Strategies for Web and GUI Testing

Mark Blackburn
Software Productivity Consortium
blackburn@software.org
(703) 742-7136

Aaron Nauman
Software Productivity Consortium
nauman@software.org
(703) 742-7104

Abstract

This paper describes strategies for functional testing of graphical user interfaces (GUIs) and web-based applications because these activities are manually intensive and a costly problem. Tools exist for regression testing of interface functionality through capture/playback mechanisms, but this approach is manually intensive and difficult to maintain. There are better ways to design for testability that provide the infrastructure for other types of more cost-effective test automation. This paper explores and discusses these approaches.

Contents

Introduction

The cost of verifying and testing computer systems typically is underestimated. Factors such as increased complexity, short release schedules, and lack of well-defined requirements contribute to verification and testing efforts that consume 40% to 70% of development effort over a product's lifetime. Graphical user interfaces (GUIs) to computer systems, whether application interfaces (e.g., Excel), web-browser-based, or device interfaces (e.g., cell phones, navigation systems), are ubiquitous. Verifying and testing systems that include GUIs offers additional unique challenges.

GUIs typically have a large number of potential inputs and input sequences. For example, a cellular telephone has a large number of states and inputs. To reasonably verify the system's functionality requires a large amount of testing. Performing these tests manually is costly and can be practically impossible. Therefore, it is necessary to perform automated testing. Many organizations have difficulty applying test automation to systems with user interfaces because test automation is not well-understood, is not supported by available tools, or is complicated by the system's design.

As Fewster and Graham point out, test tools cannot replace human intelligence in testing, but without them, testing complex systems at a reasonable cost will never be possible [Fewster 1999]. There are commercial products to support GUI testing, most based on capture/playback mechanisms. Organizations that have adopted these tools have realized that these approaches are still manually intensive and difficult to maintain. Even small changes to the application's functionality or GUI can render captured test

Excel is a Registered Trademark of Microsoft Corporation.
Java is a trademark of Sun Microsystems, Inc.
Visual Basic is a registered trademark of Microsoft Corporation.

Copyright © 2004, Software Productivity Consortium NFP, Inc. and T-VEC Technologies, Inc. All rights reserved. This document is proprietary property of the Software Productivity Consortium NFP, Inc. The contents of this document shall be kept confidential pursuant to the terms of the Membership Rules, as amended from time to time, of the Software Productivity Consortium NFP, Inc. This document shall only be disseminated in accordance with the terms and conditions of those Rules. All complete or partial copies of this document must contain a copy of this statement.

sessions useless. More importantly, these tools do not help test organizations figure out what tests are necessary, nor do they give any information about test coverage of the GUI functionality.

There has been limited systematic study of this problem resulting in an effective testing strategy that is not only easy to apply but also scalable to increasing test complexity, test coverage, and completeness of the test process [Belli 2003]. Therefore, this report looks at strategies to reduce cost through a combination of better design for testability, systematic GUI requirement analysis, and improved test design. In addition, it describes how types of test automation can work better with specific test design and implementation strategies.

Context and Scope

This paper provides members of the Software Productivity Consortium (Consortium) with information on strategies for functional testing of GUI and web-based applications. This paper assumes readers work in an organization that struggles with GUI testing. It also assumes readers are interested in functional testing as opposed to performance, load, or stress testing. It discusses the need to partition the data processing logic and other server-side functionality from the GUI because it is extremely inefficient to thoroughly test system functionality strictly through a GUI.

A GUI has aesthetic attributes, including how nice it looks and how usable it is. This paper does not address aesthetics of the GUI. Experts should perform usability analysis of a system prior to user interface development. In addition, experts must verify rendering of the GUI manually. For example, in most cases, it is not feasible to use tools to determine whether buttons are laid out correctly. Therefore, the paper does not discuss this type of verification.

Specialized user interfaces, such as devices, wireless phones, avionics displays, kiosk, graphic rendering, and moving maps (air traffic control), may be of interest to some Consortium members. This white paper focuses on general guidelines for testing the typical GUI and web GUI. However, much of the discussion on design for testability applies to all GUI-based testing.

There are other challenges in GUI testing—the paper discusses some of these along with strategies to address some of the issues associated with the challenges. Some issues are not strictly technical and require organizational change to affect a solution or an improvement to the current process.

Audience and Benefits

This paper is applicable to managers, project leads, software developers, quality assurance staff, and test engineers who are responsible for managing, planning, accessing, estimating, and performing GUI and web-based testing. This paper should provide guidance to help Consortium members:

- Understand how to better partition the GUI testing responsibilities from the data processing testing to reduce the GUI testing complexity
- Understand the challenges in web and GUI testing to support better planning of upfront design and testing efforts to reduce late rework
- Understand how to design the system for GUI testability to support increased test automation and reduce manual testing
- Apply strategies that more systematically define tests to cover the functionality of the web and GUI interface
- Understand how tools and test automation can support these testing strategies

Organization of This Paper

The first section of the paper focuses on design strategies for making the GUI more testable. The next section discusses analyzing, elaborating, and prioritizing the GUI requirements to drive the test effort. The third section discusses test automation techniques and the tools that support them. The paper closes with some miscellaneous topics, including test coverage and organization change, and the summary.

Design for Testability

Figure 1 provides a conceptual representation of a System Under Test (SUT) that includes a GUI. The SUT typically is composed of two or more parts: the GUI code and the logic and data processing code. The GUI code accepts user inputs, passes that information to the data processing code, and updates the display as directed by the data processing code. The user inputs typically include mouse movements, menu selections, and key presses. Often the coupling of the user interface code and data processing code is a significant impediment to testing.

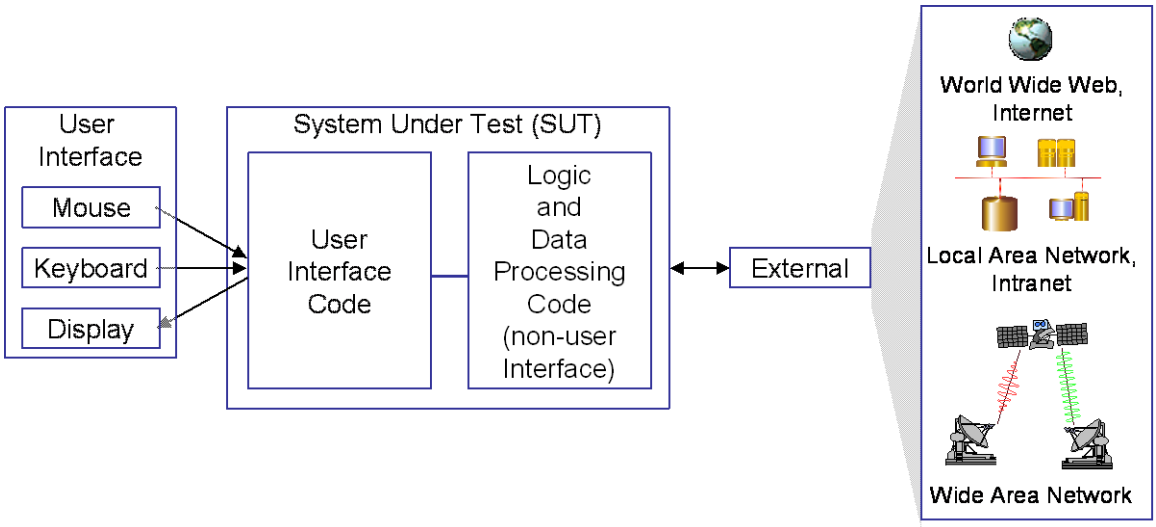

Figure 1. Generic Elements for User Interface Testing

When testing a GUI (or any application), three key properties affect testability [Williams 1982]:

1. Predictability is a measure of how difficult it is to determine what a test's outcome should be.
2. Controllability is a measure of how difficult it is to provide inputs to the system to drive its execution.
3. Observability is a measure of how difficult it is to capture and determine whether the test results are correct.

The complexity and consistency of an application affects its predictability in terms of testing. As complexity increases and consistency decreases, requirements should document more precisely the features and behaviors of the system to support testing. The architecture and design of an application impacts its controllability and observability. Designing a system for testability eases the effort required to test it, and is often critical to supporting test automation.

Guideline: Design for testability from the beginning of the project.

Once the system architecture has been designed and implemented, it is very difficult to modify it to support testing. Designing for testability must occur during the initial phases of development and be communicated to designers and implementers of the system.

Guideline: Separate GUI from data processing logic.

As shown in Figure 2, the GUI and data processing logic should be separated programmatically to provide more controllability and observability. Significant amounts of the application's functionality should be tested independently of the user interface. Changes to GUI layout should not impact the data processing code. Updates to the data processing code should not unnecessarily impact the GUI.

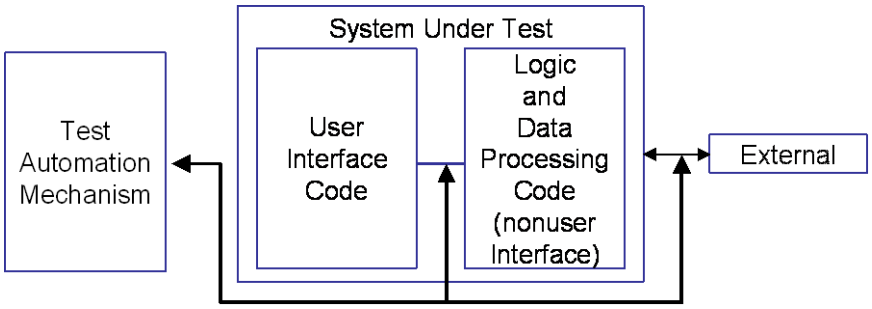

Figure 2. Program-Based Interfaces to SUT

Guideline: Provide application interfaces to support controllability and observability.

Figure 2 illustrates an interface between the GUI and data processing code. Well-defined interfaces support controllability and observability, which ease testing efforts. Testers can use the interfaces to initialize the system, set the test inputs, and capture the test results. If the test requires sequences of events, then the interfaces support verifying intermediate results are correct. This approach provides a basis for test automation and allows GUI testing to be separated from data processing testing.

Because it is practically impossible to test a system fully through black-box testing at the system level, systems should be tested at multiple levels. Developers should test at the unit level. Developers and test engineers should test at the component level and at various levels of integration. Finally, testing should be performed at the system and possibly the acceptance levels. Testing at multiple levels allows the testing to begin earlier and helps make the system-level testing tractable. While it is still necessary to perform system testing through the GUI to demonstrate its functionality, the number of test cases to demonstrate this can be reduced by one to several orders of magnitude by testing data processing functionality independently of the GUI. This approach can reduce the manual testing effort and cost significantly, but it depends on architecting the system to support it.

GUI Controls and Test Automation

GUI controls, also known as components or widgets, present information and accept inputs. Common controls include text fields, push buttons, menus, and list boxes. GUI test tools use GUI drivers that are associated with control types to record user events and trigger events. User events include entering text in a text field, clicking a button, or selecting an item from a menu. GUI test tools make assumptions of how the windows and controls in the interface are structured. The GUI drivers of many tools expect to recognize and operate controls using a set of standard protocols or messages; therefore, it is important to design or use GUI objects that can support test automation [Pettichord 2002].

Guideline: Ensure that the test tools are compatible with the GUI development tools.

Guideline: Use GUI objects that have support for automated testing.

Guideline: Avoid the use of custom controls, unless the custom control libraries are developed to support testing.

Most GUI test automation tools support a standard set of graphical controls provided with development environments. Often they do not work well or at all outside these environments. In addition, custom controls that are not part of this standard set often are not supported by the test automation tools. Assess the impact on testing of using custom controls before using them in development.

Guideline: Define standards for naming GUI objects. Make sure each GUI object is named uniquely.

Most GUI test automation tools make heavy use of GUI control names for storing information and accessing the control. Clear, concise, meaningful control names ease the process of working with these tools.

Guideline: Add features to application infrastructure to support testing.

The system designers should consider inclusion of other features that support testing, such as verbose output, event logging, assertions, resource monitoring, test points, and fault injection hooks [Pettichord 2002]. Verbose output and event logging can help trace bugs that are difficult to replicate. Assertions report incorrect assumptions in the application when it is running in debug mode. Test points and fault injection hooks support test execution.

Define the GUI Requirements

Testing is an activity that helps verify that a system satisfies the requirements of its functionality. Tests are derived from the requirements and executed on the system in order to verify them. Requirements are the basis of functional testing. They are captured a variety of ways with varying degrees of precision. Lack of, ambiguities in, or errors in requirements account for about half of all problems discovered during testing.

Requirements for a GUI, if specified, are some times specified as use cases or usage scenarios. For testing, the use cases are refined to specific test scenarios that define normal behavior and alternative scenarios. These alternative test scenarios are related to abnormal or unexpected behavior. Test scenarios can be defined as interaction sequences that define the sequences of interactions with GUI objects [White 2000]. An interaction sequence is one complete path through one scenario. There are often many complete paths through the application to cover the end-user scenarios.

Guideline: Understand the requirements allocated to the GUI.

As mentioned, there are many ways requirements are defined; however, it is often the case that the GUI requirements may not be documented at all, and testers must determine the requirements allocated to the GUI by interacting with developers, domain experts, and customers.

Guideline: Prioritize the requirements.

This is important for planning because the time for GUI and web deployment often is constrained to some predefined release schedule. It is better to test the most critical functions of the system first, rather than randomly "pounding" on the keyboard in a "poke and hope" type strategy.

Guideline: Construct interaction sequences for the more important scenarios of the application.

This can be complex, so this task can be broken down into the construction of a navigation map of the GUI or web interface and one or more state machine models that define the different GUI states and the event transitions between states.

Guideline: Test abnormal or unexpected user events.

It is important for GUIs to handle unexpected behavior because users who are often unfamiliar with the application can interact with the system in unexpected ways, causing the system to traverse some unexpected path that causes a failure.

The following section provides an example. The sections following it provide more details on analyzing GUI scenarios to support test design.

Example

Although the authors are not advocating a particular approach for defining requirements, several members are investigating use-case-based approaches as used in this example. Use cases are effective in capturing business functionality, but they are somewhat lacking in capturing usage and behavior characteristics of the system [Meyer 1999]. This section provides an automated teller machine (ATM) example to illustrate a use case, scenario, and interaction sequence. Figure 3 depicts a use-case diagram for an ATM. It shows only a partial and simplified set of use cases compared to those for a real ATM. The example treats the bank accounting system as part of the ATM system rather than as an external actor.

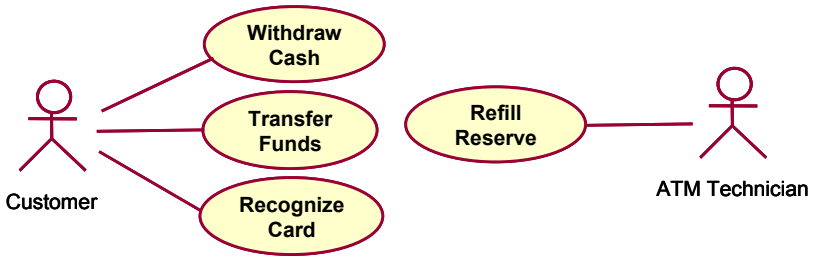

Figure 3. Use Case Diagram—ATM Example

To support testing, the required behavior must be identified and expressed in terms of some interface. From a high-level perspective, the use cases denote various broad behaviors, but this is not sufficient for building test cases. More detailed information is required to formalize the interfaces and required behavior.

Use Case

Table 1 specifies a portion of the use case for a withdrawal. The portion shown omits alternate scenarios that would be present in the complete use case, such as handling timeouts, the consequences of an invalid personal identification number (PIN), and foreign language options.

Table 1. Withdraw Cash Use Case (Extract)

Name/Goal	Withdraw Cash
Actors	Customer
Preconditions	1. The Recognize Card use case has completed successfully.
Postconditions	1. Current balance = original balance – dispensed cash – service charge. 2. Cash reserve = original reserve – dispensed cash.
Main scenario	1. Customer requests cash withdrawal. 2. ATM asks Customer for amount. 3. Customer inputs desired amount. 4. ATM dispenses amount of cash requested, card, and receipt.
Alternate scenarios	2a. Customer card linked to multiple accounts. 2a1. ATM asks Customer to select one account (savings, checking, or credit) for the transaction. 2a2. Customer selects an account. 2a3. Go to Step 2. 2b. Customer card linked to nonaffiliate account. 2b1. ATM asks whether Customer will accept $2 service charge. 2b2. Customer accepts charge. 2b3. Go to Step 2. 2c. Card linked to credit account. 2c1. ATM asks whether Customer will accept $10 service charge. 2b2. Customer accepts charge. 2b3. Go to Step 2. 4a. Customer requested amount not a multiple of $20. 4a1. ATM asks Customer to enter a multiple of $20. 4a2. Go to Step 3. 4b. ATM has less cash in reserve than requested amount. (etc.)

Prioritize Requirements

It is common for requirements to be poorly defined or nonexistent. However, test requirements should be developed and prioritized so that test case development can be prioritized when time or resources are limited. The following provides some guidelines for prioritizing requirements.

Guideline: Identify the test requirements and test cases for the main scenarios of the SUT followed by the alternative scenarios.

The main scenarios define the primary user functionality. The alternative scenarios define how to handle unusual or unexpected input events. In addition, some requirements can be subsumed by other test cases. If the primary test cases cover part or all of another requirement, that requirement is subsumed and covered by the other test.

Guideline: Prioritize the requirements by criticality.

Guideline: Prioritize the requirements by the phase of the delivery of the product.

If the test team is working with the design team, tests can be prioritized based on the expected availability of the functionality and importance. For example, if the withdrawal funds functionality is going to be completed before the transfer funds capability, then identify, develop, and plan to test the requirements for withdrawal first.

Table 2 illustrates a simple example for prioritizing requirements. The priorities in column two indicate that the tests for the main scenario and alternative scenarios 4a and 4b are the most important and should be tested first. Next, alternative 2a should be tested second. This should be followed by alternative scenarios for cards linked to other nonaffiliated banks and credit cards.

Table 2. Prioritizing Requirements Example

Requirement Scenarios	Priority
Main scenario: withdrawal cash	1
Alternative scenario: 2a. Customer card linked to multiple accounts.	2
Alternative scenario: 2b. Customer card linked to nonaffiliate account.	3
Alternative scenario: 2c. Card linked to credit account.	3
Alternative scenario: 4a. Customer requested amount not a multiple of $20.	1
Alternative scenario: 4b. ATM has less cash in reserve than requested amount.	1
…	

Prioritizing requirements can be important during the test planning and estimation phase of the project. Prioritization of the test design and test execution also can be performed in terms of a more detailed representation of the requirements such as scenarios, interaction sequences, or decision tables. Examples of these more detailed representations follow.

Scenario

This section provides an informal discussion about developing interaction sequences for the main scenario 1, Customer requests cash withdrawal, described in Table 1. The following is a list of the scenario steps:

- Customer requests cash withdrawal.
- ATM asks Customer for amount.
- Customer inputs desired amount.
- ATM dispenses amount of cash requested, card, and receipt.

This is a simplified scenario, which includes system queries and interactions that are part of the scenario to deliver cash to the user. If a test for this scenario is performed manually, then it may not be necessary to formalize the scenario into an interaction sequence; however, as illustrated in the next section, the analysis to define an interaction sequence exposes important details for designing the test cases.

Interaction Sequences

A useful way to define the detailed requirements of the GUI is to map the relationships of the GUI objects based on events that transition the GUI to different states. The scenario described above is now

represented as GUI map, as shown in Figure 4, which shows the different GUI windows and the relationships between the windows. A GUI map can represent the various screen images, including menu dropdowns, user forms, and screen messages. Even though some of the details are not shown, the map is more complex than the simple list of steps in the scenario. Analyzing the sequences uncovers additional details. The map makes it more obvious that the application can transition to different states based on different event sequences. For example, if the user enters the PIN incorrectly, there is a screen that permits the user to reenter the PIN a few times, before the transaction is aborted, and the ATM machine consumes the user's card. If the user selects an amount for withdrawal that exceeds the current balance of the account, the system may present another screen (or window), indicating insufficient funds, and permit the user to reenter a smaller amount. If the user selects Transfer or Inquiry there are alternative maps that correspond to different states of the system (not shown here). There are at least eight different interaction sequences reflected in this GUI map. The following sections discuss how to formalize a scenario associated with a map and describe some approaches for systematically identifying a more comprehensive set of interaction sequences.

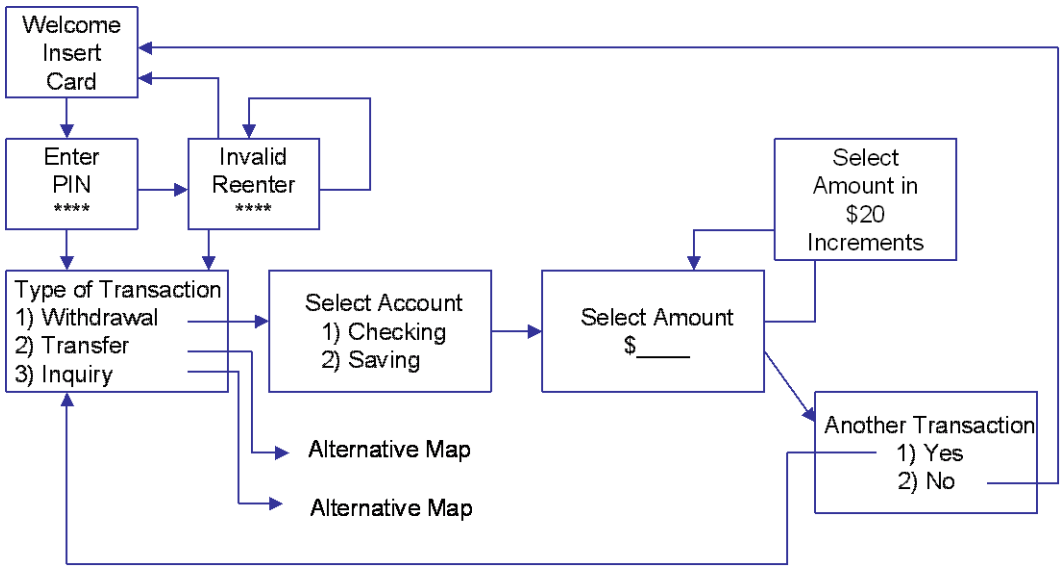

Figure 4. Partial GUI Map for Withdrawal Cash Scenario

Deriving Interaction Sequences

A few structured approaches are useful in identifying the interaction sequences for a GUI scenario. This section describes how to derive interaction sequences by using a state machine model (also known as a finite state model [FSM]).

It is typical to represent an FSM using a state transition diagram (STD) as shown in Figure 5. A STD is a directed graph, having an entry node and an exit node, and there is at least one path from entry to exit. Given some state s1 (e.g., Welcome Insert Card state), and an event e1 associated with some input (e.g., inserting the card), there is a function that maps the state s1 and the input e1 to the next state s2 (e.g., Enter PIN).

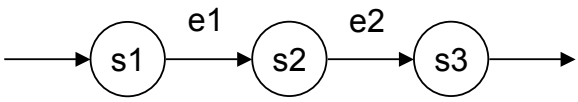

Figure 5. State Transition Diagram for Finite State Machine

Guideline: Create tests for each path through the STD.

By representing some part of the scenarios with a FSM, interaction sequences are identified by making traversals through the state machine to discover possible alternatives. It might take several FSMs to define all the scenarios of a GUI.

To represent a GUI, inputs events are operations on objects such as window, menus, icons, and fields. The linked set of transitions from one state to another that is associated by a sequence of user inputs defines an interaction sequence. If the set of transitions results in an expected system response, then that is a complete interaction sequence that has observable results.

Figure 6 shows a portion of the GUI map for the ATM of Figure 4, with labeled states (S1, S2, …, S8). Figure 7 is an equivalent representation of Figure 6, but GUI events are added to illustrate the events that cause transitions to the different states. By analyzing the paths through the STD shown in Figure 7, a set of interaction sequences with associated outcomes can be derived as shown in Table 3. These interaction sequences define some of the test cases that should be performed to ensure that the ATM GUI functions correctly.

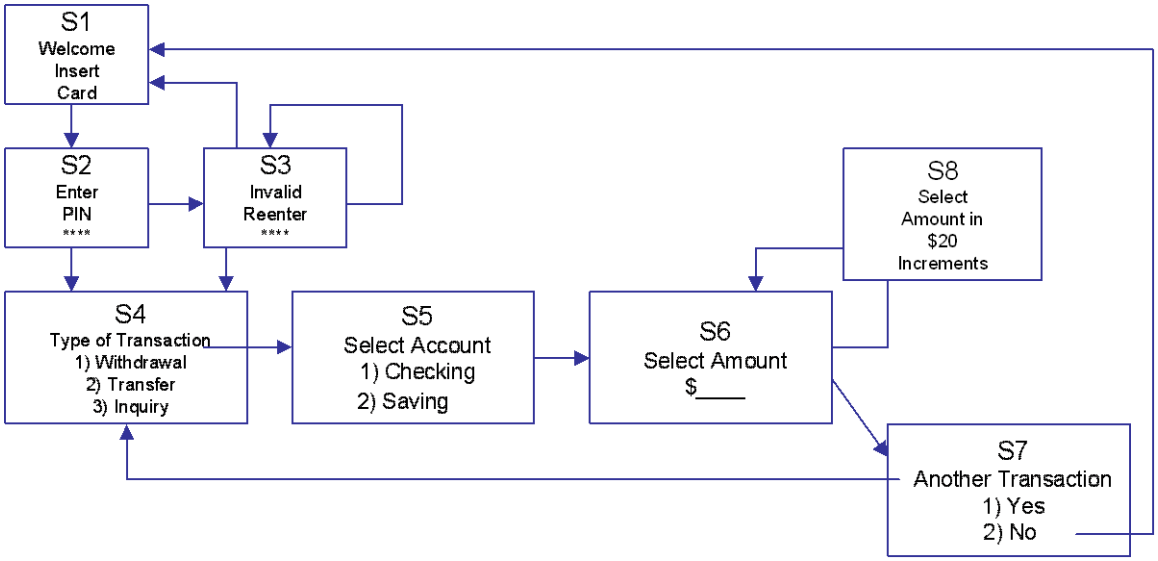

Figure 6. Label States of GUI Map

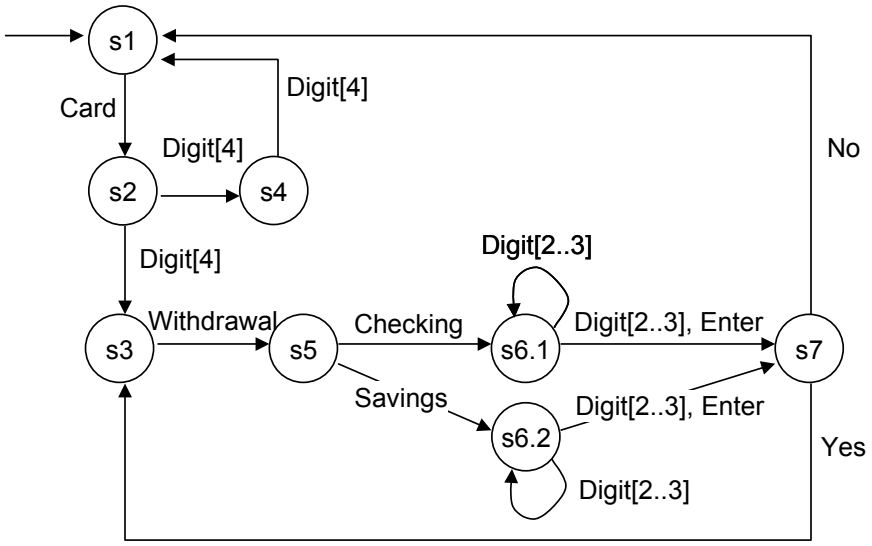

Figure 7. Partial State Transition Diagram for ATM

To better understand the traversal of the paths, consider the description of the example sequences defined in Table 3.

- The first sequence defines the case where a user inserts the card, enters the 4-digit PIN incorrectly, then attempts to enter the 4-digit PIN a second time, and the ATM consumes the card.
- The second sequence defines the case where the user inserts the card, enters the 4-digit PIN correctly, selects Withdrawal as the type of transaction, selects Checking as the account, enters a 2-digit amount, selects Enter, and responds No when asked for another transaction. The outcome is dispensed cash and an ejected card.
- The third sequence is similar to the second except that the amount entered has 3 digits.
- The fourth sequence is similar to the second transaction except that the account type is Savings.
- The fifth sequence is similar to the fourth except that the amount entered has 3 digits.
- The sixth sequence is similar to the second except that the response to the prompt for another transaction is Yes instead of No.

Table 3. Interaction Sequences for Partial ATM

Interaction Sequence	Outcome
1) Card, Digit[4], Digit[4]	Consume card
2) Card, Digit[4], Withdrawal, Checking, Digit[2], Enter, No	Dispense cash, Eject card
3) Card, Digit[4], Withdrawal, Checking, Digit[3], Enter, No	Dispense cash, Eject card
4) Card, Digit[4], Withdrawal, Savings, Digit[2], Enter, No	Dispense cash, Eject card
5) Card, Digit[4], Withdrawal, Savings, Digit[3], Enter, No	Dispense cash, Eject card
6) Card, Digit[4], Withdrawal, Checking, Digit[2], Enter, Yes, [cycle]	Dispense cash, Eject card
…	

Identify Unexpected Event Sequences

The sequences in Table 3 for the main scenario verify that the functionality satisfies the requirements. However, it is also important to test how the system behaves when unexpected inputs are received to ensure that the SUT handles them appropriately. As an example, refer to Figure 6, an unexpected event from state S4 (Type of Transaction) is created by entering a digit other than 1, 2, or 3, such as 0 or 4.

Guideline: Identify unexpected event sequences to test for system failures and graceful error handling.

Fault injection is a technique for testing how software handles unusual situations, such as an unexpected sequence of events. Effective fault testing often requires testability hooks for simulating faults [Petticord 2002], therefore design for testability also is important for fault testing.

There are systematic approaches for analyzing the FSM representation to identify unexpected event sequences, but it requires transforming the FSM into an event sequence graph [Belli 2003]. This is beyond the scope of this version of the paper.

Approaches to Test Automation

Testing can be broken down into a set of activities. The following list includes these activities in increasing order of testing maturity.

- Test Execution and Result Analysis – running the tests and checking the test results
- Test Design – determining the starting state, test input value, and expected test output values
- Test Planning – analyzing the requirements to determine what to test and in what order
- Test Management – using tools to management testing artifacts
- Test Measurement – using techniques to measure complexity of the system and test coverage

In terms of test automation, test design and test execution are the primary focus. Automating test execution is the easiest to accomplish. There are many tools and techniques that support it. Test design is a far more expensive activity and also more difficult to automate with few tools available to support it. This section provides a broad summary of various approaches to test automation and the tools that support it.

The effectiveness and feasibility of automated test execution depends on the testability (controllability and observability) of the SUT. Currently, most GUI testing, both test design and test execution, is manual. It requires manual determination and manual entry of test cases and associated data. Ensuring that all combinations of logic are tested requires significant human expertise (domain expertise) and time. This test creation process is error-prone, with testers sometimes unintentionally repeating cases while leaving others untested.

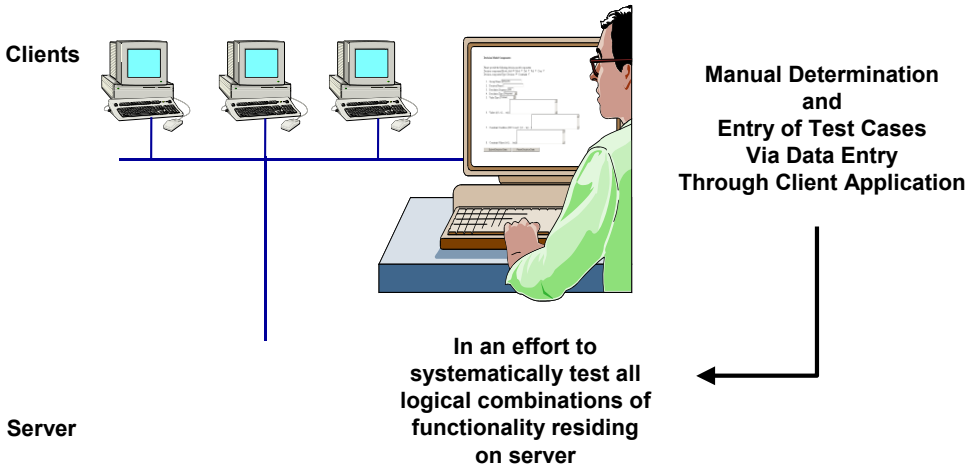

Figure 8. Manual Testing

Test Execution

For functional testing, test execution typically has a common pattern that includes:

- Initialize the SUT.
- Loop through a set of test cases, and for each test case:
  - Initialize the output to a value other than the expected output (if possible).
  - Set the inputs.
  - Execute the SUT.
  - Capture the output and stores off the results to be verified against the actual output at some later time, when a test report can be created.

Test Scripting

Test scripts are general mechanisms that automate the process of test execution. Most test execution tools are based on some form of test script that runs automatically without human interaction. Test scripts can be developed using standard application languages such as VB, C, C++, Java, Perl; specialized languages such as Tcl, and Python; or custom languages supported by test execution tools. Test scripting usually requires programmatic interfaces to the SUT to control test execution.

Capture Playback

Capture/playback tools capture sequences of manual test execution operations performed by an engineer in a test script. These test scripts are then executed (playback) to verify that functionality is still correct. Although the underlying mechanisms of capture/replay are test scripts, most users operate these tools using the recording and playback features, rather than modifying the test scripts directly. The capture/playback approach requires engineers to manually determine and execute the test cases initially. Subsequent executions of the test are automated. The primary benefit of this approach is regression testing. However, there are shortcomings to capture/playback:

- When the system functionality changes, the capture/playback session can be invalidated and require recapturing. The recapturing process is still manual and typically takes as long to perform as it did the first time.

- Capture/playback tools are designed to record events either by context (object) or absolute position (analog). Recording based on context, such as name, prevents invalidating scripts when GUI controls are repositioned. Analog capture stores the absolute pixel position of each user event. Using this approach, simply moving a button a couple of pixels can invalidate a script and require recapturing. The support for context or object recording varies from tool to tool.
- As mentioned in the section on design for testability, the effective use of capture/playback tools often depends on test tool's support for the development environment. Users often find that test tools depend on the development environment, and test tools that were able to recognize GUI objects created by C++ development systems are not able to recognize objects within other applications [Imbus 1997].
- More advanced capture/playback tools often provide some level of abstraction when recording user actions and increased portability of test scenarios (for instance, by recording general browser actions instead of mouse actions on specific screen coordinates), but changes in the structure of a website may prevent previously recorded test scenarios from being replayed and hence may require regenerating and re-recording a new set of test scenarios from scratch [Benedikt 2002].
- Many of the capture/playback tools provide a scripting language, and it is possible for engineers to edit and maintain such scripts, but this does require the test engineer to have programming skills.

Capture/playback may sometimes reduce the effort over the completely manual approach; however overall savings is usually minimal.

Extended Test Scripting

A number of test execution approaches have extended the test scripting concepts in an effort to make them more powerful and/or easier for nondevelopers to use. These tools are commonly called third-generation tools and use abstraction mechanisms to bring test scripting to a higher level. All these techniques require at least one user with software development skills to manage the underlying test scripts.

Data-Driven Approaches

Data-driven approaches augment test scripts with data stored in a database or spreadsheet. The test script executes the test multiple times using different sets of data. This approach eases maintenance because data can be updated independently of the test script.

Keyword-Based Approaches

Keyword- or action-based approaches help insolate test engineers from test development and maintenance. Test designers with application expertise but little or no programming skills combine action words or keywords to define the test cases. Each action or keyword has a mapping to a test script fragment that implements the action. The tool combines the test script fragments associated with the sequence of keywords and executes them to run the test. A test automation engineer with programming skills develops and maintains the scripts for each keyword. In some cases, the keywords are parameterized for additional flexibility.

As an example, the test engineer might create a test script with the following keywords and parameters.

```
Logon “joecool” “hispassword”
SelectRecord “5551212”
DeleteRecord
Logout
```

Each keyword would have a corresponding test script fragment that would implement each keyword on the target system.

Window-Based Approach

The window-based approach derives from the capture/playback tools where the GUI controls (e.g., fields and menus) are associated with parameter values that are represented in a spreadsheet. Testers define scenarios of windows and associated sets of parameter values to cover various test cases. The test execution tools transfer the values in the spreadsheet into the associated GUI controls and perform additional actions such as pressing buttons or selecting menu to execute the test.

Object and Class-Based Approaches

The object and class-based approaches extend test scripting with object-oriented (OO) concepts and include specialized functionality for controlling the GUI through an OO approach. The tools typically include an object model that maps to and can be used to control GUI widgets. In addition, new objects and classes can be defined with custom functionality. The approach is an OO equivalent of the keyword-based approach where the test scripts are associated with attributes and methods of the object model.

Automated Test Design

Test design is the process of analyzing the requirements targeted for testing and determining test inputs necessary to perform the test. It also includes determining the expected output value produced by the test case. This section describes some techniques and tools for automating test design.

Monkey Testing

GUI applications should provide good exception handling mechanisms to respond to incorrect or illegal inputs with constructive warnings to help the user to move in the right direction. In order to validate this behavior, tests could be generated systematically as discussed in the section Identify Unexpected Event. Often, such events drive software to an illegal state causing a system crash. If an organization does not have the time or skills to support a more rigorous approach to identify fault-based test cases, then monkey testing can play a role. Monkey testing is an approach where test scripts are written to randomly click buttons and send key input to the application in an attempt to make the application crash. Logging the keystrokes is required in order to identify the sequence of inputs that led to the crash.

James Tierney, former Director of Testing at Microsoft, reported in internal presentations that some Microsoft applications groups have found 10 to 20 % of the bugs in their projects by using monkey test tools. Monkey testing should not be the only testing because it finds only bugs that cause the system to crash and does not verify that the system operations correctly. Monkeys do not understand the application and will not add much value to embedded systems, software running in simple environments, or projects that are difficult to automate [Nyman 2001].

Website Exploration and Model Checking

Determining interaction sequences for dynamic websites is difficult. Some tools, such as VeriWeb tool by VeriSoft, automatically discover execution paths through systematic exploration. Unlike traditional static web content analysis, which is limited to the exploration of static links, VeriWeb navigates through dynamic components of websites, including form submissions and execution of client-side scripts [Benedikt 2001]. There are a number of benefits of this tool, including the ability to plug in different types of error-checking. However, there are also limitations, such as the expected output of the tool is not verified during the test process. This class of tools is relatively new, and advancements in terms of capabilities and usability are expected in the future.

Model-Based Test Generation

There are several approaches to model-based test design to support GUI testing. Most of the approaches use automated means to extract the test sequences from a finite state machine. Once the tests are generated, they are transformed into test scripts that execute the test sequences in a manner similar to the capture/playback tools. The key advantage of this technique is that the test generation can systematically derive all of the sequences of some given length associated with an FSM. The website www.model-based-testing.org has many papers that discuss various approaches to modeling and test generation.

Test Automation Framework

The Consortium's Test Automation Framework (TAF) supports automated test design and test execution. It has been used by a Consortium member to test the functionality of a web application. The member company developed a model of the application's requirements and generated tests from it. The test cases were then automatically transformed into the WinRunner[1] scripting language for automated test execution against the web application.

Using the TAF approach, test engineers work in parallel with developers to refine the requirements and model them to support automated test design and test execution. The following outlines the process, as depicted in Figure 9:

1. Working from whatever requirements artifacts are available, testers create a model using a tool based on the Software Cost Reduction (SCR) method [Alspaugh 1992], such as the SCRtool [Heitmeyer 1996] or T-VEC Tabular Modeler (TTM). Tables in the model represent each output, specifying the relationship between input values and resulting output values. Models are automatically checked for inconsistencies. The tester interacts with the requirements engineers or analysts to validate the model as a complete and correct interpretation of the requirements.
2. The tester maps the variables (inputs and outputs) of the model to the interfaces of the system in object mappings. The nature of these interfaces depends on the level of testing performed. At the system level, the interfaces may include GUI widgets, database application programming interfaces (APIs), or hardware interfaces. At the lowest level, they can include class interfaces or library APIs. The tester uses these object mappings with a test-driver pattern to support automated test script generation. The tester works with the designers to ensure the validity of the mappings from model to implementation.

[1] WinRunner is a capture/playback tool developed by Mercury Interactive.

3. The T-VEC tool generates an optimal set of test vectors for testing each (alternative) path in the model. These test vectors include test inputs and expected test outputs, as well as model-to-test traceability.

4. T-VEC generates the test drivers using the object mappings and schema. A schema is created once for each test environment. The schema defines the algorithmic pattern to carry out the execution of the test cases. The test driver executes in the target or host environment. The test drivers typically are designed as an automated test script that sets up the test inputs enumerated in each test vector, invokes the element under test, and captures the results. The test drivers are executed in the same way as the test scripts produced by capture/playback tools or via manually developed test scripts.

5. Finally, T-VEC analyzes the test results. It compares the actual test results to the expected results and highlights any discrepancies in a summary report.

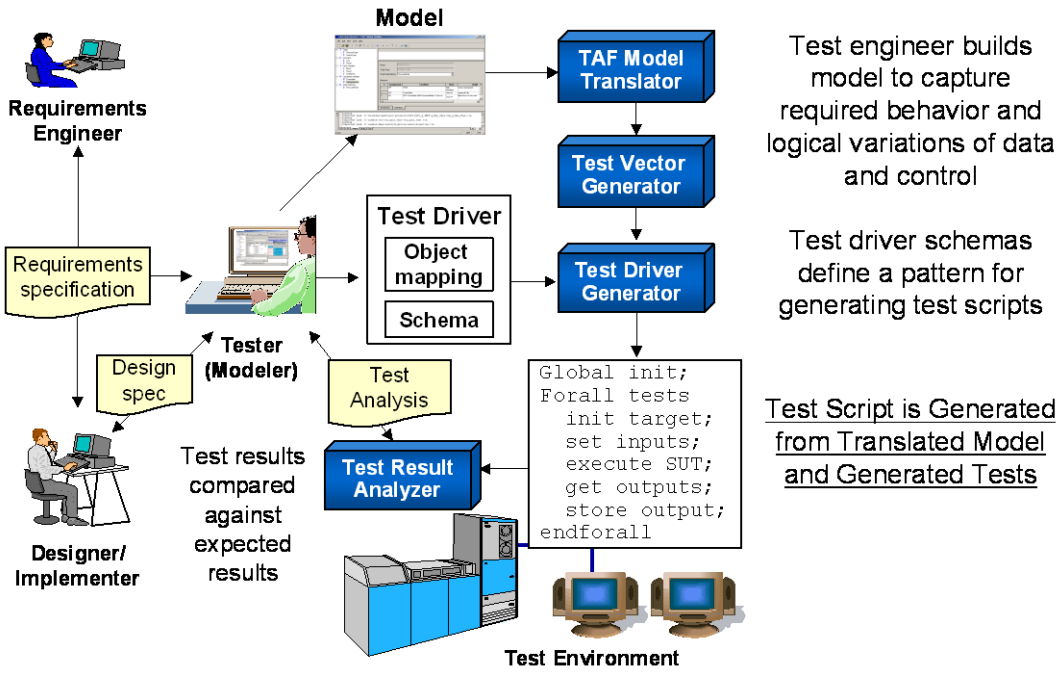

Figure 9. Model-Based Test Automation

Some key advantages of this approach include:

- Requirement defects are discovered early through the refinement and modeling process and through model checking.
- When system functionality changes or evolves, the logic in the models is updated, and all related tests and test scripts are regenerated using the existing test driver schema.
- If the test environment changes, only the test driver schema requires modification. The test scripts associated for each model can be re-generated without any changes to the model.
- Related models can share the same test driver schema to produce test scripts for a specific test environment.
- The model helps refine unclear and poorly defined requirements. Once the models are refined, tests are generated to verify the SUT. Eliminating model defects before coding

begins, automating the design of tests, and generating the test drivers or scripts result in a more efficient process, significant cost savings, and higher quality code.

These results are common among users of model-based testing approaches. The initial expectation is that model-based testing supports automated test generation, but the unexpected benefit achieved is better understanding of the requirements, improved consistency, completeness, and most importantly, early requirement defect identification and removal. Models provide a means for stakeholders to better understand the requirements and assist in recognizing omissions. Tests automatically derived from the model support requirement validation through manual inspection or execution within simulation or host environments.

Test Automation Guidelines

Guideline: Understand how test automation tools apply to specific testing activities, and evaluate the tools that fit your needs.

There are many different testing activities. These activities are supported to various degrees by test automation tools. For example, a comprehensive guide to selecting user-interface development tools cited 66 issues to consider in the areas of usability, functionality, flexibility, portability, support, and cost without any mention of testability [Valaer 1997]. There are no comprehensive tools that support all testing activities; therefore, it is important to assess the project's needs and select a set of tools that best support them.

Guideline: Use small pilot projects before starting a new project to better understand the capabilities of the test automation tools, to understand how they work with the development tools and architecture, and to access how the tools fit into the existing process.

Guideline: Use object-recording mode rather than analog recording mode when using capture/playback tools.

Test-scripting-based mechanisms can provide significant support for automated test execution, but programming expertise is required.

Guideline: Use monkey testing or fault-based testing to check for unexpected event handling if you do not have the time to develop fault-based tests systematically.

Guideline: Use model-based testing if you want to automate the test design, test-script generation, and test execution.

Guideline: Ensure that the test automation tools have a logging mechanism so that errors can be traced to their source.

Other Considerations

In addition to the technological issues, there are other practical issues that must be addressed as part of using test automation.

Knowing When to Stop Testing Using Test Coverage Estimates

Often, testing stops when the scheduled release date arrives rather than when testing is deemed complete. The resulting poor quality can have negative impact on a product, brand, and/or company. There are several ways to make more educated decisions about test completion.

Because testing should be requirements-based, it is best to determine whether testing is complete based on whether all requirements have been tested. This involves identifying what requirements are going to be tested early in the process and tracking test design and execution against the testing goals. With the use-case-based approach described earlier, this would entail measuring the percentage of interaction sequences covered by the test cases.

Structural test coverage provides another way to assess test coverage. It measures what percentage of the system's implementation is exercised by the test cases. This percentage is obtained by tracking which paths are executed in the system versus the total set of paths in the system.

Guideline: Develop tests to cover all the interaction sequences.

Guideline: Develop tests to cover all the interaction sequences for the prioritized interaction sequences.

If time or resources are limited, then attempt to develop tests for the interaction sequences that subsume the most important tests of the system.

Guideline: If time and resources permit, augment tests with fault-based tests to ensure that illegal user interactions are identified.

Guideline: Identify subsequences of interaction sequences that can be verified independently.

These tests can then be abstracted out as one large state, thereby reducing the overall number of sequences that are required to test the FSM for the application. There are additional strategies that can be used to reduce the number of interaction sequences, but these are not discussed here.

Guideline: When the system is updated, assess the impact of the functionality changes and perform regression testing using existing processes and methods.

After problems identified through testing have been corrected, regression testing helps ensure that the modified parts of the software have not affected previously tested parts adversely. Often, software is not regression tested sufficiently, resulting in costly rework once the software is released and new defects are discovered. Traditional (non-GUI) regression testing can cost as much as one-third of the total cost of software production [Beizer 1990]. GUI regression testing can be much worse because the automated support for testing GUIs is limited.

NOTE: When model-based testing is used, the models can usually be updated more easily, tests can be regenerated, and all generated tests can be reexecuted automatically. This can significantly reduce the cost of regression testing.

Process and Organizational Changes

Many organizations are beginning to realize that they need to treat testing as an engineering activity. It should start early with requirement analysis to better understand the testability of the requirements. When testers interact with stakeholders, such as the end user, requirements engineer, or designer, they get a

better understanding of the requirements that often are ambiguously written, stated, or undocumented. In interpreting the requirements, testers also will recognize requirements that are untestable. Working early in the development process, testers can help stakeholders refine the requirements to ensure that they are testable. In addition, testers interacting with designers can work to ensure that the system is designed for testability and supports test automation. Through test automation, testing can be more systematic, provide greater coverage, and be performed earlier to help eliminate late defect discovery and reduce associated rework.

Summary

This paper identifies a number of guidelines that can help promote a more cost-effective approach to functional GUI testing. It recommends designing for testability to promote the separation of GUI and data processing logic and expose application programming interfaces to support test automation. It recommends design for testability guidelines for the GUI-specific objects. It discusses the need to identify the requirements allocated to the GUI functionality and the need to prioritize and phase the testing of the GUI functionality to reduce the chances of late bug discovery. It also recommends approaches to help better understand the detailed functionality allocated to the GUI. One technique relies on systematic analysis through the identification of interaction sequences, which are detailed scenarios associated with a set of input events to the GUI. By performing systematic analysis of the detailed set of scenarios allocated to the GUI, the assessment of test completeness becomes more objective. It also discusses various types of test automation that can be used to support test design and test execution. Table 4 provides a loosely ordered list of the guidelines strategies discussed in this paper.

Table 4. Strategies for GUI and Web Testing

Guideline	Activity
1. Design the GUI for testability to maximize the controllability, observability, and predictability, by separating the requirements of the GUI from the data processing logic.	Design, requirement allocation
2. Ensure that GUI objects used in the implementation have support for testability.	Design/implementation
3. Prioritize requirements for test planning in three ways: test important functions first, understand how subsumed functions are covered by other functions, plan to perform tests early for features delivered early.	Test planning
4. Decompose or elaborate the implicit or explicit requirements allocated to GUI to support test case design.	Test planning, test design
5. Lightweight strategy: identify scenarios for GUI requirements.	Test design
6. Construct one or more state machine representations and derive the interaction sequences for the required GUI functionality.	Test design
7. Construct interaction sequences for testing unexpected GUI inputs.	Test design
8. For web-based systems, consider the use of static content automation to check for errors in web pages such as broken links, misspellings, and HTML-conformance violations.	Test planning, test execution
9. Select test automation that suits the testability of the GUI implementation as well as the test implementation and test design approaches.	Test planning, test implementation, test execution

10. Assess and estimate test coverage of the requirement and associated implementation to determine when to stop testing.	Test coverage analysis
11. Allocate sufficient resources for planning and performing regression testing.	Regression testing
12. Apply iterative and continuous testing that includes early assessment of testability of the requirements and GUI design.	All

References

[Alspaugh 1992] Alspaugh, T.A., S.R. Faulk, K.H. Britton, R.A. Parker, D.L. Parnas, and J.E. Shore. *Software requirements for the A-7E aircraft*, Tech. Rep. NRL/FR/5546-92-9194. Washington, D.C.: Naval Research Lab, 1992.

[Belli 2003] Belli, F. *Finite-State Testing of Graphical User Interfaces*, 2003. (http://sigma.uni-paderborn.de/download/papers/finitestatetesting.pdf)

[Benedikt 2002] Benedikt, M., J. Freire, and P. Godefroid. *VeriWeb: Automatically Testing Dynamic Web Site*. Bell Laboratories, Lucent Technologies, 2002. http://www2002.org/CDROM/alternate/654/,

[Beizer 1990] Beizer, B. *Software Testing Techniques*, 2d ed. New York, New York: Van Nostrand Reinhold, 1990.

[Busser 2001] Busser, R.D., M.R. Blackburn, and A.M. Nauman. "Automated Model Analysis and Test Generation for Flight Guidance Mode Logic." Digital Avionics System Conference, 2001.

[Chow1978] Chow, T.S. "Testing software design modeled by finite-state machines." *IEEE Transactions on Software Engineering* 4,3(1978):178-87.

[Cockburn 2000] Cockburn, A. *Writing Effective Use Cases.* Reading, Massachusetts: Addison-Wesley, 2000.

[Fewster 1999] Fewster, M., D. Graham. *Software Test Automation: Effective Use of Test Execution Tools.* Boston, Massachusetts: Addison-Wesley, 1999.

[Fujiwara 1991] Fujiwara, S., G. von Bochmann, F. Khendek, M. Amalou, A. Ghedamsi. "Test selection based on finite state models*." IEEE Transactions on Software Engineering* 17(6):591-603, 1991.

[Goeschl 2002] Goeschl, S., and H. M. Sneed. "Case study of testing a distributed internet-system." "Software Testing, *Verification and Reliability* (12, 2 (2002): 77 – 9.

[Heitmeyer 1996] Heitmeyer, C., R. Jeffords, B. Labaw, Automated Consistency Checking of Requirements Specifications. *ACM TOSEM*, 5(3):231-261, 1996.

[Hieatt 2002] Heiatt, E. R. “Going Faster:Testing the Web Application.” *IEEE Software* 19, 2 (March/April 2002).

[Imbus , 1997] Imbus, xx. *Automated Testing Of Graphical User Interfaces (GUIs)*. The Business Benefits of Software Best Practice Case Study: 24306 (IMBUS GmbH), 1997. http://www.esi.es/VASIE/Reports/All/24151/gui-test1.pdf.

[Jia 2002] Jia, X., H. Liu. “Rigorous and Automatic Testing of Web Applications.” In *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications* (SEA 2002). Cambridge, Massachusetts, pp.280-285, 2002.

[Luo 1994] Luo, G., G. von Bochmann, and A. Petrenko. “Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized WP-method.” *IEEE Transactions on Software Engineering* 20, 2(February 1994):149-62.

[Memon 2001] Memon, A. M. “A Comprehensive Framework For Testing Graphical User Interfaces.” Ph.D. diss, University of Pittsburgh, 2001.

[Memon 2001] Memon, A. M., M. E. Pollack, and M. L. Soffa. “Hierarchical GUI Test Case Generation Using Automated Planning.” *IEEE Transactions On Software Engineering* 27 (2001): 144-55.

[Memon 2003] Memon, A. M., and M. L. Soffa. *Regression Testing of GUIs.* ACM SIGSOFT Software Engineering Notes archive Volume 2, 2003.

[Meyer 1999] Meyer, S., and L Apfelbaum. *Use-Cases Are Not Requirements*, 1999. http://www.geocities.com/model_based_testing/notreqts.pdf

[Nyman 2001] Nyman, N. *In Defense of Monkey Testing*, Test Automation SIG Group, 2001. http://www.softtest.org/

[Pettichord 2002] Pettichord B. "**Design for Testability** ." A paper delivered at the Pacific Northwest Software Quality Conference (PNSQC), October 2002. http://www.io.com/~wazmo/papers/design_for_testability_PNSQC.pdf

[Ricca 2001] Ricca, F., and P. Tonella. “Analysis and Testing of Web Applications.” In *Proceedings of the 23rd International Conference on Software Engineering.* Toronto, Ontario, Canada, pp. 25-34, 2001.

[Robinson 1999] Robinson H. *Graph Theory Techniques in Model-Based Testing*. *International Conference on Testing Computer Software*, 1999.

[Valaer 1997] Valaer, L., and R. Babb. “Choosing a User Interface Development Tool.” *IEEE Software* (July 1997).

[Williams 1982] Williams, T. W., and K. P. Parker. "Design for Testability - A Survey." *IEEE*

Trans. Comp. 31 (1982): 2-15.

[White 2000] White, L. and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences", Proc. of Int. Symp. on Software Reliability Eng.-2000, San Jose, CA, Oct. 2000, pp 110-121.

[Wu 2002] Wu, Y., and J. Offutt. *Modeling and Testing Web-based Applications,"* GMU ISE Technical ISE- TR- 02-08. Fairfax, Virginia,2002.

About the Software Productivity Consortium

The Consortium is a nonprofit initiative of approximately 100 U.S. companies, government agencies, and universities, founded in 1985. Consortium members pool their resources to create a community of interest dedicated to advancing the state of practice in systems and software engineering and process improvement. Within these domains, a full-time staff provides reduced-to-practice processes, methods, tools, and services through websites, e-periodicals, technical reports, training, and consulting to meet member needs.

For More Information

Members with general questions or comments on any of the topics in this paper or related topics, or members interested in applying TAF with Consortium assistance, should contact the author or their member account director (see http://www.software.org/pub/keycontacts.asp).

For more on TAF, see the Consortium's TAF website at http://www.software.org/pub/taf/testing.html, or contact the authors.