Design for Testability

Bret Pettichord
bret@pettichord.com
www.pettichord.com

Abstract

The paper provides practical suggestions that will inspire teams to make their software products more testable. It cites examples of testability features that have been used in testing various software products, including verbose output, event logging, assertions, diagnostics, resource monitoring, test points, fault injection hooks and features to support software installation and configuration. It also hopes to change some common prejudices about testability. It argues that all automation requires some instrumentation and that all automated testing must differ, to some degree, from actual customer usage. Wise automators will seek to manage these differences, rather than avoid them. It also provides a detailed exploration of how testability issues affect GUI test automation and what kinds of provisions testers must make when testability is lacking. It argues, therefore, that many of the objections to creating test interfaces are often overstated. The paper concludes with a discussion of how testers can work with product developers to get important testability features built into software products.

Bret Pettichord helps software teams with their testing and test automation. He is co-author of *Lessons Learned in Software Testing*, a columnist for Stickyminds.com, and the editor of TestingHotlist.com. Based in Austin, Texas, he is the founder of Pettichord Consulting and provides consulting and training throughout North America. He is currently developing methods for testing with Extreme Programming and other agile methods.

Design for Testability [*]

Bret Pettichord
bret@pettichord.com
www.pettichord.com

To be presented at the
Pacific Northwest Software Quality Conference, Portland, Oregon, October 2002

1. Introduction

This paper provides practical suggestions that will inspire teams to make their software products more testable. It also hopes to change some common prejudices about testability.

My interest in testability stems from my experience automating software tests for over a decade. Time after time, the success of test automation projects has hinged on testability issues. Automation sometimes has required subtle software changes. In other circumstances, the entire testing strategy depended on interfaces or facilities fortuitously provided by the product architecture. On the other hand, when the software wasn't well adapted for testing, automation was disappointing, forcing the team to depend on manual testing. Other authors have also noted the importance of testability for test automation.

I was slow to realize the importance of testability for test automation (Pettichord 1999). I'd already identified three keys to test automation: (1) a constructive relationship between testers and developers, (2) a commitment by the whole team for successful automation and (3) a chance for testers to engage early in the product development cycle (Pettichord 2000). These turn out to be exactly what is needed to get testability designed into a product. You need (1) cooperation from developers to add testability features, (2) a realization that testability issues blocking automation warrant attention from the whole team, and (3) a chance to uncover these challenges early, when the product is still open for design changes. Indeed, teams often find that it is more efficient to build testability support into a product than to construct the external scaffolding that automation would otherwise require.

Testability goes by different definitions. This paper defines testability as *visibility* and *control*. *Visibility* is our ability to observe the states, outputs, resource usage and other side effects of the software under test. *Control* is our ability to apply inputs to the software under test

[*] I thank the following for comments on early drafts: Noel Nyman, Johanna Rothman, Marge Farrell, Ross Collard, Brian Marick, Sam Guckenheimer, Phil Joung, Melora Svoboda, Jamie Mitchell, Nancy Landau, Dawn Hayes, Ken Pugh and anonymous reviewers provided by PNSQC. This paper benefits from prior presentations at the Test Automation Conferences in San Jose (March 2001) and Boston (August 2001) and from discussions at the 2nd Austin Workshop on Test Automation (January 2001; participants: Alan Jorgensen, Allen Johnson, Al Lowenstein, Barton Layne, Bret Pettichord, Brian Tervo, Harvey Deutsch, Jamie Mitchell, Cem Kaner, Keith Zambelich, Linda Hayes, Noel Nyman, Ross Collard, Sam Guckenheimer and Stan Taylor) and the 6th Los Altos Workshop on Software Testing (February 1999; participants: Ned Young, Jack Falk, James Bach, Jeff Payne, Doug Hoffman, Noel Nyman, Melora Svoboda, Chris Agruss, Brian Marick, Payson Hall, Bret Pettichord, Johanna Rothman, Dave Gelperin, Bob Johnson, Sandra Love, Cem Kaner, Brian Lawrence and III).

or place it in specified states. These are the aspects of testability that are affected by the software design and that have greatest impact on the feasibility of automated testing. Ultimately, testability means having reliable and convenient interfaces to drive the execution and verification of tests. The appendix discusses other definitions of software testability in the literature.

Many teams are frustrated trying to get the testability features they want. Testers don't expect their requests to be met. Programmers are often unaware of what features would help testers the most. This paper concludes by suggesting strategies for bringing testers, programmers and managers together in supporting testability.

A convenient distinction would be to distinguish between testability—test support included in the software product code—and test automation—test support external to the product. It would be convenient, but as this paper will illustrate, it is often a difficult line to draw. Traditional test automation approaches actually rely on hidden testability implants. Monitors may be inserted into either the product code or the operating system; either way they provide testability. Indeed, as you move away from the notion of an independent, arms-length testing team—as this paper encourages—it becomes harder to draw a clear line between automation and testability. This paper argues that close cooperation between testers and programmers is a key for testability. But it also hopes to facilitate that cooperation by providing a catalog of specific features that product teams have used to make software more controllable by or more visible to testing. These include test interfaces that facilitate automation and diagnostic features that facilitate verification and analysis. It's a long list, though not exhaustive. Use it to generate ideas your team can use for your product.

After discussing some basics, this paper visits various software features and modifications that have improved testability. These include logging events, providing diagnostics, monitoring resources, facilitating configuration, and test interfaces. I've been collecting examples of testability features included in software products for some time. These examples come from various sources: conference talks, magazine articles, workshop reports and my own experiences. One goal of this paper is to simply collect and organize these notes in one place.

There's been some controversy regarding the merit of using test interfaces as opposed to testing via the user interface. This paper presents several arguments for seriously considering creating test interfaces. It then provides a detailed discussion of graphical user interface (GUI) test automation and the testability issues that often affect it, and the alternatives that are available when testability isn't available. This discussion should be helpful to GUI automators, but it is also provided to dismiss some of the misconceptions regarding the ease and veracity of automated GUI testing.

2. Visibility Basics

Visibility means the ability to observe the outputs, side effects and internal states of the software under test. It's a behavioral attribute: it refers to the software when it runs dynamically.

But a more fundamental aspect of visibility is simply having access to the source code, design documents and the change records. This is clearly a prerequisite for most testability improvements. The change records are stored in the configuration management system. They indicate what changes were made to what files by whom and when. Conscientious programmers

will attach additional comments indicating the purpose of the change. Many configuration management systems can be set up to automatically send out email when changes are checked in.

Not only do testers need access to this information; they need to know what to make of it. At least some of the testers on your team need to know how to read code and how to interpret the design modeling languages used. Nowadays these often include sequence diagrams, activity diagrams (née flowcharts), class diagrams, and entity relationship diagrams. (Guckenheimer 2000, Adams 2001) Before testers can ask for test interfaces, fault injection hooks or other testability features, they need to have a basic understanding of system design, what interfaces are already available and where testing hooks might be placed. Testability requires collaboration between testers and developers using a common language.

3. Verbose Output

The verbose modes found in many programs are good examples of testability features that allow a view into the software operation. These are especially common with Unix commands and programs.

Consider the Unix mail command. It's a command line program that takes an email address for an argument, and then prompts for a subject line and the body of the message. If you're not using the verbose mode, that's all you'll see. But when you use the "-v" switch with the command, the mail program will display its communication with the local mail server. An example is shown in Figure 1.

This example shows a short email message being sent by wazmo@io.com to bret@pettichord.com. The first four lines show the interaction between the mail client program running on the machine named "eris" and the user known as "wazmo".

The remainder is the verbose output. The local system (eris.io.com) is sending the email to the local mail server, which in turn will take care of delivering it to pettichord.com. The verbose output shows the communication between eris and the local mail server (deliverator.io.com). The lines prefixed by ">>>" are the commands sent by eris to deliverator. The lines prefixed by three-digit numbers are deliverator's replies. (This example is based on DiMaggio 2000)

There are several ways that you might take advantage of this verbose mode. For one, this information could help you if you were testing the mail client. It could also be useful if you were testing a mail server (in this case, an SMTP server). The verbose mode might tip you off that the server is not responding correctly. It could also be used to help debug mail configuration problems. If mail isn't getting through, the verbose mode could help expose where things are going wrong.

Verbose mode also helps you learn about how this communication between mail client and server (i.e. the SMTP protocol) is supposed to work. The communication protocol is actually just text being passed back and forth. That's what the verbose mode is exposing. After using the verbose mode to learn how it usually goes, you might want to use the telnet program to communicate directly with the mail server and see how it handles command variations. This would allow you to test the error handling of the mail server.

Figure 1. Unix Mail's Verbose Mode

```
mail -v bret@pettichord.com
Subject:   testability example
           Sample text.
.
Cc:
bret@pettichord.com... Connecting to mx.io.com. via relay...  220-deliverator.io.com
           ESMTP Sendmail 8.9.3/8.9.3; Fri, 12 Jan 2001 15:34:36 -00 220 Welcome to
           Illuminati Online, Fnord!
>>> EHLO eris.io.com
250-deliverator.io.com Hello IDENT:wazmo@eris.io.com [199.170.88.11], pleased tu
250-8BITMIME
250-SIZE 5000000
250-DSN
250-ONEX
250-ETRN
250-XUSR
250 HELP
>>> MAIL From:      <wazmo@eris.io.com> SIZE=67
250 <wazmo@eris.io.com>... Sender ok
>>> RCPT To:        <bret@pettichord.com>
250 <bret@pettichord.com>... Recipient ok
>>> DATA
354 Enter mail, end with "." on a line by itself
>>> .
250 PAA07752 Message accepted for delivery bret@pettichord.com... Sent (PAA07752
           Message accepted for delivery) Closing connection to mx.io.com.
>>> QUIT
221 deliverator.io.com closing connection
```

4. Logging Events

A verbose mode is one technique for logging software events. Logs help testing by making the software operations easier to understand. They also detect bugs that might be difficult to notice otherwise. When bugs are found, logs can help focus on the faulty code and help with debugging.

However, you can easily log too much. Overburdened with detail, logs may become hard to decipher. And they may get very long—too long to read and possibly taking up too much storage, causing problems themselves. Logging excessive detail can also burden system performance.

Errors are probably the most important things to place in the logs. But one of the purposes of logs is not just to record that things went wrong, but also what happened beforehand. Like a flight recorder, software logs can uncover the events that lead up to the failure. Too often

programmers are unable to fix bugs because they don't know how to replicate them. Good logs contain the information needed to replicate.

To address these concerns, Marick (2000) has developed several heuristics for determining what should be logged and how log storage can be managed. One recommendation is to log user events:

> For purposes of most programmers, a UI event is something that provokes messages to other subsystems. Something that happens solely within the UI layer is not of enough general interest to justify its space in the log. For example, most programmers do not want to know that a particular mouse button had been clicked somewhere in a dialog box, but rather that the changes of the whole dialog had been applied. (Marick 2000: 8)

Other events to consider logging include the following:

- Significant processing in a subsystem.
- Major system milestones. These include the initialization or shutdown of a subsystem or the establishment of a persistent connection.
- Internal interface crossings.
- Internal errors.
- Unusual events. These may be logged as warnings and a sign of trouble if they happen too often.
- Completion of transactions.

A ring buffer is a data structure that is filled sequentially. When the data written reaches the end of the buffer, additional entries overwrite whatever was written at the beginning. Ring buffers are often used to store logs. Because they have a fixed size, you don't have to worry about them filling all available storage. Marick recommends saving a copy of the ring buffer whenever an error occurs.

Log messages should include timestamps and identify the subsystem that produced them.

Using variable levels of logging helps keep logs manageable. When a debug or verbose mode is enabled, more detailed messages are logged. Because of the potential volume, it is often desirable to be able to turn on debugging on a subsystem-by-subsystem level.

When making use of standard components, such as web servers, application servers or databases, you can probably take advantage of the logging mechanisms already built into such components. For these, you often must enable or configure the logging feature. Indeed most operating systems have logging services that may already be logging information you need.

Commercial tools that capture execution history when errors occur provide another approach to logging. These tools, including RootCause by OC Systems and Halo by InCert, work by instrumenting the executables when they are loaded. (Rubenstein 2002)

Once you have some sort of logging in place, how do you best make use of it? When I'm testing a new system, one of the first things I do is review any logs that it generates. I'll try to understand how the logs correlate to the actions I'm executing on the software. I do this to learn how the different parts of the system interact and also to understand what a normal log looks

like. This can then serve as a baseline for later testing, when I'm seeing errors or unusual messages appearing in the logs.

What else can you do with logs? Johnson (2001) and others provide several suggestions.

- Detect internal errors before they appear at the system level.
- Find out how to reproduce bugs.
- Learn about your test coverage. What messages haven't your tests yet triggered?
- Learn about customer usage. What features do they use the most? What kinds of mistakes do they make? Focus your testing on what customers actually do.
- Determine which customer configurations are most common and focus testing on them.
- Uncover usability problems. Are there certain pages that lead to a high level of user abandonment?
- Understand load patterns and use these as a basis of setting realistic load test objectives.

5. Diagnostics, Monitoring, and Fault Injection

Diagnostics are techniques for identifying bugs when they occur. *Monitors* provide access to the internal workings of the code. Event logs are an example of a type of monitor. Diagnostic and monitoring features make it easier to detect bugs when tests encounter them. Some bugs don't immediately lead to obvious failures. Diagnostics and monitoring help uncover them. They also help testers understand software behavior and therefore think of more good tests. *Fault injection* is a technique for testing how software handles unusual situations, such as running out of memory. Effective fault testing often requires testability hooks for simulating faults.

A common diagnostic is an assertion. Code often makes assumptions about the data that it works with. *Assertions* are additional lines of code that make these assumptions explicit. When assertions are violated (i.e. assumptions prove untrue), errors or exceptions are automatically raised. Assertion violations should not normally occur; when they do, they signal a bug. For example, code for computing a square root could include an assertion to check that the input was non-negative. If not, it would abort.

Without assertions you might not notice when a bug occurs. Internal data may now be corrupt, but not result in a noticeable failure until further testing accesses the data. Assertions also help isolate the locations of bugs. Assertion violations typically report the location of the assertion code.

Recent interest has focused on a special kind of assertion, known as a probe. A *probe* is like an ordinary assertion, checking a precondition of a function, except that it handles failures differently. Ordinary assertions typically abort when an assertion fails, usually with a little error message. Probes continue running, placing a warning in a log. Or at least they give you the option of continuing. From a testing perspective, either works for finding bugs.

Classic practice includes assertions in debug code that is tested internally, but not deployed to the customer. However, more and more, developers are including assertions (or probes) in the delivered executables. A common objection to this is that assertions may impair

performance. Although, this could happen, this isn't the best way to address performance. Various studies have shown that even good programmers are poor at predicting which parts of their software take the most time. Performance improvements need to be made with the use of a profiler, a tool that measures the amount of time spent executing different parts of the code. No one should assume that assertions in general will hurt performance. Useful references for how and when to use assertions include Fowler (2000: 267 - 270), Payne et al (1998) and Maguire (1993).

Design by contract is a design technique developed by Bertrand Meyer. In effect, it specifies how to use assertions to define a class interface. A helpful overview is contained in Fowler (2000a: 62-65) and a detailed exposition is contained in Meyer (1997: Ch. 11).

Another strategy for detecting invalid data is to write a program that walks through a database looking for referential integrity violations. It is, of course, possible to configure databases to automatically enforce integrity rules. But this is sometimes avoided because of performance issues. Right or wrong, it opens up the likelihood for data integrity bugs. A diagnostic program that searches for such problems can help isolate these bugs and detect them earlier. Such a diagnostic program may be included with the product, to help users isolate problems in the field. Or it might simply be used as an in house testing tool.

A technique used to help find memory errors in Microsoft Windows NT 4.0 and Windows 2000 is to modify the memory allocation algorithm to put buffers at the end of the allocated heap and then allocate backwards into the heap. This makes it more likely that buffer overrun errors will cross application memory boundaries and trigger a memory error from the operating system. A similar technique can be used with the recent Visual C++ .NET compiler. Its /GS switch adds a code to the end of each buffer allocated by your application. Any code that overwrites the buffer will also overwrite the /GS code, resulting in a new stack layout that can be easily detected (Bray 2002).

Effective methods for finding memory leaks require monitoring memory usage. Otherwise you won't notice them until you run out. Some applications have embedded memory monitors that give a more detailed view of the application memory usage (Kaner 2000). In other cases, teams have used commercial tools such as Purify or BoundsChecker.

Testing is often easier when you can monitor internal memory settings for inspection. A good example of this can be seen by entering "about:config" in Netscape. It will dump out all the configuration settings. It's a great tool for tracking down the source of configuration problems. (I used it while writing this paper.) This kind of output is particularly helpful for problems that only occur on particular machines.

Applications running on Windows 2000 or XP can use a special test monitor called AppVerifier. While you are testing your application, AppVerifier is monitoring the software for a number of targeted errors, including: heap corruptions and overruns, lock usage errors, invalid handle usage, insufficient initial stack allocation, hard-coded file paths, and inappropriate or dangerous registry calls. The AppVerifier includes monitoring, fault injection and heuristics for error detection (Phillips 2002, Nyman 2002.

Sometimes testers need access to internal data. Test points are a testability feature that allows data to be examined and/or inserted at various points in a system. As such they allow for

monitoring as well as fault injection. They're particularly useful for dataflow applications (Cohen 2000). Mullins (2000) describes a product that could have benefited from test points. He tested a system composed of multiple components communicating over a bus. One of the components was sending bad data over the bus. This went unnoticed because other components screened out bad inputs. All but one, that is. The faulty data lead to a failure in the field when this component failed due to a particular bad input. Observing the data stream during testing through test points would have detected this fault before it resulted in a component failure.

Fault injection features assist with testing error-handling code. In many cases, environmental errors are exceptional conditions that are difficult to replicate in the lab, especially in a repeated and predictable way (which is necessary for automated testing). Example errors include disk-full errors, bad-media errors or loss of network connectivity. One technique is to add hooks for injecting these faults and triggering the product's error-handling code. These hooks simulate the errors and facilitate testing how the program reacts. For example, in testing a tape back-up product, it was important to check how it handled bad media. Crumpling up sections of back-up tape was one way of testing this. But we also had a hook for indicating that the software should act *as if* it found bad media on a specified sector of the tape. The low-level tape-reading code acted on the hook, allowing us to test the higher-level code and ensure there was no data loss in the process. (It would skip the bad section of tape, or ask for another.) Houlihan (2002) reports using a similar technique in testing distributed file system software.

Another technique for fault injection is to use a tool, such as HEAT or Holodeck, that acts as an intermediary between the application and the operating system. Because of its position, it can be configured to starve the application of various system services. It can spoof the application into acting as if the system were out of memory, the disk were full…remove network connectivity, available file system storage and the like. (HEAT and Holodeck are included in a CD accompanying Whittaker 2003.). Note that fault injection is a testing technique that depends on either adding code to your product (testability) or using an external tool (automation). If you don't have a tool for the faults you want to inject, you may find it easier to build in the necessary support than build an external tool.

6. Installation and Configuration

This paper reviews several examples of how software can be made easier to test: sometimes by building and customizing external tools; sometimes by building support for testing into the product. Sometimes it's a blurry distinction, with external code later incorporated into the product. The same holds for testing software installation and configuration. Testers often spend a significant amount of time installing software. Tools and features that simplify installation often provide effective means for automating testing.

Making the installation process scriptable can yield significant advantages. Install scripts can automatically install the software on multiple machines with different configurations. Many software products use InstallShield to create their installers. InstallShield installers feature a built-in recorder that stores installation options in a *response* file. Testers can use response files to drive unattended installations. These response files have an easy-to-read format, making them easy to modify with different location paths or installation options. (InstallShield 1998). Microsoft Windows also comes with support for scripting unattended installations. These

scriptable interfaces amount to test APIs for the installers. They are useful regardless of the specific installer technology used.

Installers should reliably and clearly report whether any errors occurred during installation. Some products have installers that generate detailed installation logs but no single indicator of success, suggesting that users search through the log to see if it lists any errors. Of course testers must check this as well. While I was testing such a project, other testers wrote scripts to parse through the logs, collecting lines that began with the word "error". I worried that we might miss errors that were labeled differently in the log. The product used multiple installation modules; it was risky to assume they were all using the same error reporting conventions. Far better would have been to expect the product to define its own mechanism for determining whether an error had occurred. Including this logic in the product (rather than in the test automation code) would likely result in a more reliable design, would lead to it being tested more thoroughly and seriously, and would also provide value to the users.

Many software products store persistent data in a database. Testers need to be able to control and reset this data. Whether reproducing bugs or running regression tests, testers must be able to revert the database to a known state. Realistic testing often requires preloading a test bed into the database. These testing requirements define the need for products to support configuring and initializing test databases and automatically loading test data in them. Even products that don't use databases can benefit from routines to automatically pre-load data.

Another feature that facilitates testing is the ability to install multiple versions of a software product on a single machine. This helps testers compare versions, isolating when a bug was introduced. Networked application testing is particularly aided by the ability to install multiple product instances on a single machine. This aids testing deployment configurations and products scalability by allowing the full use of the available lab resources. Typically this requires configurable communication ports and provisions for avoiding collisions over resources like the registry. Because installed products are nowadays more likely to wired into the operating system, supporting multiple installations is sometimes difficult. Another approach is to use products like vmWare that allow multiple virtual machines to run on a single physical machine.

7. Test Interfaces

Software testing occurs through an interface. It might be a graphical user interface (GUI). It might be an application programming interface (API), a component interface (e.g. COM, ActiveX, or J2EE), a protocol interface (e.g. XMI, HTTP, SOAP), or a private debugging or control interface. Some interfaces are meant for humans, others for programs.

Manual testing is often easiest using a human interface. Automated testing, as this paper argues, is often easiest using programming interfaces. In particular, programming interfaces to core business logic and functionality are significant testability features, providing control and visibility.

The likelihood and willingness of development teams to provide test interfaces varies considerably. Some see it as a necessity; others don't see the point. Some will even create them without notifying testers, using them for their own debugging and integration testing. Programming interfaces often support automated testing earlier and with less effort than required when testing via a GUI. Because of this, I've often recommended the use and development of

programming interfaces for testing. This is admittedly a controversial suggestion. Some testers have said that programming interfaces they've had access to were less stable than the user interfaces and therefore less suitable for testing (see Szymkowiak's comments in Kaner et al, 2001: p 119, note 5). Clearly test interfaces need to be stable or else tests that use them will be worthless.

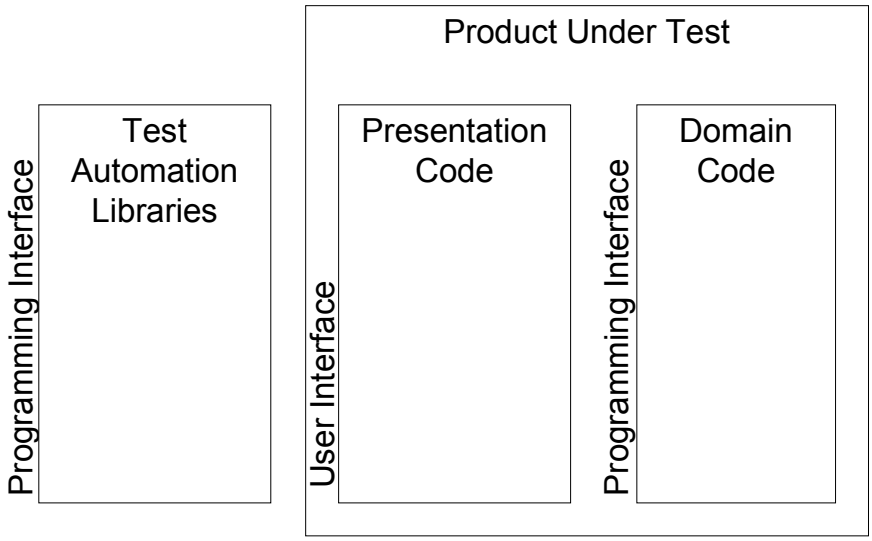
Unconfirmed reports indicate that the original version of Excel included a test interface. Because mathematical accuracy is a key requirement for spreadsheet software, the ability to run lots of automated tests frequently was essential. The test interface that was provided to Excel was later exposed to users and is now accessible through its Visual Basic interface. Thus did a testability feature eventually develop into a popular customer feature.

The value of test interfaces has even been clear to public-domain game developers. Xconq is a public-domain game. It was developed with a game engine and separate interfaces for different platforms. One of its interfaces is a command-line interface specifically provided to facilitate testing (Xconq). If public-domain games can have their own test interfaces, then commercial business software should be able to get them as well.

Many business software products have included customer-focused APIs which testers have made use of. Examples include AutoCAD, which has interfaces in several languages, Interleaf, which includes a Lisp interface, and Tivoli, which has a command-line interface as well as CORBA bindings. In these cases, testers exercised significant business functionality through the APIs.

Some have questioned the idea of testing via a different interface from what customers will use. They question the *veracity* of the tests: will these tests find the same bugs that users can be expected to encounter?

Figure 2. Programming Interfaces Available for Testing



A software product can often been divided into the core business logic or key functional—we'll call this the domain code—and the code that interacts with the users—we'll call this the presentation code. Sometimes the boundary between these two layers is clearly defined. In other cases it's rather murky. Ideally, the presentation code calls a clearly defined programming interface to the domain code. As Figure 2 makes clear, if you test via the internal programming interface, you won't be covering the presentation code.

GUI test automation requires a means to describe the tests. At some level, this amounts to an interface that allows testers to describe what buttons to press and what text to type. Automated test scripts are programs, and the language they use to specify the tests amounts to a programming interface. This programming interface is supported by the test tool, its libraries, and customization code that the automators may have provided. This automation support code often also includes maintainability support (to make it easier to revise the tests when the inevitable changes are made to the user interface) (Kaner 1998).

The figure illustrates that testing via the test automation libraries (an external programming interface) covers the presentation code. It also "covers" the automation libraries themselves. This isn't code you care to have covered; problems in it can lead to false alarms. *False alarms* are tests that fail because of a problem with the test apparatus or environment, and not a problem with the software under test. (Some people call these *false positives*, but I find this term confusing, as they feel rather negative.)

Products that will be tested via an internal programming interface are sometimes designed with a thin GUI. This is a minimal layer of presentation code. This approach also requires assurances that that presentation layer calls the same internal interface as the automated tests. (Fowler 2001)

My view is that either approach—automated GUI testing or API testing—exercises the product somewhat differently from the way an actual user would. Because of these differences, and because potential errors in either test approach could cause bugs to go undetected, either approach should be supplemented with manual testing. Both are valid strategies. The API approach, however, is often simpler and more productive. Concerns regarding its effectiveness are often over-stated.

8. User Interface Testability

GUI test automators know that testability issues often bedevil automation. This section provides an overview of GUI test automation and the testability issues that automators often encounter.

This paper devotes considerable space to these issues in order to demonstrate two points. One claimed benefit of GUI automation is that it tests the way a user does. Obviously this isn't true for API-based testing (of products that are primarily used through a GUI). However, automated GUI testing also often requires deviations from actual user behavior. This does not indict GUI automation; rather it demonstrates that any automation strategy—GUI or not—must contend with issues of veracity.

The second point is that GUI automation, like API testing, often requires testability modifications. Some think that mixing test code with product code undermines the integrity of the testing. I don't believe this. In any case, this paper will demonstrate that GUI test automation

is only possible with some amount of instrumentation. Sometimes it happens rather unobtrusively. In other cases the demands are fairly large. Once you accept the necessity of changing the product to support testing, I hope you will consider the many testability ideas cataloged in this paper.

Modern graphical user interfaces are composed of windows that contain multiple controls. Windows, also known as dialogs, typically can be moved or resized and have title bars across their tops. Controls, also known as widgets, present information and accept inputs. Common controls include text fields, push buttons, menus, and list boxes. These interface elements are common whether the product runs on Microsoft Windows, Macintosh, Linux or Unix.

Web-based applications have similar elements. The various pages of a web application correspond to windows. Pages allow all the standard controls to be embedded in them. Some web applications use Java or ActiveX to give them the feel of standard user interfaces.

GUI test tools are scripting frameworks with GUI drivers. Their GUI drivers allow them to identify controls, record user events affecting controls, and trigger events targeted at specified controls. User events include entering text in a text field, clicking a button or selecting an item from a menu.

Product developers use GUI toolkits and libraries to build product interfaces. These contain the code that creates the windows and controls in the interface. Some will use the libraries that are provided with the operating system or browser libraries. Others will use libraries provided with their development tools or obtained from outside vendors. Still others build their own.

A common problem that GUI test automators face is custom controls. *Custom controls* are controls that are not recognized by a given GUI test tool. The following sections outline several strategies that automators have used to make custom controls testable. Sometimes this requires tool customization. Sometimes it requires testability changes to the code.

Some examples of custom controls that have been reported by students in my seminars include:

- Grids, or tables, with other controls embedded in them, such as drop-down list boxes.
- Tree views, also known as explorer views. These are lists that allow some items to be expanded to show more items “under” them.
- Lists in Delphi, a development environment.
- 3270 emulators used to access mainframe applications.
- Powerbuilder, another development environment.
- Icons that appear in an interface.
- List boxes with specialized content, e.g. graphics.
- Flow graphs used to display dependencies between application objects.

The process of assessing and ensuring testability of your product with a given GUI test tool is thus:

1. Test compatibility of your test tool and development tools early.
2. Define standards for naming user interface elements.
3. Look for custom controls. If there are any, plan to provide testability support.
    a. Ensure that the tool can recognize the control as an object with a name, class and location.
    b. Ensure that the name used by the tool to identify the control is unique. It won't do if multiple windows or controls are given the same identifiers.
    c. Ensure that the tool can verify the contents of the control. Can it access the text of a text field? Can it determine whether a check box is checked? (This step doesn't apply to some controls, like push buttons, that don't have contents.)
    d. Ensure that the tool can operate the control. Can it click a push button? Can it select the desired item from a menu? (This step doesn't apply to some controls, like static text fields, that can't be operated by the user.)

The following sections provide a detailed examination of these processes. Along they way this paper highlights several testability issues faced by GUI test automation:

- A lack of standards makes GUI testability often a matter of trial and error.
- GUI control recognition technology requires instrumentation. The difficulty of doing this varies.
- GUI test automation requires programmers to adhere to naming standards.
- Custom controls represent a major challenge for test automation. Many techniques are available, but it's often a matter of trial and error to determine which will work. Successful techniques often reduce veracity or require calls to testability interfaces.

9. User Interface Standards and Compatibility

The lack of standard testing hooks into user interface technologies has often made compatibility a major issue for GUI test automation. This is one reason why test tools must often be hand customized, and why GUI test automation must often resort to workarounds that reduce veracity.

When selecting a GUI test tool, you must ensure its compatibility with the tools and technology you're using to develop your GUIs. Just reading product literature isn't enough: claims are usually vague, incomplete or out-of-date. You have to actually try the tools out to verify whether a particular test tool will be able to execute and verify the interfaces created by the development tool. Occasionally teams use development technology for which no compatible test technology exists. They end up having to build extensive test tool customizations or else forgo automated testing entirely.

Standards for user interface testability have been slow to develop. Indeed, interface development technology vendors have been slow to acknowledge the importance of testability. 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 & Babb, 1997).

Even when they have an interest, programmers often have a hard time ensuring their user interfaces will be compatible with specific tools. All GUI test tools make assumptions of how the windows and controls in the interface will be structured. Their GUI drivers expect to be able to recognize and operate controls using a set of standard protocols or messages. Programmers who want to make compatible controls need to know these assumptions but may have a hard time getting this information from test tool vendors. Some vendors claim this information is proprietary and refuse to disclose it. Others only release it contingent to non-disclosure agreements. It may take several calls just to find out what a vendor's policy is.

For several years I was hopeful that some standardization and disclosure would reduce the compatibility problems plaguing GUI test automation. But it seems that the test and development tool vendors lack the motivation to work together. There has been some improvement, however. For one, there has been some consolidation in user interface technologies. With fewer choices, it's easier for tool vendors to cover them. Secondly, the development of standard component technologies (J2EE, .NET, COM, VB) allows tool vendors to provide the glue needed to make their tools easier to customize. The availability of reflection has been particularly helpful. But a lack of a standard for method signatures of GUI controls ensures that automators will have to do some hand stitching when using custom and new technologies.

Another problem is that as standard computer graphical interface technologies mature and are made more testable, more and more applications are moving to thinner client technologies: they are using web browsers, often on handheld computers or cell phones which again lack the testability support for smooth automated testing.

Thus gaps remain. Testers continue to encounter product interfaces and controls that require specific customization before the GUI test tools will operate. Handling these issues often calls on significant skill and time from test automation experts (Hayes 2000, Zambelich 2001).

10. Recognizing User Interface Controls

GUI test tools must instrument the system in order for them to be able to recognize user interface controls. Sometimes this instrumentation is to the software application; other times it is to the operating system. Sometimes it is automation; other times it must be applied manually. Regardless, GUI test automation requires testability instrumentation.

Automation experts use a number of techniques to provide GUI testability. Many techniques are specific to a particular test tool and interface technology, whether it be using WinRunner with Netscape, SilkTest with Java Swing, or Robot with Visual Basic. This paper, however, can't address these issues in detail. Instead it describes some general principles that have been used in multiple cases.

These principles are based on my several years experience as a QA Partner and SilkTest tool expert, tips shared with various colleagues, as well as Lesuer (2001), Zambelich (2001), Segue (1998) and Mercury (2000). Discussing techniques for supporting custom controls is always popular at testing conferences such as STAR and on forums such as QAForums.com.

Simply getting the test tool to recognize the various window controls, including the text fields, list boxes, check boxes and menu lists that constitute modern user interfaces, presents a challenge. There are an endless variety of window controls, sometimes known as widgets or simply "objects". Test tools require some kind of instrumentation in order for them to identify the controls as such—and not as simply patterns on a bitmap.

Without being able to recognize the controls as controls, the tools must resort to dealing with the interface in purely analog terms: sending mouse and key events to locations on the screen and reading and trying to interpret bitmap patterns. This was all the first generation GUI test tools were capable of in the early 1990's. This analog approach was unsuccessful. There were a number of problems, but the most serious was that they were unable to know whether they were actually working with the desired controls.

Imagine a test in which a dialog box appears and the OK button needs to be pressed to dismiss it and continue. Analog methods would wait a "suitable" amount of time for the dialog to appear and then click on the location where the OK button was expected. If the dialog took longer than expected to appear, or if it appeared in a different location, the test script would fail. These kinds of problems were often intermittent and hard to diagnose. As a general label, such problems were known as "windows synchronization" problems.

A new generation of GUI test tools came along that were able to recognize individual controls. They could "see" the OK button as a push button (or "command button") with the label "OK". The tools could now automatically wait for such an object to appear and then click on it. It didn't matter *where* it appeared. If the button didn't appear as expected, an error to that effect would be logged, greatly increasing diagnosability.

In order to see the controls, they needed internal access to the window manager and event queue, and therefore some kind of instrumentation. Initially, this took the form of modified window managers or code that had to be inserted into the main event loop of the application code. Such intrusions weren't popular; with time, the tools became better at inserting the instrumentation automatically. Nowadays this typically takes the form of replacing system DLLs (dynamically linked libraries) or modifying hooks in the operating system.

Automatic instrumentation isn't always successful. When it fails, a test tool expert often has to understand the instrumentation need and then apply it by hand. For example, with a Java Swing GUI running on Windows, the operating system-level test instrumentation will only allow the tool to see the outer Java window. It won't be able to "see" the contents unless it also has instrumentation in the JVM (java virtual machine).

Exceptions to the need for instrumentation are "non-intrusive" test tools such as TestQuest, which connect directly at the hardware level. These can't have direct access to the software objects presenting the window controls and therefore must depend solely on analog methods. They require special tuning and customization in order to be able to view and operate interface controls. (Of course, these tools *are* intrusive at a hardware level.)

Thus the first step in GUI testability is simply getting the tool to see the individual controls.

11. Naming Windows and Interface Controls

Effective GUI test automation requires the ability to name and distinguish windows and controls. Sometimes this requires programmers to add suitable identifiers or conform to naming standards. Some development tools make this difficult.

If your product uses a standard, uncustomized set of interface controls that your GUI test tool supports, you may be all set. But you may still see problems with the names or attributes that the test tool uses to identify the controls. In order for automated tests to be useful, they need to be able to identify different controls with different names. And they need to be able to use the same names through multiple builds of the product. For example, a common problem is that different windows will have the same labels. It's not uncommon for early product versions to use generic labels for all windows. This will lead the test tool will see them all as the same. This makes it hard for the tool to "learn" all the controls in the interface; tests won't be able to tell if the desired window is displayed or if it's a different window with the same name. The programmers need to provide unique names for different pages to avoid these complications.

Naming problems may also affect controls. Test tools may have a hard time associating the text labels with the controls they are adjacent to. They typically look for identifying human-readable text to the left of controls, but I have seen instances where the window controls weren't laid out straight. Since the text label was slightly lower than the text field, the tool did not associate it with the control. These kinds of issues may require adjusting the layout to the tool's assumptions. Sometimes it requires tuning the test tool to understand the fonts being used in the interface.

Often using internal control names instead of the human-language labels avoids the problem. Programmers may need to provide these. Using internal names is particularly helpful when the labels are later re-phrased or when translated interfaces need to be tested. Some GUI development tools automatically assign internal names, which can be problematic if they assign new names every time the system regenerates a window.

When testers can use neither external nor internal names to identify the controls, then they must use other techniques. They may identify controls by their ordinal position (e.g. the third pushbutton), their relative position (e.g. the pushbutton 40 pixels across and 125 down) or by use of other properties of the control. These techniques, however, typically result in automated tests that are brittle and hard to maintain and debug. Thus a key issue with GUI testability is being able to name and distinguish windows and controls using stable identifiers.

12. Mapping Custom Interface Controls

Once you have a name for a control, you also have to be able to operate it. If the control is part of a standard toolkit, support for the control is likely already built into the test tool. No further testability support is needed.

A custom control requires additional support before it can be accessed from test scripts. You may need to customize the tool or modify the control's code. Often the best solution

includes some of each. Whether a particular control is "custom" may depend on which test tool you are using. One test tool may intrinsically support controls that another treats as custom.

Sometimes a control is "custom" simply because the tool can't recognize the control type. For example, a push button may look like an ordinary push button to you and me, but not to the test tool. The control object is different enough to throw off recognition. In many cases, once you configure the tool to recognize the control type, its normal support for that control will work.

Tools define control types as "classes". They support each class with various methods for operating the controls: a method for pushing buttons, a method for typing text into text fields, a method for scrolling list boxes. You can instruct the tool to map otherwise unrecognized classes to standard classes (with their methods). This *class mapping* can use class names or other properties of the controls to establish the association. For example, your test tool might not recognize a push button in your interface, instead seeing it as a custom control labeled a "thunder button." Customizing your tool's class map with an indication that a thunder button should be treated as a push button will allow it to support this control. (Segue 1998)

Class mapping is necessary with certain software applications created with MFC (Microsoft Foundation Classes). These applications generate the class names for their controls at run time (rather than compile time); consequently the names change from run to run. These class names appear with an "Afx" prefix ("application frameworks"). Handling them requires specialized class maps. (Nyman 2001)

Testers also use the class mapping technique to instruct a test tool to ignore internal container classes. Programmers and development tools use these structures, invisible to users, to organize window objects. Ignoring them makes test scripts easier to understand. But many custom controls require more work than simply class mapping.

13. Supporting Custom Interface Controls

When it works, class mapping is the simplest way to support custom controls. This section describes other methods that also do not require modifications to the product code.

Are these testability suggestions? Without some sort of support for custom controls, they can't be tested automatically. The ideas in this section don't require changes to the product code (or the control code) but they do make the product more testable.

As discussed above, first you must ensure recognition (can the tool "see" the control?) and identification (can it uniquely identify it?). Then, you must address operation (can it manipulate it?) and verification (can it check its value?)

To develop the necessary testability support, list the actions your tests need to make with the control and the information they'll need to verify from it. The existing tool support for similar controls often serves as a useful guide. Generally you should encapsulate whatever tricks or techniques you make use of within library methods or functions. Thus they won't complicate the test scripts themselves. This also makes maintenance easier should you later find a better approach. Basing the interfaces on similar classes improves consistency and understandability. (This exposition follows Lesuer 2001.)

Finding tricks and techniques that work often involves trial and error. Suggestions include:

- Key events
- Initializing controls
- Mouse events
- Character recognition
- Clipboard access
- Custom windows
- External access
- Internal interfaces

These techniques are explained in this and the following section.

Key Events. A technique that is elegant, when it works, is simply to use key events. For example, you may find that the HOME key always selects the first item in a custom list box and that the DOWN key then moves the selection down one. Knowing this makes it a simple matter to write a function, say CustomListSelect (int i), that emits the correct key events for selecting the desired list item. Many controls respond to various keys. Experiment with CONTROL, ALT, DEL, HOME, END, SHIFT, SPACE and all the arrow keys.

Initializing Controls. In the list box example, the HOME key explicitly puts the control into a known state (first item selected). This is important. If you assume that a control starts in a particular state, you are likely to introduce bugs into your testability support.

Consider a custom text box. If you assume that it always starts out empty, your test support will work for tests that create new records, but may fail when your tests edit existing records. Your test code for entering text probably needs to clear the box before entering new text.

Mouse Events. Mouse events can also be used. Suppose you have a row of buttons that appear to your tool as a single object. You may be able to compute the location of the individual buttons and send mouse-click events to those locations. (A mouse click is actually composed of a mouse-down event followed immediately by a mouse-down event.)

These techniques provide methods for operating controls. You'll also need to observe the state or contents of controls in order to verify their contents.

Character Recognition. Some tools have character recognition technology built into them. This is the same technology that is used in OCR (optical character recognition) except that it works directly with the digital image. It's an analog technique that is very useful for text-based controls. You can use third-party OCR tools when this ability isn't built into the tool you're using. (Structu Rise)

Clipboard Access. Another clever trick is to copy text from the control to the clipboard, which most tools can then read directly (if not they could always paste it into notepad or another application that they *could* observe directly). I've used this technique, using keyboard commands to copy the text from the control. This technique has an unintended side effect, however. You're effectively using the clipboard as a temporary storage buffer. Careful automators will save the original clipboard contents and restore them afterwards. Otherwise, you are likely to stumble into

false alarms when you test the copy/paste features of your software product in conjunction with your custom control.

Custom Windows. Another problem for testability arises from windows missing standard controls. Most windows have minimize, maximize and close buttons, but developers sometimes choose to remove them or disable them. This effectively makes such windows custom. The standard method your test tool uses for closing the window may not work. It's rarely hard to create a custom close method (e.g. ALT-F4 usually works), but this can complicate testing.

These custom windows cause problems for error recovery systems. Many test suites use error recovery mechanisms to reset applications after failed tests. They are essential if you are to avoid cascading failures. *Cascading failures* occur when the failure of one test in a test suite triggers the failure of successive tests. Error recovery mechanisms depend on heuristics for detecting and closing extraneous windows and dialogs. These heuristics will themselves require customization if developers don't conform to the tool's expectations regarding window standards. Until you devise suitable heuristics, your test suites may not run reliably. It's simpler all around just to stick with standard windows.

External Access. When you can't get the control's contents directly, you may find that you have to resort to other sources for the information. You may have to get the information from a different type of control on another screen, or directly access a file or database that stores the information. Expediency often requires such workarounds.

When these techniques fail, you will need to make use of internal interfaces to the controls. Indeed the use of internal interfaces often provides more reliable support for testing.

14. Accessing Internal Interfaces to Custom Controls

Another set of techniques involves making calls to internal interfaces of custom controls. There are generally two sources of custom controls: third-party vendors or in-house developers. With in-house custom controls, your team may need to specify and provide the internal API's, or interfaces, you need for testing. This usually means adding or exposing methods that allow the control to report its content, state, and location. With third-party controls, you don't have this option, but they tend to be delivered with a much richer set of API's than in-house controls. You may have to be crafty, however, to take full advantage of them.

The API mechanisms vary depending on the type of interface technology used. Many Microsoft Windows-based controls (i.e. Win32 controls) will respond to window messages sent to the control's handle. Other controls can be accessed through DLL (dynamically linked library) interfaces. Most Java, COM and ActiveX controls can be accessed by their public methods. Accessing these interfaces challenges many testers. Each test tool provides different levels of support for the different API mechanisms. You often need a test tool expert to implement this support.

Many control APIs include methods for operating the control and changing its state. Many testers fear that directly manipulating the control using these internal interface hooks will lead tests to take a different path than manual testing would, potentially avoiding product bugs. A strategy that addresses this concern is to use internal methods exclusively for obtaining state and location information about the control. Use this information to figure out where events

should be directed using the normal event queue (Lesuer 2001). Even so, automated support for controls often uses different event sequences than manual users would. Because automated testing can never completely replicate manual usage, wise testers will always supplement automated tests with some manual testing.

Accessing the state of a control allows you to both verify the contents of the control (e.g. that a text box contains the correct text or that a list contains the correct items) as well as learn how to operate it safely. Consider a custom tree view control. Tree views contain hierarchical lists of items. Items with others "under" them are referred to as parents of the children. A tree view could be used to display files in a file system, descendents of George Washington or the organization of genus and species within a phylum. Let's suppose that we're presenting genus and species information in a custom tree view. One of the items in this tree view is an octopus: species Octopoda, genus Cephalopoda, phyla Mullusca. Let's examine how we might build support for selecting this item in the tree view.

Let's also suppose that each item in the list has a text label and that duplicate items with the same name under the same parent aren't allowed. Tests of applications with this tree view would need a function for selecting items in the list by name. This function would need to scroll through the list, expanding items as necessary—just as a user would. A call to this function might look like this:

```
TreeViewSelect (PhylaTree, "Mollusca/Cephalopoda/Octopoda")
```

It would also need to signal an error if the requested item didn't appear in the list.

How would we implement such a function? Our strategy would depend on what we had to work with. Let's suppose we can map this control to a list box class. The list box select method would allow selecting by ordinal position, scrolling as necessary. But it wouldn't know how to expand items.

Further, suppose the control's API provides access to the text label, the indentation level and an expansion flag for each item in the tree view. The indentation level indicates how many parents are above the node. The expansion flag indicates whether the node is expanded or not. The API indexes this information in terms of the internal representation of the tree view, which corresponds to the order in the displayed tree view when all items are fully expanded.

To make use of this API, we'd first need functions to map between the indexes of the currently displayed tree view and the fully expanded tree view. The API provides the information needed to define these mappings; the algorithms are left as an exercise for the reader.

With these mapping functions, we're now ready to write the tree view select function. It must select by name (e.g. "Mollusca/Cephalopoda/Octopoda") expanding nodes as necessary. Our function must first parse its argument into nodes (three, in our example).

Then it must check to see if each node (save the last) is expanded. Obtain this information from the API. Place the expansion logic in its own function. A call might look like this:

```
TreeViewExpand (PhylaTree, "Mollusca")
```


Let's suppose that we use the strategy of sending a mouse click to the expansion hot spot. Our function needs to compute the location of the hot spot. This function needs to be able to obtain the following information from the internal interface: the absolute position of the control (from the built-in control support), the relative vertical position of the item in the list (from the list box support), and the horizontal position of the hot spot. The latter can be computed from the indentation level in the API, using the index mapping function and some observations of the geometry of the hot spots. We'll need the geometric information to be expressed in pixels. We'll also need to ensure that the list item is displayed; long tree views may require us to scroll before the targeted item is visible.

Another workable strategy would be to simply expand the entire tree view first off. This would be simpler to implement, if the tree view has an expand-all command. This strategy would also veer testing further away from normal customer usage and behavior patterns. This illustrates a general principle: automated testing is easier when you're less concerned with complete veracity. This doesn't indict automated testing, but it does require a testing strategy that uses other techniques for addressing the deviations that are often a practical necessity. But I digress.

After using TreeViewExpand to expand the parent nodes, our selection function must select the final node. We can find its index in the internal representation using the API. Our mapping function maps this to the index in the tree view display. And we can use this to select it using the list box select method. This completes our TreeViewSelect function. If we have an API reporting the text of the currently selected item, we might add a check of this as a post condition in our function. This would help catch potential errors in our testability support.

Clearly testability functions like these require unit tests, both positive and negative. This necessitates building a test application incorporating the custom controls and sample data.

15. Conclusion: Knowing Testability Options

This paper's long digression into GUI test automation testability issues underlines the following conclusions:

Automation requires testability. Whether testing via an API or a GUI, some accommodations to testability are necessary for successful automation. If you're going to have to do it, why not plan for it upfront?

Successful automation requires a focus on testability. Again, regardless of approach, the greater the emphasis on testability within the development team, the more likely automated testing will successful.

Programming interfaces provide more productive support for automated testing than user interfaces. This is an arguable point, and is surely not applicable in all situations. But in many situations, the instability of the user interface and the difficulties of automating GUI technology indicate that test automation will be productive when they use programming interfaces.

It's ultimately hard to draw a clear and useful boundary between testability and automation, where testability is internal test support code and automation is external support

code. Any form of automation changes the system to some degree, whether these changes are considered to be inside the product or external to it are often a matter of interpretation. With time, techniques for making these changes at are more external level are developed.

Test automation always affects test veracity to some degree. Automating tests always makes them somewhat different from actual customer usage. Wise testers will understand these deviations and develop compensating testing strategies, rather than act as though it can be eliminated.

This paper has largely been directed to testers and programmers working in traditional software development methodologies. As the deadline for completing this paper approached, I attended the XP Agile Universe conference (Chicago, August 2002). I came knowing that agile development methodologies—and especially Extreme Programming—already incorporate the three keys to test automation: (1) testers and programmers are expected to work together as a single team, (2) the team treats test automation as a team responsibility and (3) testers are expected to be engaged and testing throughout, often writing tests before code's been written. These tests, called acceptance tests, test the system from a customer perspective. The tests themselves are variously written by product programmers, dedicated testers, or analysts acting in the customer role. They use the testing framework created by the product programmers to execute the tests. This is a significant difference from the traditional arrangement that assigns the responsibility for test automation to an independent testing team.

The range of approaches used was impressive. I was surprised to learn that in case after case agile development teams had evaluated and then rejected commercial GUI test tools. Many found their scripting languages weak, awkward or obscure. ("Heinous" was how one put programmer put it.) Working in a methodology that required everyone to be running tests all the time, they also found it cost-prohibitive to purchase expensive licenses ($3,000-5,000) for everyone on the team and unacceptable to only allow tests to be run on a few machines.

These reports have only hardened my impression that regardless of development methodology, testability is the door to successful test automation and that the keys to that door are cooperation, commitment and early engagement.

16. Conclusion: Making Testability Happen

Some teams think that testing is best when it is as independent from development as possible (ISEB 1999). Such teams may get little benefit from this paper. This paper collects examples of testability features that testers and programmers can use to stimulate their own thinking. It presumes they collaborate, develop trust (but verify), and understand the challenges each role faces.

Testers must realize that testability requires them to understand the software design, review existing design documents and read code. This provides the basis for suggestions and facilitates concrete discussions. They need to focus on finding bugs that are important to programmers, managers and/or customers. Focusing on minor problems and process issues will undermine the trust they need.

Testers are often reluctant to ask for testability out of fear that their requests will be rejected (Hayes, 2000). Results vary, but I've been surprised at the number of times that requests

for testability features have elicited replies that something similar is already in place or in planning for other reasons. Testability is definitely worth asking for.

Programmers must realize that a tough tester is a good tester. Building software entails making assumptions. Good testers test these assumptions. Programmers must be frank with concerns about their software. And they need to realize that no amount of explanation can alleviate many testing concerns. Sometimes nothing but a test will do.

Programmers must share their source code, design documents, error catalogs and other technical artifacts. They also need to be ready to provide reasonable explanations of product design and architecture.

Testability is a design issue and needs to be addressed with the design of the rest of the system. Projects that defer testing to the later stages of a project will find their programmers unwilling to consider testability changes by the time testers actually roll onto the project and start making suggestions.

Some teams are interested in testability but feel like their management won't support investing time in it. In fact testability often saves time by speeding testing and making bugs easier to track down. More and more managers are also realizing that effective testing helps communicate the actual state of a project, making late surprises less likely.

When discussing testability features, testers should focus on features: things the software could do to make it easier to test. Be specific. Testability requests are sometimes met with concerns regarding security ("Testability will open up back doors that can be used by hackers."), privacy ("Testability will reveal private information") or performance ("Testability will slow down the software."). Sometimes these issues are raised as an insincere way of forestalling further discussion. More conscientious teams find there are often ways to address these concerns. There is a basis for concerns with security. A security flaw in the Palm OS that allowed anyone to access password-protected information took advantage of a test interface (Lemos 2001). It's fair to take a serious look at how testability hooks and interfaces might be exploited. There are two ways to protect interfaces: one is to simply remove them from production code. This makes many people nervous because it means that many of the software testing suites and techniques developed for the product won't be able to executed against the final production version of the software. The other method is to use encryption keys to lock testability interfaces. As long as the keys are secret, the interfaces are inaccessible. A third method, that doesn't recommend itself, is to keep the test interface a secret and hope attackers don't discover it.

Privacy issues arise with logging. One solution for this is to log information in a form that can be reviewed by users. If they can see how the information is being logged, they're likely to be more comfortable sharing their logs when errors occur. As mentioned in the paper, blanket concerns about performance have no merit. Testability code should be added as appropriate, and performance concerns should only be raised if profiling actually indicates that it is causing a problem. If so, then it will need to be reduced.

There are surely risks entailed by testability. But a lack of testability has serious risks as well.

Some testers use opportunities to discuss testability as an opening to vent a litany of complaints about programmers' lack of discipline, poor documentation or chaotic processes.

Airing such complaints helps ensure that such opportunities won't arise again. If testers need information or cooperation, they should be specific and let programmers know how this will help them.

Sometimes testers are convinced that the design is chaotic ("a big ball of mud": Foote and Yoder 2000). Sometimes they are right. But what is the point of making a scene? If it really is bad, then it is going to be easy to find bugs: they'll be popping out everywhere. Testers shouldn't be quiet about their concerns, but there is no point in being emotional. The fact is that many programmers accept less-than-ideal designs because of project constraints. They know they are taking a risk. Testers who are uncomfortable with these risks need to demonstrate how large that risk really is. That's what testing is all about.

Testability features often result in direct value to customers as well. This includes testing hooks that developed into user scripting languages cited earlier or accessibility hooks that make software easier to use both by visually impaired users and by testing tools (Benner 1999). It's often hard to predict how users or other products will take advantage of the increased ability to observe and control a software application.

Testability takes cooperation, appreciation and a team commitment to reliability. In the words of President Reagan, trust, but verify.

Appendix. What Is Testability?

This paper defines testability in terms of visibility and control, focusing on having reliable and convenient interfaces to drive the execution and verification of tests. This definition is meant to broadly include software design issues that affect testing. It is meant to stimulate thinking.

This appendix surveys other definitions for testability and the contexts in which you may find them useful.

Some define testability even more broadly: anything that makes software easier to test improves its testability, whether by making it easier to design tests and test more efficiently (Bach 1999, Gelperin 1999). As such, Bach describes testability as composed of the following.

- *Control*. The better we can control it, the more the testing can be automated and optimized.
- *Visibility*. What we see is what we test.
- *Operability*. The better it works, the more efficiently it can be tested.
- *Simplicity*. The less there is to test, the more quickly we can test it.
- *Understandability*. The more information we have, the smarter we test.
- *Suitability*. The more we know about the intended use of the software, the better we can organize our testing to find important bugs.
- *Stability*. The fewer the changes, the fewer the disruptions to testing.

This broader perspective is useful when you need to estimate the effort required for testing or justify your estimates to others.

The IEEE defines software testability as "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met." (IEEE 1990) This gives us an unpunctuated clod of ambiguous grammar and vague sentiments. In other words: no help.

Voas defines testability in terms of observability. To wit, he defines it as the inverse of the likelihood that a fault could lay undetected after running a set of tests. This happens when a fault is triggered but nonetheless fails to propagate and cause a failure (Voas & Friedman 1995). This is a good thing for the user, but not for the tester. Voas defines the apparatus needed to turn this definition into a measurement. It may be useful when measuring the visibility of software to testing.

References

Adams, Ed (2001). "Achieving Quality by Design, Part II: Using UML." The Rational Edge, September. http://www.therationaledge.com/content/sep_01/m_qualityByDesign_ea.html

Bach, James (1999). "Heuristics of Software Testability."

Benner, Joe (1999). "Understanding Microsoft Active Accessibility", in *Visual Test 6 Bible* by Thomas R. Arnold II, IDG Books.

Bray, Brandon (2002). "Compiler Security Checks In Depth," MSDN, February, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchCompilerSecurityChecksInDepth.asp

Cohen, Noam (2000). "Building a Testpoint Framework," Dr. Dobb's Journal, March.

DiMaggio, Len (2000). "Looking Under the Hood," STQE Magazine, January.

Foote, Brian and Joseph Yoder (2000). "Big Ball of Mud" in *Pattern Languages of Program Design 4,* Harrison et al, eds. Addison-Wesley. http://www.laputan.org/mud/mud.html

Fowler, Martin (2000). *Refactoring*, Addison-Wesley.

Fowler, Martin (2000a). *UML Distilled,* Addision-Wesley.

Fowler, Martin (2001). "Separating User Interface Code," IEEE Software, March.

Friedman, Michael & Jeffrey Voas (1995). *Software Assessment: Reliability, Safety, Testability*, Wiley.

Gelperin, David (1999). "Assessing Support for Test," Software Quality Engineering Report, January.

Guckenheimer, Sam (2000). "What Every Tester Should Know About UML," STAR East, May.

Hayes, Linda (2000). "I Want My Test API," Datamation, December. http://datamation.earthweb.com/dlink.resource-jhtml.72.1202.|repository||itmanagement|content|article|2000|12|06|EMhayesquest12|EMhayesquest12~xml.0.jhtml?cda=true

Houlihan, Paul (2002). "Targeted Fault Insertion," STQE Magazine, May.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 610.12-1990.

Information Systems Examination Board (ISEB) (1999). Software Testing Foundation Syllabus, version 2.0, February. http://www1.bcs.org.uk/DocsRepository/00900/913/docs/syllabus.pdf

InstallShield (1998). "Creating a Silent Installation." http://support.installshield.com/kb/display.asp?documents_id=101901

Johnson, Karen (2001). "Mining Gold from Server Logs," STQE Magazine, January.

Kaner, Cem (1998). "Avoiding Shelfware: A Manager's View of Automated GUI Testing."

Kaner, Cem (2000). "Architectures of Test Automation." http://kaner.com/testarch.html

Lemos, Robert (2001). "Passwords don't protect Palm data, security firm warns," CNET News.com, March 2. http://news.com.com/2009-1040-253481.html

Lesuer, Brian (2001). "Supporting Custom Controls with SilkTest," Test Automation Conference, August.

Marick, Brian (2000). "Using Ring Buffer Logging to Help Find Bugs." http://visibleworkings.com/trace/Documentation/ring-buffer.pdf

Maguire, Steve (1993). *Writing Solid Code.* Microsoft Press.

Mercury Software (2000). *WinRunner Advanced Training 6.0: Student Workbook*.

Meyer, Bertrand (1997), *Object Oriented Software Construction*.

Mitchell, Jamie L (2001). "Testability Engineering," QAI.

Mullins, Bill (2000). "When Applications Collide," STQE Magazine, September.

Nyman, Noel (2001). Personal communication.

Nyman, Noel (2002). Personal communication.

Payne, Jeffery, Michael Schatz, and Matthew Schmid (1998). "Implementing Assertions for Java," Dr Dobbs, January.

Philips, Todd (2002). "Testing Applications with AppVerifier," MSDN, June. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnappcom/html/AppVerifier.asp?frame=true

Pettichord, Bret (1999). "Seven Steps to Test Automation Success," Star West, November 1999. http://www.io.com/~wazmo/papers/seven_steps.html

Pettichord, Bret (2000). "Three Keys to Test Automation," Stickyminds.com, December http://www.stickyminds.com/sitewide.asp?ObjectId=2084&ObjectType=COL&Function=edetail

Rubenstein, David (2002). "Who Knows What Evil Lurks in the Hearts of Source Code?," SD Times, June 1. http://www.sdtimes.com/news/055/story4.htm

Structu Rise (undated). Textract, a screen text capture OCR library. http://www.structurise.com/textract/index.htm

Valaer, Laura and Robert Babb (1997). "Choosing a User Interface Development Tool," IEEE Software, July.

Segue Software (1998). *Testing Custom Objects: Student Guide*.

Whittaker, James A (2003!). *How to Break Software*, Addison Wesley.

Xconq. http://dev.scriptics.com/community/features/Xconq.html

Zambelich, Keith (2001). "Designing for Testability: An Ounce of Prevention," unpublished.

