user must click on “Manage Assignment Groups” from the “View/Edit Assignments” page. The user may add a new group by clicking on the “Add New Assignment Group” button and filling out the details in the dialog that opens. A Group Name, Weight (of overall grade), and Initial Assignment are required. The assignment groups are displayed on the page, and the user can directly edit
the Weight field and edit the name by clicking on a button called “Edit Group Name”.

The user can also add assignments to and delete assignments from the group by using the Add and Delete buttons. The user can add to a group any Assignment in the database that is associated with their user ID but may only delete Assignments from within the group. Deleting assignments from a group does not delete the Assignment from the database; it just removes the group relationship. Lastly, the user may delete whole Assignment Groups by clicking on the “Delete Assignment Group” button. A dialog pops up with a dropdown list of groups, and the user must select a group and click on the “Delete Group” button to complete the transaction.

4.2.11 Viewing/Editing Grades

Grades can be viewed and edited on a per assignment basis. The user enters the Viewing/Editing Grades page by clicking on the “View/Edit Grades” button on the Classroom Information page. The user must then select an assignment from a dropdown list. Selecting an assignment populates a grid that displays each student as a row in a table which contains cells in the row for Score, Grade, and Comments. The title of the assignment and all relevant information appears at the top of the screen. The user types the correct grade for each student and can add comments, if desired. The user must click on the “Save Changes to Database” button to save any grades that were entered for the assignment.

As mentioned before, interim, quarter, midterm, semester, and final grades can all be recorded as individual “assignments”. The user simply creates an assignment with the
appropriate title, such as “3rd Quarter Grade”, and then inputs grades and comments in the table as described above.

4.2.12 Classroom Map

The Classroom Map section of the program allows the user to create layouts for his or her classroom(s), as well as assign seating charts for the students. An example screenshot of the main Classroom Map screen is shown in Figure 4-5.

![Classroom Map screenshot]

**Figure 4-5: Classroom Map**
The first step in using the Classroom Map tool is to define the boundary of the user's classroom. The user can accomplish this by either drawing walls manually using the Wall object or by entering room dimensions to have the walls drawn to scale automatically. For the second option, the user must click on the “Define Dimensions” button and then input the length and width of the classroom. Upon clicking OK the walls will be placed on the drawing area according to the values entered in the dialog. If either or both values are left empty in the dialog, then an error message will be displayed, prompting the user to input both values.

A user may also choose to open an existing diagram for editing. Using the File > Open option from the menu at the top of the window, the user can browse to a saved diagrams and open it for further viewing and editing.

4.2.13 Adding Objects

The next step in using the map tool is to add different objects to the classroom. A drawing toolbar is displayed on the left side of the screen that shows the different objects a user can enter into to grid area, such as a desk, table, or chair (see Figure 4-5). The user can add these objects by clicking the appropriate button in the toolbar and then clicking the area on the map where the object should be placed.

4.2.14 Deleting Objects

Delete functionality was included in the initial design, but was not implemented. The user must clear the drawing area using the “Clear All” button or create a new drawing to remove objects from the diagram.
4.2.15 Saving and Printing

When finished diagramming the classroom, the user can choose to save the diagram in one of two ways. First, he or she can select File > Save from the menu at the top and save the drawing as a classroom map (.map) file. The other way to save is to select the “Save as Seating Chart” button and save the diagram as a seating chart (.cht) file that can be used for assigning students’ names to chairs or desks.

Users can print the diagrams, saved or unsaved, by either selecting File > Print or by clicking on the “Print” button on the screen.

4.2.16 Seating Charts

This section of the system can also be used to assign seating charts for the students. The user must first select the “Assign Seats” button from the main program screen, which will bring up the classroom diagram. The user must create or open a classroom map (.map) or seating chart file (.cht) and then he or she can assign students to specific seats in the classroom. A drop-down list at the top of the page is populated with all the students in the class and the user simply needs to select a name from the drop-down list and then click on the seat, desk, or table where he or she would like to assign the student to sit. The user can then save and/or print the current seating arrangement in using the same options as described for the classroom diagram in section 4.2.16.

4.2.17 Lesson Planner

The basic layout of the Lesson Planner, shown in an example screenshot Figure 4-6, is set to show all the activities that a user has scheduled for a certain day. From this
main screen the user can choose to view the current day’s activities, add new lessons, or select to view the activities and lessons from any day by use of a calendar. Each day will further be broken into time segments that represent the activities for that day.

Figure 4-6: Lesson Planner

The main screen of the Lesson Planner shows how the day’s events are displayed for the user. The user can view times earlier or later than those shown by scrolling up or down in that section. Also, the user can select a different day to view by either cycling through the days sequentially using Next and Previous buttons or by navigating through a small calendar control to choose a specific day for viewing. Once a day is chosen the
view will be updated to show the events, if any, for that day. The user can also select a
Print button to print the events for a particular day.

4.2.18 Adding/Editing Lessons

The user may choose to add a new lesson or event to the Lesson Planner by
clicking the “Add New Lesson” button. This opens a dialog that contains textboxes for
the fields the user can edit such as title, start and end time, etc. Once all the information
has been entered, the user can save the lesson and thus add it to the Lesson Planner
calendar by clicking on the “Add Lesson” button in the dialog.

The user can edit the events displayed in the daytime planner section by double-
clicking on a specific event in the day calendar. This will take the user to a new screen
that gives him/her access to the different fields for an event. The user can make changes
as necessary and then save the changed event, which will be reflected in the daytime
planner.
Chapter 5 Results

Listed below are the numbers and raw data for CIS, tests, faults, and transitions.

5.1 CIS, FSMs, and Tests

Total number of CIS: 73
Total number of GUI objects in all CIS: 141
Total number of FSM (some represent multiple CIS): 58
Total number of design tests: 342
Total number of implementation tests: 677
Total number of implementation tests that test how the system differs from its design: 355
Tests/CIS: 9.27 (677 / 73)

The numbers above give a rough idea of how complex the SUT is by showing the CIS in the system and all the tests required to fully test them. Either the number of GUI objects or the number of CIS can be used as a measurement of the relative size of the GUI system. The implementation tests include the design tests as a subset, and the tests/CIS give a rough comparison of the complexity of each CIS. Having 9.27 tests/CIS is somewhat high compared to previous studies that utilized the CIS testing method. These earlier studies had tests/CIS metrics around 2.00 with a previous high of 6.88 tests/CIS for a GUI system called EasyPhoto [1][2]. This would seem to indicate that either the GUI SUT was more complex than some of the earlier SUTs, or that the FSMs selected to model the CIS were not as simple as models chosen in earlier studies.
5.2 Surprises, Defects, Faults, and Level of Contamination

Total number of unique surprises and defects (sum is total number of faults): 47. *Note that some surprises and defects were detected by more than one test.

Total number of unique faults detected by design tests: 37

Total number of unique faults detected by only implementation tests (i.e., not due to design problems): 10

Level of contamination:

- Faults/CIS: 0.644 (47/73 = 64.4%)
- Faults/Test: 0.069 (47/677 = 6.9%). Conversely, it took an average of 14.4 tests to find each fault.

The level of contamination metrics are roughly in line with those seen in earlier studies using the CIS testing method. White and Almezen [1] discussed a rough estimate of the level of contamination of a system where the total number of faults found is divided by the total number of CIS in the system. This number gives a very rough idea of the quality of the software by indicating the percentage of functions that are contaminated with faults. In this system 47 distinct faults were found in 73 complete interaction sequences which equal a level of contamination of 0.64. This indicates that the SUT is quite contaminated compared to the levels of contamination found in White and Almezen’s earlier studies, where the faults/CIS was typically between 0.05 and 0.50. The 0.64 level in this thesis makes sense considering the SUT was brand new (never tested or used), and was developed by a relatively inexperienced developer. It would follow that subsequent testing of newer releases of the system would likely be less contaminated as
existing faults are addressed. The rate of new faults could be determined in regression
testing by using a regression testing method such as the firewall method described by
White, Almezen, and Sastry in [3].

Interestingly, the number of tests/CIS (9.27 – see Table 5-1) is a little high
compared to values seen in the previous studies using the CIS testing method. Developing
FSMs for a CIS is something of an art form since a system can be represented as multiple
FSMs depending on where the developer of the FSMs decides to set the boundary for the
CIS being modeled. As was mentioned earlier, loops can greatly impact the number of
test cases required to fully test an FSM representing a CIS. One possible explanation for
the higher number of tests per CIS is that the SUT for this thesis was more complex than
those used in previous studies in terms of possible transitions from each state. It is also
possible that more loops were included in the design of the FSMs for this system.

Related to this is the faults/test metric, which tells a little about the efficiency of
the tests. A higher value for this metric indicates that fewer tests are needed to detect
faults in the system. In the previous studies the faults/test metric has varied from 0 to 0.5,
whereas the value for this thesis is 0.069. The number of design and implementation tests
used in this thesis is higher than most of the test sets used in previous studies, so it is not
surprising to see a fairly low fault/test metric.

Lastly, as in previous studies, Table 5.1 shows that the faults/test for design tests is
greater than for implementation tests as many more implementation tests are consistently
needed to detect additional faults. Table 5-1 summarizes the GUI system under test and
the results of applying the CIS testing method:
<table>
<thead>
<tr>
<th>No. of GUI Objects</th>
<th>No. of CIS</th>
<th>Test Type</th>
<th>No. of Tests</th>
<th>Failures from Defects</th>
<th>Failures from Surprises</th>
<th>Distinct Faults</th>
<th>Faults/CIS</th>
<th>Tests/CIS</th>
<th>Faults/Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>141</td>
<td>73</td>
<td>Design</td>
<td>342</td>
<td>121</td>
<td>37</td>
<td>37</td>
<td>0.644</td>
<td>4.68</td>
<td>0.108</td>
</tr>
<tr>
<td>Impl</td>
<td>677*</td>
<td></td>
<td>123*</td>
<td>48*</td>
<td>47*</td>
<td></td>
<td></td>
<td>9.27</td>
<td>0.069</td>
</tr>
</tbody>
</table>

*Implementation tests include Design tests as a subset

**Table 5-1: Summary of Results**

Below is a small subset of all the defects and surprises uncovered by these tests, included here to illustrate the types of defects and surprises found in the system. For a full synopsis of all the defects, surprises, and underlying faults refer to Appendix C:

**Defects**

- The user needs to click the “Log in” button twice to log into the system.
- New users with the same usernames as existing users are saved to the database if a different password is supplied.

**Surprises**

- When utilizing the “undo” function, only 10 characters of typing were removed when it was expected that all the characters in the textbox would be removed.
- The timer function on the main page is continuously called even when the user is in other pages in the system.

**5.3 New Transitions**

Total number of FSMs with new transitions (found by implementation tests): 41

Percentage of FSMs with new transitions: 70.7%  (41/58 total FSMs)

Total number of transitions in these FSMs: 313

Total number of FSMs with new transitions that contained faults that were discovered by implementation tests alone: 10
Total number of new transitions in this subset of FSMs: 73

Total number of new transitions in this subset of FSMs that caused defects or surprises in the system: 13

Total number of FSMs with new transitions that did not lead to faults: 31

Total number of new transitions in this subset of FSMs: 300

The numbers above illustrate how FSMs with new transitions do not always contain new faults. Of the 41 FSMs that had new transitions, only 10 contained faults, while 31 were benign in that they introduced new behavior but did not introduce any negative consequences. These 31 FSMs are not of particular interest in this thesis because they did not contain any faults; however, they do represent behaviors that are different from the design and should not go without mention. These FSMs are discussed further in Section 6.3.

Figure 5-1: “Edit Student Information and Save” FSM with New Transitions
An example of a CIS ("Edit Student Information and Save") with new transitions found by implementation tests is shown in Figure 5-1. The FSM is shown in its original form with new transitions highlighted as dashed lines. The new transition that caused a failure (#5 in Section 6.2) is highlighted with a red, heavier dashed line.

Table 5-2 details all the CIS (and corresponding FSMs) where additional transitions were found. Note that some FSMs accounted for multiple CIS because the same objects were used. For example, CIS 1, 2, and 3 shared the same GUI objects and all three responsibilities could be met by taking different paths through the FSM.

Table 5-3 is a brief summary of the 13 additional transitions that led to the 13 failures discovered by implementation tests alone. These 13 failures traced back to 10 distinct underlying faults. Of special importance is the column with the label “Introduced by GUI Design Tool”. Note that 9 of the failures were found to be introduced by the GUI design tool whereas the remaining 4 were simply explainable errors in the implementation that had nothing to do with the GUI design tool.

Also presented in Table 5-3 is the column titled “Require additional transitions to fail?” with Yes or No values for each row. A No value indicates that the user could select this transition in the system and cause a failure without any other new transitions being used. A Yes value, on the other hand, would only be reachable after the user followed some other additional transition in the revised FSM. For example, the failure discovered by Transition 4 occurred when the user clicked on a Save button, but this option is only available to the user after he or she has clicked on the Add New Student button, which is another new transition in the FSM.
<table>
<thead>
<tr>
<th>CIS</th>
<th>Total Number of Add'l Transitions</th>
<th># of Add'l Transitions that Caused Failures</th>
<th>Failures caused by Defects</th>
<th>Failures caused by Surprises</th>
<th>Shared FSM?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CIS 1, 2, and 3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CIS 1, 2, and 3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CIS 1, 2, and 3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td></td>
<td>CIS 31 and 32</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CIS 31 and 32</td>
</tr>
<tr>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>CIS 43 and 44</td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CIS 43 and 44</td>
</tr>
<tr>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>11</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTALS</td>
<td>313</td>
<td>13</td>
<td>2</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Table 5-2: Additional Transitions
<table>
<thead>
<tr>
<th>Transition #</th>
<th>CIS</th>
<th>Severity</th>
<th>Fault #</th>
<th>Introduced by GUI Design Tool?</th>
<th>Require additional transition(s) to fail?</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>2</td>
<td>38</td>
<td>Yes</td>
<td>No</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #38).</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2</td>
<td>38</td>
<td>Yes</td>
<td>No</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #38).</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td>2</td>
<td>39</td>
<td>No</td>
<td>No</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #39). Similar transitions (CIS 20, 21, 27, 28) did not cause any failures.</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>4</td>
<td>40</td>
<td>Yes</td>
<td>Yes</td>
<td>This transition was not introduced in any other FSMs. Caused a defect (fault #40).</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
<td>4</td>
<td>41</td>
<td>Yes</td>
<td>Yes</td>
<td>This transition was not introduced in any other FSMs. Caused a defect (fault #41).</td>
</tr>
<tr>
<td>6</td>
<td>37</td>
<td>2</td>
<td>42</td>
<td>No</td>
<td>Yes</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #42). A similar transition in CIS 38 did not cause a failure.</td>
</tr>
<tr>
<td>7</td>
<td>37</td>
<td>2</td>
<td>43</td>
<td>No</td>
<td>Yes</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #43). A similar transition in CIS 38 did not cause a failure.</td>
</tr>
<tr>
<td>8</td>
<td>39</td>
<td>2</td>
<td>44</td>
<td>No</td>
<td>Yes</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #44).</td>
</tr>
<tr>
<td>9</td>
<td>48</td>
<td>2</td>
<td>45</td>
<td>Yes</td>
<td>No</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #45). Transitions 9 and 10 are similar. A similar transition in CIS 39 did not cause a failure.</td>
</tr>
<tr>
<td>10</td>
<td>49</td>
<td>2</td>
<td>45</td>
<td>Yes</td>
<td>No</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #45). Transitions 9 and 10 are similar. A similar transition in CIS 39 did not cause a failure.</td>
</tr>
<tr>
<td>11</td>
<td>70</td>
<td>2</td>
<td>46</td>
<td>Yes</td>
<td>Yes</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #46).</td>
</tr>
<tr>
<td>12</td>
<td>70</td>
<td>2</td>
<td>46</td>
<td>Yes</td>
<td>Yes</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #46).</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>47</td>
<td>Yes</td>
<td>No</td>
<td>This transition was not introduced in any other FSMs. Caused a surprise (fault #47).</td>
</tr>
</tbody>
</table>

Table 5-3: Summary of Additional Transitions that Led to Failures
A detailed analysis of each transition, its corresponding failure, and the underlying fault is available in Section 6.2. Note that failures that correspond to a shared fault are shown in the table in order and share the same Fault #.
Chapter 6 Analysis of Raw Data

6.1 Design Test Results

The design tests for this system uncovered a majority of the faults: 78.7% of the faults were discovered by design tests alone (37 out of 47 distinct faults). This is not much of a surprise since the developer was relatively inexperienced, and unit and system testing were both completed by the developer (neither of which were documented). It is likely that more “production-ready” code would yield a lower percentage of faults discovered by design tests alone as this had been the case in earlier studies [1, 2].

6.2 Faults Found by Implementation Tests Alone

Nearly half of the total number of tests were implementation tests that tested where the system differed from its design: 335 out of 677. Only 13 failures with 10 distinct underlying faults were detected by implementation tests compared with 37 distinct faults found by the design tests. The implementation tests doubled the testing effort, and since these tests were all run manually, it added a considerable amount of man-hours for the running of the tests and the subsequent analysis of the results. (This is further evidence for the necessity of automated testing for GUIs as discussed in [10] and [14].) It should also be noted that the 10 faults uncovered by implementation tests alone were severe enough that this system should not be considered ready for release. Using the severity scale in Table 3-2, three of the ten faults were deemed cosmetic (level 1 severity), five faults had workarounds available (level 2), and the remaining two faults caused system crashes (level 4), which is unacceptable for users.
Below is a detailed analysis of the 13 new transitions that generated failures found by implementation tests alone. There are 10 unique faults shared across 13 transitions and their corresponding failures. Three pairs of transitions share common faults: transitions 1 and 2, transitions 9 and 10, and transitions 11 and 12.

**Transition 1: (CIS #7, Fault #38) Caused the surprise “Additional Add User dialogs stay open and each works as expected” - Introduced by the GUI design tool**

The new transition introduced in this FSM was that the user could click on the Add New User button a second time after already clicking on the Add New User button and opening the Add New User dialog. The result was that two Add New User dialogs were opened which, although it did not cause any errors, could be confusing to the user.

This and many other transitions were introduced because the GUI design tool allowed the developer to open modal dialogs while not locking the controls in the main window. A modal dialog is a popup dialog that has a Boolean return value, meaning that exiting the dialog can only return either True or False. As such, no other processing in the system should occur until the modal dialog is closed. The system as a whole, and specifically this CIS, were designed such that a user would execute a responsibility until it was either completed or cancelled. The user was not supposed to be able to start any other CIS until the current one was either completed or cancelled. The GUI design tool should have prevented this surprise by automatically locking the other controls upon opening a modal dialog.
Transition 2: (CIS #7, Fault #38) Caused the surprise “Could click on any button or menu choice after clicking Add New User” - *Introduced by the GUI design tool*

The root cause of this surprise was the same as described in Transition 1 above. Transitions 1 and 2 can be traced back to the same fault in the system. In this FSM, the user could also click on any menu choice in the parent window after opening the Add New User dialog. The negative effect for the user is that he or she might be confused by having multiple dialogs open, or he or she may forget to finalize an Add New User transaction by clicking on the “Add User” (save) button in the original dialog and thus not create the new user as intended.

**Transition 3: (CIS #26, Fault #39) Caused the surprise “Things to Do Today section does not get updated by changing date or class” - *Introduced by the developer; no relation to the GUI design tool***

This surprise is caused by an additional transition that allows the user to select a different class in the class selection listbox after selecting a day on the calendar to show “Things to Do Today.” The specific surprise occurs when fewer events exist for the newly selected class than were loaded in the calendar for the previously selected class. When this happens, any events from the previous class greater than the new ones are still displayed in the “Things to Do Today” section. For example, if the previous selection displayed 5 events, and the new selection only contained 3 events, the first 3 items in the list would be updated for the new events but the 4\textsuperscript{th} and 5\textsuperscript{th} items from the previous selection would still be displayed. The negative consequence for
the user is that he or she might think the events from the previous selection are related to the current selection.

This new transition, though not part of the original design, was not introduced by the GUI design tool, but rather was simply not caught during the design phase. The root cause of the issue is a programming error and should have been addressed by the developer, not by the GUI design tool. The "Things to Do Today" section is loaded during the MainForm object's constructor and updated in the object’s "_Enter" event. The initial values will be overwitten only if new items are found for the new class. Otherwise, the original ones that were loaded at startup will still be displayed in the window because the label GUI objects are never reset when a new class is selected.

Transition 4: (CIS #31, Fault #40) Caused the defect “New student saved, but changes to data grid were not” - Introduced by the GUI design tool

When a new student is successfully added to the system, the dialog used to add the student returns a success value (DialogResult.OK) to the calling function. This triggers a call to the function updateGrid() so that the new student will be displayed in the student grid on the page. In this test case, the user then clicks on the "Save Changes" button to save edited information in the grid that was made before the new student was added to the grid. The "Save Changes" button then calls a function to update the student data set. The issue is that the original data set object (dsStudents1), which contains the edits made in the first part of the test, is different than the data currently in the grid as a result of adding the new student in the second half of the test.
The data set object that the system tries to save to the database is corrupted and the database connector object throws an exception that crashes the program.

This all happens because the GUI design tool introduced an additional transition that allows the user to perform a second action (adding a new student) while in the middle of an action (editing an existing student). There were other additional transitions on this page in the system that did not result in any kind of defect or surprise because the system was acting on unrelated components. However, in this case both actions were utilizing the underlying data set object that was common to both actions.

This transition can only be reached if the user follows another additional transition in the FSM that does not cause a failure by itself. The user must first click on the Add New Student button, and then the failure occurs when the user clicks on the Save button. The GUI design tool could be changed to help in situations like these by setting the default behavior for DataGrid objects so that, if information in the grid is changed, some sort of warning message is displayed to the user if he or she takes any other action outside of the grid before saving the changes. This would prevent the user from clicking on the Add New Student button that started the chain of events that cause this failure. With a solution like this, the programmer should obviously be able to override this default behavior and have the ability to customize the message and how it is displayed, but this seems like it would be a useful default behavior for many systems that utilize the DataGrid in the same way as this student information system.
Transition 5: (CIS #34, Fault #41) Caused the defect “Add new student and navigate to parent screen: nothing was saved, program eventually crashed” -

*Introduced by the GUI design tool*

This defect occurs when the user inserts a new student into the grid on the classroom information screen and then tries to navigate to the Parent/Emergency Contact screen for the newly inserted student. The user can add students to the system either by clicking on an Add Student button and filling out a form or by entering data in the blank row at the bottom of the student grid. When a new student is added via the form, the changes are saved to the database before the new student is displayed in the refreshed grid. However, when the user adds a new student on the grid itself, he or she must click on the “Save Changes” button to insert the data into the database.

Much like Transition 4, this transition can only be reached if the user follows another additional transition in the FSM that does not cause a failure by itself. The user must first insert a new unsaved row in the student DataGrid and then click on the “View Parent/Emergency Contact button, which causes the failure to occur. An analysis of the source code show that, when the user clicks on the “View Parent/EC Information” button, the system gets the key field “studentID” from the grid based on which row the user currently has selected. In this case where the new row has data but the changes have not been saved to the database, the studentID key field is null because it is generated within the database itself, triggered by the insertion of a new row into the back-end student table. Thus, when the user tries to navigate to the Parent/Emergency Contact screen, the system crashes because it does not know how to handle the null value in the studentID field when trying to retrieve Parent or
Emergency Contact information from the database. Parents (and Emergency Contacts) are stored in tables in the database that are child tables of the student table, and this studentID is a foreign key in both tables (see section 4.1.4), so a null value in this key field is simply not allowed.

This fault was introduced into the system because of the delivered behavior of the student table. The design called for student information to be presented in a table similar to a spreadsheet. A DataGrid object was selected for the implementation because it matched the requirements for displaying and editing student information and it was implemented in the drag-and-drop environment of the GUI design tool with a few properties being set for proper configuration. However, it also created a new transition as a new way to add students to the system. This new transition, as described above, eventually caused a defect when the system tried to use data from this unsaved row in the same way it uses data from the saved rows in the DataGrid.

The solution presented to prevent the failure from Transition 4 above would also prevent this failure. Assuming that a majority of DataGrid objects would be used in a similar manner as how they are used in this system, the GUI design tool should include some sort of delivered edit to make sure that unsaved changes in DataGrid objects are not lost.

Transition 6: (CIS #37, Fault #42) Caused the surprise “Previous edit was gone after clicking ‘Add Parent’ button in dialog; changes in the parent DataGrid reverted to their original values” - Introduced by the developer; no relation to the GUI design tool
For this defect to occur, a user must make changes to information in the Parent data grid, then click on the Add Parent button and successfully add a new parent. The changes originally made in the grid do not get saved and disappear when the Parent data grid is refreshed upon the successful addition of a new parent.

This fault that causes this defect is similar to Transition #4 above. Once again, another additional transition in the FSM must be traversed to get to the state that allows usage of this transition that caused the surprise. The user must first click on the Add New Parent button, and then must successfully save the new parent for the surprise to occur. When the new parent is successfully added to the system, the dialog used to add the parent returns a success value (DialogResult.OK) to the calling function in the “View Parent/EC Information” screen. The DialogResult.OK return value causes a refresh of the parent data set object (dsParents1) so that the new parent will be displayed in the grid on the page. When dsParents1 is refreshed from the database, it pulls in the original values in the fields where the user first made edits because those edits were never stored back in the database. They were only local to the system memory and were overwritten with values from the database by the database connector object’s “Fill” method.

This fault occurs because the transition to add a new parent was not a part of the original design or FSM, but this fault is due to programming errors as opposed to the nature of the GUI design tool. The “Add Parent” button could be clicked on while the user was making changes in the parent data grid, and it would be up to the developer to address this. To prevent this error, the system should either save changes in the data grid automatically or prompt the user to save changes when the “Add
Parent” button is clicked. There should be some sort of mechanism that checks to see if edits have been made and prevents further action until those edits have been committed to the database or cancelled.

Transition 7: (CIS #37, Fault #43) Caused the surprise “Previous edit was gone after clicking “Add Emergency Contact” button in dialog; changes in the emergency contact DataGrid reverted to their original values” - *Introduced by the developer; no relation to the GUI design tool*

The root cause of this surprise was the same as described in Transition 6 above. Transitions 6 and 7 each correspond to distinct faults, but the faults are virtually identical; the fault for Transition 6 corresponds to the Parent section, whereas the fault for Transition 7 corresponds to the Emergency Contact section. In this FSM, the user edits information in the Emergency Contact grid and then clicks on the “Add Emergency Contact” button to add and save a new emergency contact. The same surprise happens where the changes in the Emergency Contact grid are overwritten by the addition of a new emergency contact using the dialog. Lastly, just like Transition 6, the surprise occurs as a result of saving after following another additional transition in the FSM (clicking on the Add New Emergency Contact button).

Transition 8: (CIS #39, Fault #44) Caused the surprise “Previous edit was gone after clicking “Add Assignment” button in dialog; reverted to original value” - *Introduced by the developer; no relation to the GUI design tool*
The root cause of this surprise differs only slightly from the explanation given above for Transition 6, and again the surprise only occurs in combination with another additional transition in the FSM. In this CIS, the user edits assignment information in a grid and then clicks on an “Add New Assignment” button (analogous to editing parent information and then clicking on the “Add Parent” button). The system behaves in the same way: the original changes are lost once the “Add Assignment” (save) button is clicked in the Add Assignment dialog. Just as in Transitions 6 and 7 discussed above, the code refreshes the assignments data grid upon a successful save, and the local-only edits that were never committed to the database are overwritten with the refreshed data from the database.

**Transition 9: (CIS #48, Fault #45) Caused the surprise “‘Changes saved’ message popped up when no changes were made” - Introduced by the GUI design tool**

The surprise in this CIS was very minor with a severity rating of 1 (see Table 3-2), which means it is merely a cosmetic issue. The issue is that the “Changes Saved” message box pops up when the user clicks on the “Save Changes” button even if no changes have been made to the system. The only negative effect is that the user might get confused thinking that he or she made changes when in fact none were made.

This transition was introduced into the system because the GUI design tool did not create any automatic checks when creating the DataGrid object for assignments or the “Save Changes” button. To address this issue the GUI design system should either disable the “Save Changes” button until a change has been made in the data grid or
check to see if any changes have been made when the user clicks on the “Save Changes” button. Investigation into this fault showed that the dataset object used to populate the data grid has a method called “HasChanges” that returns a Boolean value indicating whether or not any changes have been made to the data set. This property could have been used to trigger some sort of message to the user to let him or her know that there are no changes that need to be saved to the database.

**Transition 10: (CIS #49, Fault #45) Caused the surprise “‘Changes saved’ message popped up when no changes were made” - Introduced by the GUI design tool**

This new transition exposed the same root fault as described above for Transition 9. This test and behavior were almost exactly the same except that this transition occurred after selecting an assignment on the Assignment screen, whereas Transition 9 occurred when the user first enters the screen and has not yet selected an assignment.

**Transition 11: (CIS #70, Fault #46) Caused the surprise “Student from previous test case was placed on newly loaded seating chart” - Introduced by the GUI design tool**

The surprise discovered by this transition was that a student from a previous selection (test case) would still be active and placed on a seating chart after loading a new seating chart from disk. The expected behavior was that no name would be
placed on the chart because the user would be required to first select a student after loading a new seating chart.

The underlying fault behind this surprise is that the drop-down list of students at the top of the seating chart screen was not refreshed upon loading a new chart. When a new seating chart is loaded from disk, the code simply sets the bitmap area on the screen to the image saved in the seating chart file but does not do any other processing. Consequently, the previous student selected in the drop-down list was still selected, so when the user clicks on the seating chart the system finds an existing selection and places the student’s name on the seat that was clicked on.

The specific transition that caused this failure was actually the result of multiple transitions that were introduced into the GUI via the GUI design tool. The original design was to be very linear: the user opens a new file and initializes the Assign Seats screen with the seating chart file and current class information, then selects students from the drop-down list and places them in seats, and then saves the newly populated chart. However, the GUI design tool allowed the Load Seating Chart and Save Seating Chart (see Transition 12 below) buttons to be clicked while in the middle of the assign seats process. This caused only the drawing area to be reloaded, and the initial code that runs when the screen is first entered was not called. At this point the user could then follow even more transitions that were added to the CIS, such as clicking on a seat without first selecting a student from the drop-down list (as required the first time through).

This ability to click on other objects while in the middle of an action is a common and complex problem with GUI systems that have multiple paths through the
system. For this specific issue, one potential way the GUI design tool could help would be to have some sort of option where the developer must choose Yes or No upon opening or saving a file using the delivered OpenFileDialog or SaveFileDialog objects. This option (maybe a Boolean property in the object) could be called something like TriggerPageLoadEvents, which, when set to true, would refresh the screen in which the document is being loaded or saved, causing all page load code to be executed automatically. When set to false it would behave as it did in this system where no additional code is called.

**Transition 12:** (CIS #70, Fault #46) Caused the surprise “Student from previous test case was placed on newly saved seating chart” - *Introduced by the GUI design tool*

This failure is a result of the same fault that caused the surprise for Transition 11 above. In this case the user was navigating a different new transition in the CIS that involved saving a seating chart that was created in the system to disk (as opposed to loading a chart from disk). The end result and the fault are the same; this is just another transition that was introduced by the GUI design tool that also introduced Transition 11.

**Transition 13:** (CIS #1, Fault #47) Caused the surprise “User can resize Login/Switch User dialog so that input textboxes are hidden, which could confuse the user” - *Introduced by the GUI design tool*
The root fault of this transition is the default behavior of the login dialog that was created in the development tool. The dialog, which is a Windows.Form object, has a property called "FormBorderStyle" that controls how the border looks and operates. By default it is set to "Sizable", meaning the user can change the size of the dialog manually by dragging on the edges of the dialog with the mouse pointer. This property should have been set to another option that begins with "Fixed" such as "FixedSingle" to prevent the user from resizing the dialog.

The GUI design tool should be changed to not allow resizing as the default setting when creating these types of dialogs in the program. Since the developer needs to explicitly size the dialog window when creating it in the visual development environment, it would follow that the user would not be able to override the size setting without the developer explicitly allowing resizing.

6.3 New Transitions that Did Not Lead to Failures

As listed in Section 5.3, there were 41 FSMs that contained additional transitions found during the development of the implementation tests. These 41 FSMs contained 313 individual transitions, but only 13 of these new transitions (in only 10 of the FSMs) led to faults in the system. The other 300 transitions in the remaining 31 FSMs were benign in that no observable defects or surprises were detected by traversing them. This seems like a very high number, but in fact multiple transitions were often introduced into an FSM by one GUI object simply because the object could be used from multiple states. For example, in CIS #41 (Add New Assignment Group), the user was required to enter an initial assignment for the group, which was not part of the original design. The user could
perform this operation from any point in the original FSM, which meant that 32 additional transitions were created by the addition of one GUI object.

A few examples of benign behavior that was introduced into the system because of additional transitions are given below:

- All popup dialogs in the system contained a Close button (the Windows “X”) in the upper right corner. This was not part of the original design and gave the user another way to cancel out of the dialog, but there were no ill effects because of it.

- When making changes in the Student Information data grid, the user could print the class list before saving the changes. The printout would show the changes that the user entered on the screen. The user could then still save the changes to the database.

- Additional functionality was added to the Manage Assignment Groups screen that was not specified in the design. As implemented, the user must also enter an initial assignment for the group in addition to the group name and weight that were required in the original design. There were no ill effects because of this additional functionality, and it strengthened the integrity of the assignment groups.

- In the Classroom Map screen, the user could click on a number of options such as clearing the drawing area or printing the map before actually placing the object in the map (CIS 60-69).

It is very interesting that these 300 transitions did not lead to any faults, meaning that they were benign differentiations from how the system was designed. These cannot simply be ignored, however, since they were not intended as part of the original design.
Even though no faults were discovered by the tests, it is possible that the user could operate the system in such a way that these unintended transitions could produce a fault that was not tested for. There is also a risk of developing new faults in future versions of the software. Since these transitions were not part of the original design and were not documented, future developers might overlook the transitions and introduce new code or transitions that would cause a fault in conjunction with these transitions.

Consider the example benign transition above that details how all popup dialogs contained an Exit button that was not part of the original design. Imagine that a future change to the system requires that all of the fields on one of these popups must be filled in completely before clicking the OK button, and should not allow the user to cancel the transaction once it is initiated. Perhaps it is part of a multi-step process that, once started, must be followed through to completion. This undocumented Exit button could compromise the integrity of the system if the user were allowed to close the popup without providing the required information and clicking OK. It is possible that claimed memory would be corrupted and cause a memory leak, or maybe the unfinished process would cause errors for the user the next time he or she attempted to complete the same process.
Chapter 7 Conclusions

The conclusions below are organized into subsections based on topics related to the results presented and discussed in Chapters 5 and 6, as well as some more general topics about the empirical study as described in detail in Chapter 3.

7.1 New Transitions

The primary question about the results presented in Chapter 5 is: Where did the new transitions in the FSMs come from? These were not part of the system as designed, so by what means were they introduced to the system? Of specific interest are the 13 transitions and their 10 underlying faults introduced in 10 FSMs that were only detected by implementation tests, meaning that they were not introduced as part of design problems (the other 37 faults that were found by the design tests). These faults and behaviors were introduced having nothing to do with the original design of the system.

A detailed analysis of these 10 faults and the 13 transitions that exposed them was given in Section 6.2. Based on this analysis, which traced each fault back to its origin in the source code and tried to determine how it got there, it was determined that 9 of the failures were truly new transitions introduced by the GUI design tool, and that the remaining 4 failures were simply the result of programming errors introduced during the design phase.

The 9 fault-inducing transitions introduced by the GUI could have been prevented in one way or another. Each transition, and how it could have been prevented, was discussed in detail in Section 6.2. Below is a brief summary of how these additional
transitions could have been prevented by the GUI design tool. Discussions of the 4 failures resulting from programming errors are omitted.

The detailed analysis in Section 6.2 talked about how some of the 13 transitions were only available because of other additional transitions that had been introduced into the FSM. Those pairings dealt with how a failure was caused, whereas the pairings discussed in this section (Transitions 1 and 2, 9 and 10, and 11 and 12) deal with transitions and failures that are traced back to the same underlying faults. In other words, either transition in these pairs manifests the failure.

**Transitions 1 and 2:** These transitions were caused because the user could use objects on the main window while using an open modal dialog. The GUI design tool should have automatically locked the other controls upon opening a modal dialog.

**Transition 4:** This transition was caused because the user was allowed to act on the database while having unsaved changes in the student DataGrid object. The GUI design tool could prevent this behavior by automatically displaying some sort of warning message to the user if he or she takes any action outside of the DataGrid after making unsaved changes to the DataGrid. Saving or cancelling the changes would reset the DataGrid to a stable state and the user could then proceed with other actions.
Transition 5: Allowing the user to navigate to a different screen while having unsaved changes in the student DataGrid caused this failure. The solution presented to prevent the failure from Transition 4 above would also prevent this failure.

Transitions 9 and 10: These transitions occurred because the system allowed the user to click on the “Save Changes” button before making any changes in a DataGrid object. The GUI design tool could have prevented this by using a property of the DataGrid object called “HasChanges” to trigger a message to the user to let him or her know that there no changes need to be saved to the database.

Transitions 11 and 12: These two transitions were introduced because the user was able to click on the Load and Save objects while in the middle of assigning seats in a class. They could have been prevented if the GUI design had an option where the developer must choose Yes or No upon opening or saving a file using the delivered OpenFileDialog or SaveFileDialog objects. This option, when set to true, would refresh the screen in which the document is being loaded or saved, causing all page load code to be executed automatically.

Transition 13: This last transition was caused because the default settings of the GUI design tool allowed modal dialogs to be resized. The GUI design tool should have set the default behavior for all modal dialogs to not allow resizing.
A quick review of the above summaries reveals an overall theme about how the GUI design tool could have been changed to prevent some of these failures. Any GUI design tool must draw a line for each feature it implements about where to end default, “out-of-the-box” behavior and let developer control begin. It is impossible to make these decisions for all objects and all developers because some objects require lots of input from the developer, and some developers prefer more control about how objects are implemented. However, developers who desire such complete control can, and sometimes must, learn and embrace new technology and features made available in their development tools. There are many obvious benefits to standardizing how objects are implemented such as utilization of existing language and/or interoperability standards, better compatibility with debugging and testing tools, and the advantage gained in software maintenance by using shared and delivered code. Trying to solve all problems with one technology, tool, or way of implementing things is bound to fail because real-world applications are too complex.

The GUI design tool must make assumptions about what is considered "normal" behavior, appearance, or properties for a GUI application, which may be in direct conflict with the intended GUI design. As a result, as is seen with these results, the GUI design tool can introduce defects or surprises. Thus, one of the GUI design tool’s biggest strengths is also one of its biggest problems. By utilizing so many generated objects, it becomes very easy to misuse the objects (e.g., the DataGrid object in Transition 9 [Section 6.2]). It also requires a considerable amount of time and effort for a developer to become an expert in the GUI design tool and understand all the objects and methods at his or her disposal.
Therein lays one advantage of building objects from scratch: the developer has a much more thorough understanding of the framework behind the objects and what can be done and what should be done. As with many decisions in software development, developers must weigh the risks and rewards of different approaches (languages, development tools, patterns, etc.) and decide which approach is best for the task at hand. The GUI design tool gives the developer the option to drill into the generated code for the GUI objects, so ultimately he or she has the level of control that he or she chooses. However, the developer must first fully comprehend the default settings and behavior of the generated objects before he or she knows how far to drill into the code to achieve the specified behavior.

Thus, a recommendation for improving the tool can now be given: the GUI design tool should ask the developer more questions about how an object is intended to be used when the developer adds a GUI object to the system and the design tool generates the code behind the object. Even a few yes/no questions that correspond to Boolean properties could make a big difference in how the object’s behavior meets the developer’s expectations. Perhaps different modes could be available for the developer to choose from when starting a new application. If the developer is new to the GUI design tool or would like more up-front control of objects as they are added to the system, then he or she could choose a “Beginner” or “Wizard” mode (“wizard” referring to the common step-by-step based installation wizards that are common to GUI systems). A more experienced developer could choose the streamlined (existing) mode where assumptions about the GUI objects are made and understood.
7.2 Programming and Unit Testing

The four remaining transitions (3, 6, 7, and 8) and the corresponding failures should also be addressed, albeit in less detail than the nine discussed in section 7.1. These four failures are the result of faults that had nothing to do with the GUI design tool. In all four cases the developer made simple programming mistakes and it is the developer’s responsibility to address each of them. These faults are relatively easy to fix and are simply the result of human error and developer inexperience.

The risks and benefits of having all programming and testing performed by one person (the author) were documented in section 3.5. However, some additional observations should be noted about this. The C# development environment is very complex and takes a long time to understand and master. The developer was inexperienced with this framework and likely some of the programming errors can be attributed to this inexperience. It was also discovered too late how important it is to document unit testing. Some of the faults should have been detected at an earlier stage, and maybe even were, but unit testing was not documented, so there is nothing to review to see if issues were caught and/or overlooked. It is even possible that some issues were caught but never properly addressed in the development phase.

7.3 Test Methodology

As discussed in Sections 6.1 and 6.2 (and in earlier papers by White et. al. [1], [2]), both the design and implementation tests were found to be worth executing. However, given the large number of tests, a full execution of both design and implementation tests may not always be possible because of time and/or resource constraints. In these
scenarios, one must ask if any tests can be skipped and perhaps prioritize the tests in some manner such that highest priority items are tested first and lower priority items get tested as time permits.

Certain patterns emerged as testing was conducted such that the tester could predict with some degree of accuracy which test cases were likely to uncover a fault, or perhaps more importantly, which ones were very unlikely to uncover a fault. If time for testing is extremely scarce, certain functions that had to be tested multiple times in slightly different ways could probably be skipped without affecting the overall quality of the testing effort. After running a lot of tests the tester will have developed a certain “tester’s intuition” about which tests are likely to cause failures and which can safely be skipped. At present there is no known way to quantify this to algorithmically determine which tests can be skipped, but a lot of work has been done in this area and one of the best suggestions for reducing the time and effort to test is to give a higher priority to tests that have discovered faults in previous testing efforts. It would be an interesting topic for future research to see if it would be possible to flag areas as high risk or low risk before any testing has been performed and any faults have been discovered, perhaps based on some sort of measure of complexity of each CIS.

This type of tester-based “test case pruning” could be helpful in reducing the number of tests for different methodologies. One example is the GUI testing framework proposed by Memon in [9]. The need for automated testing tools has been researched extensively because automated testing tools can greatly decrease the time to test a system, especially when the tests need to be run multiple times. For example, establishing a set of automated tests can greatly improve regression testing because the changed code can be
run through all the tests for the previous code, and the results of the two test runs can be compared against each other. However, in [9], Memon describes how various efforts to design automated tools have been limited in that they are not applicable across all aspects of the GUI testing problem.

One area that is particularly important to the GUI testing framework, and also much disputed, is the notion of the *coverage* of the test cases. The level of coverage logically affects how well faults will be detected, as shown in studies like those described by Vouk [23] and Malaiya, et. al. [24], who have shown that “the fault detection rate with respect to coverage is proportional to the coverage.” [23] Memon goes on to say in [9] that “potentially problematic user interactions” are of great importance compared to traditional testing methods that aim for complete code coverage. At present it seems that humans are better than any systems or algorithms at identifying “potentially problematic user interactions”, so it is possible for an experienced tester or user to identify these interactions so that test plans can be focused more heavily on them.

Here an important point is made: there are different types of coverage, and attaining one type of coverage (like code coverage) is not the same as attaining coverage of a different sort, like event coverage. Attempting to develop a set of tests to completely cover the code has a number of drawbacks, a few of which are:

- Even with 100% code coverage, there is no guarantee that all faults will be discovered because of how the code interacts, often in unpredictable ways.

- Complete code coverage requires a large number of tests, which requires more time, energy, and cost and often do not result in more faults or details than can be attained with other methods.
The CIS testing method is an event-driven approach to coverage. In this type of approach, the test designer focuses on events within the system that are triggered by the user or in code. Full coverage means that all events triggered by users or code are tested. The nature of events is that something within the system is being created, changed, or acted upon in some way that is important to the user, and these are exactly the type of “potentially problematic user interactions” that the CIS testing method and others like it aim to investigate with the highest priority.

In [25] Leon, Podgurski, and White define an approach to testing called observation-based testing that filters a large amount of potential test data to obtain a small subset to use on the system. Since the source code for this SUT is available and can be modified as needed, the observation-based testing method could be applied to the tests run against this SUT and the results could be profiled to find a subset of tests. The subset could be run and the results could be compared against the test results from the original test suite to see how well the subset performs. It would be interesting to see how this generated subset would compare with a subset of tests that was created manually using “tester’s intuition” to target areas of “potentially problematic user interactions”, such as the classroom map section of the SUT. This area of the system could be classified as “potentially problematic” because of the high number of GUI objects available for the user to use and the fact that actions affect the GUI considerably because of the drawing area on that screen. It would also be interesting to see if there is any sort of correlation between areas designated as “problematic” by someone using “tester’s intuition” and parts of the GUI shown to be more complex based on the number of GUI objects, number of CIS, or some other measure of GUI complexity.
7.4 Memory Tools

One surprising outcome of this study is that memory tools were not as helpful in identifying faults as in previous studies that utilized the CIS testing method. One possible explanation for this is that the computing environment that was used may have been more robust in terms of memory utilization than in previous studies. The C# .NET framework uses newer and better memory management tools such as automatic garbage collection, and the large number of delivered objects in the framework are better optimized to use the underlying resources than in earlier languages or operating systems. Garbage collection can greatly improve memory utilization over explicit memory management, which causes fewer memory leaks, object corruption, or other memory problems [26]. An early paper on C# by Lutz and Laplante [27] found that, while C# does not compare well with the C programming language in terms of efficiently utilizing physical memory for “hard real-time systems”, it does “shield developers from complex memory management logic” because of its “ability to interact with operating system APIs”. A more recent paper by Jackson and Frazer [28] asserts that the Common Language Runtime (CLR) (the virtual machine that converts Common Intermediate Language (CIL) code into native code) reduces application errors and improves overall software quality by using automatic memory management.

Another possible explanation is that the SUT for this thesis simply was not as complex as those tested in previous studies. Using either the number of GUI objects or the number of CIS to estimate the complexity of the GUI system shows that the SUT for this thesis (141 GUI objects for 73 CIS) was significantly less complex than some other systems that had hundreds of GUI objects with well over 100 (and in one case, 200) CIS.
It follows that increased complexity of a GUI system would correlate with increased complexity in the logic of the system and how memory is used. Managing the GUI objects and their corresponding memory becomes more complicated, and it is not surprising that those earlier, more complex systems would have more faults related to memory.

On a related note, all development and testing were performed on a machine running Windows XP, which is part of the NT family of Windows operating systems and is known to be more reliable than previous versions of Windows [29][31]. Previous studies on the CIS testing method that found more faults using memory tools were testing software running on Windows 98 or ME, which are part of the same OS family designed for home users as opposed to business users. Kanoun et. al. show that dependability benchmarking of the Windows NT OS family indicate that Windows NT4, 2000, and XP are equivalent [30].

It is also possible that memory tools were not as helpful because having access to the source code gives more insight about the internal interactions and dependencies in the underlying code. The goal of using memory tools is to detect changes to memory in the system, but it does not necessarily mean that the memory was corrupted or that there is in fact an underlying fault behind it. The earlier studies using the CIS method used the tools this way mainly to figure out why some areas of memory that had nothing to do with the GUI system would suddenly change, indicating some sort of dependency, good or bad.

The fault analysis in Section 6.2 shows that the most severe faults (Transitions 4 and 5) were introduced by the GUI design tool and had nothing to do with memory utilization. Although it is possible that other faults related to memory may exist in the
system, they must be considered of lesser importance because the behavior of the system was thoroughly investigated. Changes in memory that do not affect the responsibilities of the application can thus be ignored for this study. Of more concern were behaviors that were introduced or modified that affected the user experience.

A total of 11 surprises were discovered by the memory tools and none of them even registered on the severity scale because the lowest value is 1 (Cosmetic) and these surprises, unseen by the user, had absolutely no effect on the system. Below are a few examples of these surprises, included here to illustrate why they can safely be ignored in this thesis. The detailed test results in Appendix C contain notes about all 11 surprises that were detected by the tools.

- Using Memory Dump: Block 0C010510 contains username "rosie" and the corresponding password, even though the tester was logged in as the user “bjj3”. It must have been set in a previous test and was never reclaimed or overwritten.
- Using GlowCode: System.Windows.Forms.Timer::OnTick is continuously being called for the timer on the main page. It was surprising that this was being called even when the user was on different pages in the system.

7.5 Future Research

The CIS GUI testing method as used for this thesis creates a suite of tests that would be an excellent foundation for an automated test suite. The design and implementation tests, which are all run manually, could be captured so that regression testing could be done by replaying the original set of design and implementation tests. The expected outcomes from the initial manual run of the tests could be used to compare
the automated test results to verify the accuracy of the tests, and future design and implementation tests for code changes and enhancements could simply be added to the original set of tests to quickly and completely regression test the entire GUI system.

Memon, Banerjee, and Nagarajan [4] developed a method he calls “GUI Ripping”, where a program they developed automatically traverses a GUI to extract all the GUI objects, something that was done manually as part of this study. This type of process could decrease the overall time to design both the design and implementation tests and could potentially be helpful in discovering additional transitions for implementation testing.

The way memory tools were used to collect data during the testing phase was completely manual and quite cumbersome. Although the tools did not prove to be very helpful for this particular study, it would be nice if they could be integrated with the system or with automated testing tools so that different areas of memory (blocks, swap space, messages between objects, etc.) could be monitored and analyzed with less manual intervention.

It would be interesting to compare results of this thesis with a future study that uses the CIS testing method to test a commercial-grade software system where a drag-and-drop GUI design tool was used, but where the testers do not have access to the underlying source code. In this sort of black box approach, the test designers must make assumptions or guesses about what to test in the system corresponding to an underlying dependency, whereas in this thesis such dependencies were known and could be used both in designing tests and investigating failures. Further, memory tools, especially if integrated with the process as described in the preceding paragraph, could then be used to verify whether or
not the predicted dependencies exist. It seems likely that more tests would be required in this case because without knowing how the objects interact within the GUI, more tests would be required to test these possible dependencies.

Lastly, further testing on the type of GUI design tool used in this thesis should be done to investigate how fault-prone their generated code is in different scenarios. For example, if parts of the GUI were developed at different points in time, how would the overall system be affected? Would more or fewer transitions be introduced? What if the system were implemented by a team of developers, each of whom worked on different areas within the GUI? Or what if some of the suggestions given in Section 6.2 to reduce the potential for the GUI design tool to introduce new transitions were actually implemented? Would fewer transitions result, or might new and different ones be introduced? As the software development industry evolves, and as more sophisticated users interact with faster and more complex systems in new and revolutionary ways, so too must the tools and methods used to test these systems. They must be able to ensure that the systems can be relied upon in mission-critical applications, and do so in such a way as to be cost effective in an industry where time, money, and resources are in higher and higher demand.
Appendix A. Implementation Details

Below is a complete numbered list of responsibilities for the student information system.

1. Log user into system
2. Cancel login - exit program
3. Save current user as default
4. Close program
5. Minimize program window to system tray
6. Expand to/retract from full-size window
7. Add new user
8. Change user
9. Open file
10. Change file type filter
11. Save classroom map
12. Attempt to save non classroom map file
13. Print current screen
14. Close program via menu
15. Undo last action
16. Cut current selection
17. Copy current selection
18. Paste current clipboard item
19. Go to Main screen via menu
20. Go to Classroom Information screen via menu
21. Go to Lesson Planner screen via menu
22. Go to Classroom Map screen via menu
23. Display program information
24. View class information
25. Set current class as default class
26. View Lesson Planner for a specific day
27. Go to Classroom Map screen
28. Go to Assign Seats screen
29. Add new student to class list
30. Import student(s) from a file
31. Edit student information
32. Save changes to student information
33. Print class list
34. View parent/emergency contact info
35. Add new parent
36. Add new emergency contact
37. Edit parent/emergency contact info
38. View assignments
39. Edit assignments
40. Add new assignment
41. Add new assignment group
42. Delete an assignment group
43. Edit an assignment group
44. Edit an assignment group name
45. Add assignment to group
46. Delete assignment from group
47. Cancel deleting an assignment group
48. View grades
49. Edit grades
50. View Lesson Planner for a specific day on Lesson Planner screen
51. Print a daily planner
52. Add a new lesson
53. Create a new classroom drawing
54. Define dimensions of classroom
55. Cancel defining dimensions
56. Clear drawing
57. Save drawing
58. Save drawing as seating chart
59. Print drawing
60. Add a desk to drawing
61. Add a chair to drawing
62. Add a wall to drawing
63. Add a table to drawing
64. Add a board to drawing
65. Add a door to drawing
66. Add a window to drawing
67. Add a book case to drawing
68. Add a coat rack to drawing
69. Add text to drawing
70. Assign a student to a seat on a class map
71. Print seating chart
72. Load a saved seating chart
73. Save a seating chart
Appendix B. Test Cases for Fully Testing the Open File FSM

Below are the tests required to fully test the Open File FSM as shown in Figure 1-3:

1. Click File Menu → Click “Open File” → Select File → Click Cancel

2. Click File Menu → Click “Open File” → Select File → Click Close Dialog “X” button

3. Click File Menu → Click “Open File” → Select File → Click Open → File Opened

4. Click File Menu → Click “Open File” → Type File Name → Click Cancel

5. Click File Menu → Click “Open File” → Type File Name → Click Close Dialog “X” button

6. Click File Menu → Click “Open File” → Type File Name → Click Open → File Opened

7. Click File Menu → Click “Open File” → Select File → Click Open → File Not Found → Click Open File → Click Select File → Click Cancel

8. Click File Menu → Click “Open File” → Select File → Click Open → File Not Found → Click Open File → Click Select File → Click Close Dialog “X” button

9. Click File Menu → Click “Open File” → Select File → Click Open → File Not Found → Click Open File → Click Select File → Click Open → File Opened

10. Click File Menu → Click “Open File” → Select File → Click Open → File Not Found → Click Open File → Type File Name → Click Cancel

11. Click File Menu → Click “Open File” → Select File → Click Open → File Not Found → Click Open File → Type File Name → Click Close Dialog “X” button
12. Click File Menu -> Click “Open File” -> Select File -> Click Open -> File Not Found -> Click Open File -> Type File Name -> Click Open -> File Opened

13. Click File Menu -> Click “Open File” -> Type File Name -> Click Open -> File Not Found -> Click Open File -> Click Select File -> Click Cancel

14. Click File Menu -> Click “Open File” -> Type File Name -> Click Open -> File Not Found -> Click Open File -> Click Select File -> Click Close Dialog “X” button

15. Click File Menu -> Click “Open File” -> Type File Name -> Click Open -> File Not Found -> Click Open File -> Click Select File -> Click Open -> File Opened

16. Click File Menu -> Click “Open File” -> Type File Name -> Click Open -> File Not Found -> Click Open File -> Type File Name -> Click Cancel

17. Click File Menu -> Click “Open File” -> Type File Name -> Click Open -> File Not Found -> Click Open File -> Type File Name -> Click Close Dialog “X” button

18. Click File Menu -> Click “Open File” -> Type File Name -> Click Open -> File Not Found -> Click Open File -> Type File Name -> Click Open -> File Opened
Appendix C. Detailed Results

Below is a reproduction of the spreadsheet used to document and summarize all defects and surprises found in the system under test. Each defect or surprise was assigned a unique number and severity and then notes were added about the test results and investigations into the underlying faults. Failures that corresponded to a common fault were grouped together and assigned a unique fault number.

<table>
<thead>
<tr>
<th>Des/Imp #</th>
<th>Test CIS #</th>
<th>Unique Failure #</th>
<th>Defect or Surprise</th>
<th>Severity</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Des d0</td>
<td>1</td>
<td>1</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: Needed to click &quot;Log in&quot; button twice to have an effect. Upon clicking a second time I was logged into the program</td>
</tr>
<tr>
<td>Des d8</td>
<td>1</td>
<td>2</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: same as test 6</td>
</tr>
<tr>
<td>Des d12</td>
<td>1</td>
<td>3</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: same as test 6</td>
</tr>
<tr>
<td>Des d16</td>
<td>1</td>
<td>4</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: same as test 6</td>
</tr>
<tr>
<td>Des d18</td>
<td>1</td>
<td>5</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: same as test 6</td>
</tr>
<tr>
<td>Des d21</td>
<td>1</td>
<td>6</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: (related to test 6 failure) I accidentally hit the login button first, but then hit cancel (since you have to click it twice to login). When I went to rerun this test I was logged in directly as the default user. The default information was saved on the first click.</td>
</tr>
<tr>
<td>Des d22</td>
<td>1</td>
<td>7</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: same as test 6</td>
</tr>
<tr>
<td>Des d28</td>
<td>1</td>
<td>8</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: same as test 6</td>
</tr>
<tr>
<td>Des d52</td>
<td>1</td>
<td>9</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: same as test 6</td>
</tr>
<tr>
<td>Des d58</td>
<td>8</td>
<td>10</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: same as test 6</td>
</tr>
<tr>
<td>Des d62</td>
<td>8</td>
<td>11</td>
<td>Defect</td>
<td>2</td>
<td>FAILURE: same as test 6</td>
</tr>
<tr>
<td>Des d57</td>
<td>7</td>
<td>12</td>
<td>Surprise</td>
<td>3</td>
<td>SURPRISE: System added memory blocks 09600000 thru 09600FF0, 0BFCl000 thru 0C0BFFFO, 7FFD8000 thru 7FFD8FF0. 0BFCl000 contained the string &quot;System.Data.ProviderBase.DbConnectionPool&quot;, which indicates that the after state still had an open connection to the database</td>
</tr>
<tr>
<td>Des d40</td>
<td>7</td>
<td>13</td>
<td>Surprise</td>
<td>3</td>
<td>SURPRISE: New blocks claimed</td>
</tr>
</tbody>
</table>

Notes:

Severity scale:
1 = Cosmetic
2 = Workaround available
3 = Design a simple solution
4 = Major rework/program unstable
NA = does not concern user
at 09581000 and 7FFD7000
0958 1000 contained the string "System.Data.ProviderBase.DbConnectionPool", which indicates that the after state still had an open connection to the database ("sometimes" behavior - see test #1).

Des d42 7 14 Defect 3

FAILURE: "shoogie" was added to the database a second time (with the different password)

FAULT: 2 faults: 1) DB did not have a unique constraint, and 2) the program failed to check for an existing user with the input username.

#4 This fault was introduced during the database design phase. The keys and constraints for this table should have been set up to prevent this defect.

#5 This fault was introduced during the coding phase and was not caught during unit testing

Des d48 7 15 Defect

FAILURE: (same as test 42) "jak" was added to the database a second time (with the different password)

Des d53 7 16 Defect

FAILURE: (same as test 42) "rosie" was added to the database a second time (with the different password)

Imp i28 7 17 Defect

FAILURE: same as test d42

Imp i33 7 18 Defect

FAILURE: same as test d42

Imp i37 7 19 Defect

FAILURE: same as test d42

Imp i47 7 20 Defect

FAILURE: same as test d42

Imp i48 7 21 Defect

FAILURE: same as test d42

Imp i57 7 22 Defect

FAILURE: same as design test #42. jjak user was saved. Should not have been saved.

Imp i58 7 23 Defect

FAILURE: same as design test #42. jjak user was saved. Should not have been saved.

Imp i65 7 24 Defect

FAILURE: same as design test #42. rosie user was saved. Should not have been saved.

Imp i66 7 25 Defect

FAILURE: same as design test #42. rosie user was saved. Should not have been saved.

Imp i69 7 26 Defect

FAILURE: same as design test #42. Both "bella" users were added. The 2nd to be saved should have displayed an error.

Des d51 7 27 Defect

4

FAILURE: At program launch, received the error "An exception of type 'System.Data.OleDb.OleDbException' has occurred in TA4.exe." There was an error connecting to the db, but it was fixed the 2nd time I tried to run the program. This exception should have been handled properly.

This defect was introduced during the coding phase. The GUI designer created the database connection using a drag and drop feature, so it was up to the programmer to code any extra exception handling around the connection. I should have used a "try/catch" block around the "open" command to catch these types of exceptions and handle them properly.

4, 5

6
<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d17</td>
<td>29</td>
<td>Defect</td>
<td>FAILURE: When I first tried to open the program I got a DB error (NEED TO ATTACH SCREENSHOT) Maybe same as defect #5?</td>
</tr>
<tr>
<td>d01</td>
<td>8</td>
<td>Surprise</td>
<td>SURPRISE: Mem block 0BD70510 contained strings &quot;Student...to class...start...of test&quot;. Not sure where these came from.</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>Surprise</td>
<td>This is an unexplained defect, but it seems like it might have to do with the memory tool reading some area of shared memory that was shared with other open programs. Perhaps the memory blocks were in use by some or all of the other open programs (like Excel) during testing.</td>
</tr>
<tr>
<td>d80</td>
<td>16</td>
<td>Surprise</td>
<td>(similar to defect #6) SURPRISE: Block 0C010510 contains username &quot;rosie&quot; and password &quot;rosie2&quot;. Why? Logged in as bjj3.</td>
</tr>
<tr>
<td>d85</td>
<td>16</td>
<td>Surprise</td>
<td>(similar to defect #6) SURPRISE: The string &quot;Block 7FFDF200 in the before file showed&quot; was found in block 0CD10510 of the after file. Maybe block was in use by Excel, let go, and then reclaimed by TA4?</td>
</tr>
<tr>
<td>d88</td>
<td>16</td>
<td>Surprise</td>
<td>(similar to defect #6) SURPRISE: Block 0C010520 in after file contained the text &quot;g displayed - click&quot; - perhaps this is from Excel: the phrase &quot;[Error msg displayed - click OK]&quot; occurs frequently in my test cases. Same reason as test 85 above?</td>
</tr>
<tr>
<td>d90</td>
<td>19</td>
<td>Surprise</td>
<td>(similar to defect #6) SURPRISE: Laptop name (JAKUB EN-LAPTOP) appears in block 0820EF10 in after file. Windows function calls appear further down.</td>
</tr>
<tr>
<td>d113</td>
<td>29</td>
<td>Surprise</td>
<td>(similar to defect #6) SURPRISE: Block 0C020510 contained the string &quot;at time spent in WmPaint ... D Most time spent in Pai&quot;.</td>
</tr>
<tr>
<td>d121</td>
<td>29</td>
<td>Surprise</td>
<td>(similar to defect #6) SURPRISE: Excel text showed up in the before file in block 09850510: &quot;SURPRISE: Block 7FFDF200 in the before file showed&quot;.</td>
</tr>
<tr>
<td>d130</td>
<td>30</td>
<td>Surprise</td>
<td>(similar to defect #6) SURPRISE: Strange character array appears in before file at block 7CBD7000 Strings &quot;Application Data&quot;, &quot;Width&quot;, &quot;Height&quot;, &quot;LayoutPos&quot;, &quot;BorderColor&quot;, etc. appear in before file starting at block 7CBD7550</td>
</tr>
<tr>
<td>d65</td>
<td>9</td>
<td>Defect</td>
<td>FAILURE: image did not show up until I clicked on it (fault - pictureBox.refresh() missing?) Programming error - image box refresh not called when loaded. This defect occurs even when the program is first opened. The image box component gets</td>
</tr>
</tbody>
</table>

105
created and initialized at program startup, so since the object is already in memory while the program is running it should be refreshed when the user enters this screen of the program.

<table>
<thead>
<tr>
<th>Des</th>
<th>d78</th>
<th>13</th>
<th>38</th>
<th>Defect</th>
<th>FAILURE: Nothing printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Des</td>
<td>d251</td>
<td>51</td>
<td>39</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Des</td>
<td>d291</td>
<td>59</td>
<td>40</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Des</td>
<td>d325</td>
<td>71</td>
<td>41</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i85</td>
<td>31</td>
<td>42</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i158</td>
<td>60</td>
<td>43</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i162</td>
<td>60</td>
<td>44</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i166</td>
<td>60</td>
<td>45</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i170</td>
<td>61</td>
<td>46</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i174</td>
<td>61</td>
<td>47</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i178</td>
<td>61</td>
<td>48</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i182</td>
<td>63</td>
<td>49</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i186</td>
<td>63</td>
<td>50</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i190</td>
<td>63</td>
<td>51</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i194</td>
<td>64</td>
<td>52</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i198</td>
<td>64</td>
<td>53</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i202</td>
<td>64</td>
<td>54</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i206</td>
<td>65</td>
<td>55</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i210</td>
<td>65</td>
<td>56</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i214</td>
<td>65</td>
<td>57</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i218</td>
<td>66</td>
<td>58</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i222</td>
<td>66</td>
<td>59</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i226</td>
<td>66</td>
<td>60</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i230</td>
<td>67</td>
<td>61</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i234</td>
<td>67</td>
<td>62</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i238</td>
<td>67</td>
<td>63</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i242</td>
<td>68</td>
<td>64</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i246</td>
<td>68</td>
<td>65</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i250</td>
<td>68</td>
<td>66</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i254</td>
<td>69</td>
<td>67</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i258</td>
<td>69</td>
<td>68</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i262</td>
<td>69</td>
<td>69</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i266</td>
<td>69</td>
<td>70</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i270</td>
<td>69</td>
<td>71</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i274</td>
<td>69</td>
<td>72</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i278</td>
<td>69</td>
<td>73</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i282</td>
<td>69</td>
<td>74</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i286</td>
<td>69</td>
<td>75</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i290</td>
<td>69</td>
<td>76</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i294</td>
<td>69</td>
<td>77</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i298</td>
<td>69</td>
<td>78</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i302</td>
<td>69</td>
<td>79</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i306</td>
<td>69</td>
<td>80</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i310</td>
<td>69</td>
<td>81</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i314</td>
<td>69</td>
<td>82</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i318</td>
<td>70</td>
<td>83</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i316</td>
<td>70</td>
<td>84</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i321</td>
<td>70</td>
<td>85</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
<tr>
<td>Imp</td>
<td>i322</td>
<td>70</td>
<td>86</td>
<td>Defect</td>
<td>FAILURE: Nothing printed</td>
</tr>
</tbody>
</table>

Programming error - print document was never set. All of these print-related defects can be attributed to the same underlying fault: I incorrectly programmed the print feature by failing to set the "print Document" object to what is supposed to be printed from the screen.
<table>
<thead>
<tr>
<th>Imp</th>
<th>i327</th>
<th>70</th>
<th>87</th>
<th>Defect</th>
<th>FAILURE: Nothing was printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Des</td>
<td>d79</td>
<td>13</td>
<td>89</td>
<td>Surprise</td>
<td>2 SURPRISE: only 10 characters of typing were undone. I typed &quot;let's type in here&quot; in an empty cell, and when I clicked undo the cell was updated to &quot;let's tp&quot;</td>
</tr>
<tr>
<td>Des</td>
<td>d94</td>
<td>23</td>
<td>90</td>
<td>Surprise</td>
<td>1 FAILURE: Nothing happened</td>
</tr>
<tr>
<td>Des</td>
<td>d96</td>
<td>25</td>
<td>91</td>
<td>Defect</td>
<td>3 FAILURE: Tested as the user &quot;shoogie&quot;. If no class is in the select box, there is no way to add a new class. This functionality was never implemented.</td>
</tr>
<tr>
<td>Des</td>
<td>d16</td>
<td>28</td>
<td>92</td>
<td>Defect</td>
<td>3 FAILURE: The student was saved with a Byte Array[] in the Photo field</td>
</tr>
<tr>
<td>Des</td>
<td>d14</td>
<td>29</td>
<td>93</td>
<td>Defect</td>
<td>3 FAILURE: The student was saved with a Byte Array[] in the Photo field</td>
</tr>
<tr>
<td>Des</td>
<td>d12</td>
<td>29</td>
<td>94</td>
<td>Defect</td>
<td>3 FAILURE: The student was saved with a Byte Array[] in the Photo field</td>
</tr>
<tr>
<td>Des</td>
<td>d12</td>
<td>29</td>
<td>95</td>
<td>Surprise</td>
<td>3 SURPRISE: I noticed that System.Windows.Forms.Timer::OnTick is continuously being called. This must be the timer on the main page.</td>
</tr>
<tr>
<td>Des</td>
<td>d16</td>
<td>29</td>
<td>96</td>
<td>Defect</td>
<td>3 FAILURE: The students from test 131 were imported into the database again. The OpenFileDialog is reused and was never reset after previous uses. Once the filename had been set from a previous use it would persist in memory because the object was never destroyed or reset. The filename would continue to be used over and over until it was overwritten.</td>
</tr>
<tr>
<td>Des</td>
<td>d18</td>
<td>30</td>
<td>97</td>
<td>Defect</td>
<td>3 FAILURE: The students from test 137 were imported into the database again. (Same issue as test 132)</td>
</tr>
<tr>
<td>Des</td>
<td>d18</td>
<td>30</td>
<td>98</td>
<td>Defect</td>
<td>4 FAILURE: Unhandled exception has occurred - Index was outside the bounds of the array, PROGRAM CRASH</td>
</tr>
<tr>
<td>Des</td>
<td>d18</td>
<td>30</td>
<td>99</td>
<td>Defect</td>
<td>4 FAILURE: Unhandled exception has occurred - Index was outside the bounds of the array, PROGRAM CRASH</td>
</tr>
<tr>
<td>Des</td>
<td>d18</td>
<td>30</td>
<td>100</td>
<td>Defect</td>
<td>2 FAILURE: No warning message was given. Both rows were saved in the database</td>
</tr>
</tbody>
</table>

107
<table>
<thead>
<tr>
<th>Defect</th>
<th>Severity</th>
<th>Issue Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#17</td>
<td></td>
<td>This fault was introduced during the database design phase. The keys and constraints for this table should have been set up to prevent this defect.</td>
</tr>
<tr>
<td>#18</td>
<td></td>
<td>The student ID was being incremented for each student added, so the one constraint on the student table was not being violated. There should have been code to check if a student with the same name already exists so that the user could have the option of changing the name or keeping it if the names are supposed to be the same.</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>FAILURE: No warning message was given. Both rows were saved in the database.</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>FAILURE: The first student is always selected by default, so I clicked on the blank row at the bottom of the grid instead. Clicking on the parent/EC info button took me to the Parent/EC info screen with null rows showing.</td>
</tr>
<tr>
<td>Unexplained</td>
<td></td>
<td>- Programming error - the exception handling is incorrectly coded. It checks if the index of the current row is less than zero, which is not possible. The other code worked as planned but it just didn't bring any data back because of the index error above.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FAILURE: Program crashed - received a error (see screenshot d154.gif). Probably because I opened the program and came back to test it over half an hour later.</td>
</tr>
<tr>
<td>Unexplained</td>
<td></td>
<td>- Unexplained - my guess is that the connection to the database probably timed out.</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>FAILURE: Just noticed that the label on this screen shows &quot;Benny's First Grade&quot;. Went to class info screen and back into this form and it repeated a third time.</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>- FAULT - this label is probably not being initialized properly. It is initially set at program launch and gets added to during the form's &quot;_Enter&quot; event. It should be overwritten during the &quot;_Enter&quot; event rather than appended to.</td>
</tr>
<tr>
<td>Surprize</td>
<td></td>
<td>#2: same as design test #170</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>FAILURE: I typed &quot;max&quot; in the score field and an exception was thrown. This is supposed to be a number.</td>
</tr>
<tr>
<td>FAULT</td>
<td></td>
<td>- This is programmed a little different than designed, but this is a good thing because it prevents an empty group from being created.</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>FAILURE: Change not saved. Reason: Tabbed out of the field. In the 2nd attempt I hit ENTER and the change was saved, but the msg still was not displayed.</td>
</tr>
<tr>
<td>Surprize</td>
<td></td>
<td>UNEXPLAINED - this is &quot;sometimes&quot; behavior that I can't recreate on demand. The message is missing from the code, but otherwise it should work properly.</td>
</tr>
<tr>
<td>Defect</td>
<td>Failure Details</td>
<td>Traceback</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
<td>-----------</td>
</tr>
<tr>
<td>3</td>
<td>Failure: New group name was not saved.</td>
<td>2nd test: Worked as expected.</td>
</tr>
<tr>
<td>4</td>
<td>Failure: Program crashed - received an error (see screenshot d225.gif).</td>
<td>It tried changing the name of the Computers group to &quot;Test Computers&quot;. I tried using this same name in an earlier test (223) that did not save the edit. DB showed that the changes were saved but just weren't showing up on the screen.</td>
</tr>
<tr>
<td>3</td>
<td>Failure: Program crashed - received an error (see screenshot d225.gif).</td>
<td>During retesting, no changes were refreshed on the page after one change is made. The changes aren't saved to the database, either. There is a lot of strange, unexplained behavior. Other group names in the list are getting updated, but there is no detectable pattern. My guess is that this weird behavior is occurring because the same code is reused for all the different groups, which are built dynamically based on how many groups need to be loaded.</td>
</tr>
<tr>
<td>4</td>
<td>Failure: Program crashed - received an error (see screenshot d234.gif).</td>
<td>&quot;Input string was not in a correct format.&quot; I chose &quot;Hello World&quot; as the assignment to view grades for.</td>
</tr>
<tr>
<td>2</td>
<td>Failure: I typed &quot;bad&quot; in the score column which caused the program to crash. Missing input validation.</td>
<td>This error was caused by bad data - one student had a null grade in the database. There is a related programming error: the input should be validated before it is attempted to be displayed.</td>
</tr>
<tr>
<td>4</td>
<td>Failure: Program crashed - received an error (see screenshot d259.gif).</td>
<td>&quot;Object cannot be cast from DBNull to other types.&quot; I entered a date and a title and no other fields.</td>
</tr>
</tbody>
</table>

109
<table>
<thead>
<tr>
<th>Defect</th>
<th>Failure</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>d262</td>
<td>FAILURE: Program crashed - received an error (see screenshot d262.gif) &quot;Input string was not in a correct format&quot; - PROGRAM CRASH. The program is not coded to validate this input field. The code behind the button does not check for blank values first.</td>
<td></td>
</tr>
<tr>
<td>d264</td>
<td>FAILURE: same as test 262</td>
<td></td>
</tr>
<tr>
<td>d267</td>
<td>FAILURE: same as test 262</td>
<td></td>
</tr>
<tr>
<td>d268</td>
<td>FAILURE: same as test 262</td>
<td></td>
</tr>
<tr>
<td>d270</td>
<td>FAILURE: same as test 262</td>
<td></td>
</tr>
<tr>
<td>d273</td>
<td>FAILURE: same as test 262</td>
<td></td>
</tr>
<tr>
<td>d274</td>
<td>FAILURE: same as test 262</td>
<td></td>
</tr>
<tr>
<td>d260</td>
<td>SURPRISE: similar to test 6 - had to click OK twice to exit dialog. Walls appeared as expected, though.</td>
<td></td>
</tr>
<tr>
<td>d262</td>
<td>PROGRAM CRASH - Similar to the fault for test 6, the OK button wasn't handled properly.</td>
<td></td>
</tr>
<tr>
<td>d275</td>
<td>SURPRISE: same as test 266</td>
<td></td>
</tr>
<tr>
<td>d280</td>
<td>SURPRISE: File saved as test1.map.cht. .cht file extension is used for seating charts, .map is for classroom maps. Retesting: Worked as expected</td>
<td></td>
</tr>
<tr>
<td>d282</td>
<td>SURPRISE: File saved okay because of .cht extension added. Changed all file extensions in test plan to .cht</td>
<td></td>
</tr>
<tr>
<td>d297</td>
<td>FAILURE: Wall was drawn to end of drawing area</td>
<td></td>
</tr>
<tr>
<td>d319</td>
<td>FAILURE: Program crashed. See screenshot d319.gif. &quot;Object reference not set to an instance of an object&quot;</td>
<td></td>
</tr>
<tr>
<td>d321</td>
<td>FAILURE: Program crashed - same as d319</td>
<td></td>
</tr>
<tr>
<td>i40</td>
<td>SURPRISE: Add User Dialog stays open and works as expected</td>
<td></td>
</tr>
<tr>
<td>i78</td>
<td>SURPRISE: I noticed that the &quot;Things to do today&quot; section on the main screen does not get updated by changing either the date or the class</td>
<td></td>
</tr>
</tbody>
</table>

---

110
<table>
<thead>
<tr>
<th>Impact</th>
<th>Priority</th>
<th>Severity</th>
<th>Type</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imp 179 26</td>
<td>136</td>
<td>Surprise</td>
<td>Same as Impl test 78, SURPRISE: Noticed that the &quot;Things to do today&quot; section on the main screen does not get updated by changing either the date or the class. Will stay because the labels are never reset.</td>
<td></td>
</tr>
<tr>
<td>Imp 180 26</td>
<td>137</td>
<td>Surprise</td>
<td>Same as Impl test 78, SURPRISE: Noticed that the &quot;Things to do today&quot; section on the main screen does not get updated by changing either the date or the class.</td>
<td></td>
</tr>
<tr>
<td>Imp 1334 7</td>
<td>158</td>
<td>Surprise</td>
<td>SURPRISE: Could click on any button or menu choice. Add User dialog stayed open behind main window. No negative effects except clicking on Add New User again (see test below). GUI tool/configuration error - Dialog behavior was not set properly. Main screen should probably be locked to prevent user from clicking on something else and hiding/forgetting about the Add User dialog. Transition 1: This surprise was caused by an additional transition in the revised FSM for CIS 7.</td>
<td></td>
</tr>
<tr>
<td>Imp 180 26</td>
<td>138</td>
<td>Surprise</td>
<td>SURPRISE: Returning to the main screen displayed &quot;to do&quot; items from both days. It looks like the first two items were overwritten with the 2nd days' items (the 1st day had 4 items total). This is the same underlying fault as #39 above. The labels are not reset unless new &quot;to do&quot; items are found. For example, if the first class loaded has 4 &quot;to do&quot; items and the second class only has 2, then the first two items will be overwritten from the second class but the other two will remain untouched. Transition 39: This surprise was not caused by any additional transitions in the FSMs.</td>
<td></td>
</tr>
<tr>
<td>Imp 187 31</td>
<td>139</td>
<td>Defect</td>
<td>FAILURE: New student was saved but changes to data grid were not. Received OleDbException and program terminated (see 087.gif) After the student was added, the code calls an &quot;UpgradeGrid&quot; function to update the student grid (to include the newly added student). At this point the dataset being used by the &quot;Save Changes&quot; button is out of date, so it seems that attempting the save with the corrupted dataset throws this database exception. Transition 40: This defect was caused by an additional transition in the revised FSM for CIS 31.</td>
<td></td>
</tr>
<tr>
<td>Imp 189 31</td>
<td>140</td>
<td>Defect</td>
<td>FAILURE: Same error as test 87</td>
<td></td>
</tr>
<tr>
<td>Imp 190 31</td>
<td>141</td>
<td>Defect</td>
<td>FAILURE: Same error as test 87</td>
<td></td>
</tr>
<tr>
<td>Imp 192 34</td>
<td>142</td>
<td>Defect</td>
<td>FAILURE: Nothing was saved, program eventually crashed after navigating back and forth between student and parent screens. The parent/EC form didn’t have the appropriate student ID since the newly added student hadn’t been saved to the database yet. The feature should not be available for the new student until s/he is saved to the database. Transition 41: This defect was caused by an additional transition in the revised FSM for CIS 34.</td>
<td></td>
</tr>
<tr>
<td>Imp 196 37</td>
<td>143</td>
<td>Surprise</td>
<td>SURPRISE: Edit made in step 2 was gone after clicking Add Parent button in dialog. Reverted to original value. New parent was saved OK. Program is coded so that, if parent is added successfully, then the parent grid is refreshed from the database. This overwrites the change in the grid that was never saved to the database. Transition 42: This surprise was caused by an additional transition in the revised FSM for CIS 37.</td>
<td></td>
</tr>
<tr>
<td>Imp 199 38</td>
<td>144</td>
<td>Surprise</td>
<td>Surprised same as test 96 except for EC instead of Parent Same as above except for EC instead of Parent Transition 43: This surprise was caused by an additional transition in the revised FSM for CIS 37. 6, 7, 8 are similar</td>
<td></td>
</tr>
<tr>
<td>Imp 1100 39</td>
<td>145</td>
<td>Surprise</td>
<td>SURPRISE: Edit made in step 2 was gone after clicking Add Same as above except for Assignments Transition 44: This surprise was caused</td>
<td></td>
</tr>
</tbody>
</table>
### Summary:

<table>
<thead>
<tr>
<th>Unique Defects</th>
<th>Unique Surprises</th>
<th>Unique Faults</th>
<th>Unique Faults found by impl tests alone</th>
</tr>
</thead>
<tbody>
<tr>
<td>121</td>
<td>37</td>
<td>47</td>
<td>10</td>
</tr>
</tbody>
</table>

#### Severity scale:

- 1 = Cosmetic
- 2 = Workaround available
- 3 = Design a simple solution
- 4 = Major rework/program unstable
- NA = does not concern user

---

<table>
<thead>
<tr>
<th>Imp</th>
<th>Test No.</th>
<th>Type</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imp</td>
<td>1102</td>
<td>39</td>
<td>Surprise</td>
<td>Assignment button in dialog. Reverted to original value. New assignment was saved OK.</td>
</tr>
<tr>
<td>Imp</td>
<td>1135</td>
<td>48</td>
<td>Surprise</td>
<td>SURPRISE: Similar to tests 96 and 99.</td>
</tr>
<tr>
<td>Imp</td>
<td>1136</td>
<td>49</td>
<td>Surprise</td>
<td>SURPRISE: &quot;Changes saved&quot; dialog popped up</td>
</tr>
<tr>
<td>Imp</td>
<td>1137</td>
<td>49</td>
<td>Surprise</td>
<td>SURPRISE: &quot;Changes saved&quot; dialog popped up</td>
</tr>
<tr>
<td>Imp</td>
<td>1139</td>
<td>54</td>
<td>Defect</td>
<td>FAILURE: Exception caught. See 139.gif</td>
</tr>
<tr>
<td>Imp</td>
<td>1141</td>
<td>54</td>
<td>Defect</td>
<td>FAILURE: Exception caught. See 139.gif</td>
</tr>
<tr>
<td>Imp</td>
<td>1143</td>
<td>54</td>
<td>Defect</td>
<td>FAILURE: Exception caught. See 139.gif</td>
</tr>
<tr>
<td>Imp</td>
<td>1145</td>
<td>54</td>
<td>Defect</td>
<td>FAILURE: Exception caught. See 139.gif</td>
</tr>
<tr>
<td>Imp</td>
<td>1147</td>
<td>54</td>
<td>Defect</td>
<td>FAILURE: Exception caught. See 139.gif</td>
</tr>
<tr>
<td>Imp</td>
<td>1138</td>
<td>54</td>
<td>Surprise</td>
<td>SURPRISE: Did not intend to let the user resize this window. If s/he resizes it too small s/he can hide the username and other fields which could be confusing. Closing and re-opening sets it back to its default size.</td>
</tr>
<tr>
<td>Imp</td>
<td>1132</td>
<td>54</td>
<td>Defect</td>
<td>FAILURE: Same as 318</td>
</tr>
<tr>
<td>Imp</td>
<td>1133</td>
<td>54</td>
<td>Defect</td>
<td>FAILURE: Exception caught. See 139.gif</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Test No.</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>Changes saved dialog popped up</td>
<td>Same as above - just a different CIS.</td>
</tr>
<tr>
<td>99</td>
<td>Changes saved dialog popped up</td>
<td>Same as above - just a different CIS.</td>
</tr>
<tr>
<td>139</td>
<td>Exception caught. See 139.gif</td>
<td>Same as Fault #33. The program is not coded to validate this input field. The code behind the button does not check for blank values first.</td>
</tr>
<tr>
<td>47</td>
<td>Transition 11: This surprise was caused by an additional transition in the revised FSM for CIS 70.</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>Transition 13: This surprise was caused by an additional transition in the revised FSM for CIS 1, 2, and 3</td>
<td></td>
</tr>
</tbody>
</table>
References


