



## Testability Engineering

by Jamie L. Mitchell  
With  
Beth Anderson  
Chuck LeCount  
Jan Bagley  
Joy Islam

*This paper was originally presented at the QAI 2001 Testing Conference. It was later also published in the Journal of Software Testing Professionals, Volume 3, Issue 2.*

### Prologue

The one constant of software development is change. The rate of change has accelerated substantially in the last decade; the complexity of applications and environments has increased and schedules have been pinched. Under these conditions, software quality tends to suffer. If we are going to keep up with change and improve the software we are creating, we should find ways to optimize our QA processes. Improved testability is one way we can try to diminish the widening quality gap.

The IEEE definition of testability is:

**“... 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.”<sup>1</sup>**

Perhaps a clearer definition would be:

**“Testability means having reliable and convenient interfaces to drive the execution and verification of tests.”<sup>2</sup>**

For the purposes of this article, we will use the term testability (or lack thereof) when discussing incompatibilities between the testing environment, the system under test, test tools, and good testing processes. These problems can raise the cost and reduce the amount of testing that we can do. Most of the testability issues we will discuss in this article have to do with test automation, although some are also valid for manual testing.

Traditionally, software test automation is seen as one solution to the time/complexity problem. However, automation alone will not solve all of the issues. When systems are

---

<sup>1</sup> IEEE Std 610.12-1990 "IEEE Standard Glossary of Software Engineering Terminology"

<sup>2</sup> Definition by Bret Pettichord, as supplied in an unpublished document made available to the AWTA II attendees.



Innovative solutions to maximize your testing investment.

built in such a way that automation tools cannot interact efficiently with them, the cost of automation may easily exceed the cost of comparable manual testing.

There is only one good reason to utilize automation in our testing: a positive return on investment (ROI.) If we get higher quality, reduced time to market, or some other tangible, measurable benefit by using tools to enhance our testing, then we have a positive ROI. If the creation and maintenance of test automation costs more than the equivalent amount and level of manual testing over the life of the project, then we have a negative ROI and no justification to automate. Due to the large, up-front investment that must be made in automation, and the possibility of hefty payback, test automation should be seen as a strategic investment made by an organization. Any problems that negatively impact test automation, reducing its ROI, need to be dealt with in the context of the whole organization, not just the test team.

Development tool vendors are always upgrading their tool sets – often outstripping the ability of automation test tool vendors to keep up with the changes. Developers do not always use best practices when using these development tool sets. Application Programming Interface (API) differences in seemingly similar operating systems may cause unexpected behaviors in test scripts. These are some of the factors that can limit the effectiveness of test automation tools. Far too often, a sizable fraction of the cost of automating testing is simply trying to get the tool to coexist with the application. Commonly called “building workarounds”, the act of programming in the test tool language to make up for shortcomings in the tool and eccentricities in the application is an expensive, difficult-to-maintain process which limits both the amount of testing, and the ROI which can be realized. Testability engineering can be seen as a holistic approach to these kinds of problems at a high, organizational level.

### ***Pragmatic Solutions***

The solution to many testability problems will require industry wide buy-in, standards, and changes to acceptable practices. This article will not deal with an all-encompassing solution to the industry-wide testability dilemma. Instead, we will deal with smaller, pragmatic steps that can be implemented in an organization in the short term to help solve some testability problems right now.

Each section below will contain suggestions for dealing with testability issues. Many of these suggestions were discussed and developed during the AWTA II workshop<sup>3</sup>; others are the result of over thirteen years of hands-on testing by the QA consultants at BenchmarkQA, Inc. in Minneapolis, MN.

---

<sup>3</sup> AWTA II. The Austin Workshop for Test Automation (2) was held in Austin Texas on Jan 26-27, 2001. The following attendees got together in a moderated workshop to discuss the issue of software testability, directly resulting in this and several other papers on the issue: 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, Stan Taylor.



Many of the suggestions made in this article have been implemented at client sites; others are the result of the application of common sense to the constantly changing development environment of today. To be honest, we have never had one organization adopt all of the suggestions that are given here. But we haven't given up hope<sup>4</sup>. These ideas should be seen as the ideal, a destination we would like to reach one day. Many of the following suggestions, if put into practice, would also increase usability of our systems for our customers; in the final analysis, isn't that why we are in this business?

### **Automation Specific**

For automation purposes, testability might best be defined as the amenability of the system under test to operate correctly with a test tool. The two facets of good automation testability are:

- **Observability**: the ability of the test tool to capture data and information from the system under test to allow the tester to compare expected results with actual. This may include hidden information not normally seen by a manual tester.
- **Controllability**: the ability of the test tool to fully operate the system, allowing the tool to direct any action that a manual tester could complete – and possibly more. The test tool must be able to cause change to screen controls in the same way that a user is expected to, including mouse clicks, cursor movements and keystrokes. We may also want to expose additional methods generally not available to the manual tester, such as triggering an error or pre-setting a specific state.

No amount of black box testing can hope to solve all of the issues inherent in the multi-tiered, multi-application environments we are facing. We should begin to get involved in the development cycle by ensuring that the architecture and design of the system includes features that will allow us to more easily test the system. We should also ensure that our test groups have sufficient technical training to understand the environments, tools, and systems they will be testing.

Testability is rarely something that can reasonably be added late in the development cycle. If it is to be built in, it is best to build it in early, preferably starting in the high-level design phase.

### **Management Buy-in**

Testability is often not a serious technical issue – it can usually be achieved with relatively few practical innovations. Often the technology needed is well known. Instead, testability issues are more often managerially, politically, and educationally based issues; the will to design and code testability into a system is often missing. If test

---

<sup>4</sup> A person's reach should exceed their grasp: or what's a heaven for? Robert Heinlein.



Innovative solutions to maximize your testing investment.

automation is indeed a strategic investment, then it is essential that we try to get both developer and management buy-in to the needs of the test automators.

Testability has traditionally not been on management's list of priorities when considering software development concerns. Their major concerns when thinking of a software system under development are usually functionality, cost, schedule and risk. What we should do is try to convince management that ignoring testability may have major ramifications in the guise of performance, availability and quality and has a direct influence on functionality and risk. Ignoring testability causes a very real threat to both schedule and cost of the system. In essence, we should present testability as an option that usually saves money. If it is seen as an additional cost, we will not be able to argue for it effectively.

If we fail to get management buy-in, we may be unable to influence the direction of the development cycle in any meaningful way. Without management support, there may be little chance that the developers will listen to or be swayed by testability concerns. On the other hand, when management understands that addressing testability issues can result in positive ROI for the whole organization, all members of the team are likely to get the message.

### ***Developer Buy-in***

While a personal relationship between the test group and the development group may help attain buy-in for testability, we cannot always expect to achieve our strategic goals this way. The quality of the system should be everyone's aim; however, all too often in the real world, it appears to be "us" versus "them", testers versus developers. We need to find a way to convince developers that increasing the testability of the system will benefit them directly and only ask for a management mandate as a last resort.

An important consideration ought to be that the tools, processes, training and personnel supporting testability are able to keep up with the developers' schedules. If testability is seen as holding the project back, it will likely become a dead issue.

### ***Testability Considerations***

#### **Training**

The test team should have at least one<sup>5</sup> technically trained member for understanding technical documentation and interfacing with the development team. The increasing complexity of today's systems requires more technical knowledge of the internal working of systems than in the past. If developers cannot communicate efficiently with the testing team about technical details, they likely won't. Often, this technically adept member of

---

<sup>5</sup> Preferably more!



the test team will be the automator; they ought to have development class technical skills to be able to perform their job to any degree of success.

## Tools

If a test tool has not yet been chosen, a major consideration in its choice should be the match with the tools, environments, and processes used by the developers. Each test tool has strengths and weaknesses. While it is true that any tool can be made to work in any environment against any system, it is also true that this is not always cost effective. The cost of the extra development it takes to make a tool work against a system should be included in the overall cost of the testing, as well as the cost of the increased maintenance of the testing artifacts. We should select a tool which gives us the best ratio of testing to cost.

When an organization finds that the test automation tool they are using does not address all of their testing needs<sup>6</sup>, an economic analysis should be made as to the viability of bringing in other tools. In many cases, the highest ROI is afforded by using the correct tool for each system or environment; the initial purchase price and ensuing maintenance costs of the tool are only a small fraction of the overall automation cost.

If the tool supplies an implant (or add-in) for the development system, it should be compiled into the application. This increases the efficiency of the automation tool – application communications.

The fonts used in the application to be tested should all be verified for visibility in the test tool – especially if the tool uses optical character recognition (OCR) techniques for picking up text from the screen.

## Application High-Level Design

As noted above, every GUI control in an application has two absolute needs for testability: observability and controllability. To ensure testability, the automation group should be involved in the selection of controls to be used in the application. The following points should be understood about control types:

- Unless there is a strategic, structural reason, standard Windows controls should be used when building forms. Standard methods and properties should be used, as is, when the test tool works correctly with them.
- If there is a reason to override standard control methods or properties, then the testability of the changes ought to be checked with the test group against the test tool.

---

<sup>6</sup> As an example, there is one automation test tool that does a wonderful job of automating most client / server applications, but fails miserably when it comes to working with web applications. It can be made to work with browsers, but only at the cost of writing custom code to parse all of the HTML coming down to the browser. So where is the ROI in that?



Innovative solutions to maximize your testing investment.

- If standard controls are not available for the task, then the control chosen for the task should be evaluated with the test tool. A prototype interface should be created to allow real time testing of the control with the tool.
- If a control is to be built from scratch for the application, or if an existing control is to be sub-classed to create a new control, then the design of the new control should include testability characteristics.
- If the selected control is not observable or controllable, then special methods and properties should be added to the control to help the tool interact with it. These may be made available to the tool in one of several ways:
  - The methods and properties may be directly available to the test tool (example: Java methods directly executable in WinRunner.)
  - The methods and properties may be directly available to a custom written DLL which is then available to the test tool using standard DLL access calls.
  - The properties may be made visible by writing to file or shared memory. The methods may be triggerable by custom windows messages sent by the test tool.

There should be a uniform way to start and shut down the application. This method should be external to the application so the test tool can easily perform it.

### **Application Low-Level Design**

When a detached text string identifies a control, then the developers should ensure the string is positioned close enough to the control for the test tool to utilize it as an identification method. This will prevent the test tool trying to use other, less static identification methods.

Invisible controls should be completely documented. The current state of all invisible controls should be available to the test tool if work flow decisions are made based on its state.

There should be a standard way to close every discrete window in an application.

### **Coding Standards**

Assertions should be required when available in the development language. Any assumptions made by a developer should be subject to assertions. The assertion output can be made available at run time through some interface for the tester or test tool to utilize.

### **Unit Testing**

The test group should be responsible for inspecting the unit test plans for the developers. In all cases, some formality should be introduced to the unit test planning process.



## Build Testing

A set of build verification tests (BVT) – sometimes called smoke tests -- can be made available to the developers so they can conduct minimum tests on private builds before folding in changes. Catching bugs earlier will help recover the extra cost.

## Naming Conventions

Naming conventions in the application should ensure that they help rather than hinder testing. A few examples:

- Since Rational Robot uses semi-colons as field separators, they should never be used in labels, window text, or captions if Robot is to be used as the test automation tool.
- Persistent, meaningful and unique control identifying names should always be used: for example, FirstNameEdit is much better than Edit1.
- No two windows should ever have the same name. Unique window names are essential for the logical naming within the test tool.
- A single common error dialog should be used for presenting all errors. Using multiple error dialogs for different purposes makes automation error identification and handling very difficult.
- For text-screen and green-screen applications, each unique screen should contain a unique number or string identifying it.
- Avoid the single dialog, multiple window model<sup>7</sup> if at all possible. This structure is very difficult to map for GUI learning. If this model is to be used, ensure that all controls are uniquely named. Do not reuse controls for different purposes.
- If the control names are not visible to the tool being used, other identifying values should be assigned to the controls. For example, unique numbers in the MSW\_ID field.
- A prototype where the controls to be used are placed on a form to test interaction with the automation tool is very helpful in determining what additional support is needed. An example of this would be the form containing all of the Java Swing controls that are supplied with the Swing library.

## Exception and Error Handling

The ability to simulate hardware or software failures could be built into the system, triggered by a hidden control or method available to the test tool. The ability to set states that trigger exceptions and error handling routines would also very helpful.

Each error message should be unique. There should never be a case where the same error message is used for multiple errors. A unique number in each error message helps identify it to the automation tool.

---

<sup>7</sup> Multiple windows superimposed on a single parent window.



## Timing Issues

Critical timing paths should be clearly specified so that tests can be devised for validating them.

The test tool should be able to control virtual time in the application. This will be different for each system; it should be more than simply changing the real-time clock. This might be used for aging the workstation when testing financial systems, for example.

## Logging

When critical error messages are generated in the system, they should be well documented in a time-stamped log that can be accessed by the testers. Included should be an indication of where the error was generated (i.e. component, object, etc.) Testers need to be able to trigger and access all possible logs that can be written.

Debug mode should include documentation or logging of time critical response.

Debug mode could also have the ability to write resource usage to logs or be made available by API.

Allow the run-time system to be switched to verbose logging mode<sup>8</sup>. During testing when a fault is found, it will help in the trouble-shooting phase.

## Installation

After install, the system should do a self-check on the state of the install and provide a positive indication of the success. The installation should provide a log that will indicate the reason for an installation failure.

## Debug Builds

Event triggers for critical or long running tasks, signaling both the start and end of internal processes, should be available in a debug build.

Test builds should include hooks for a memory checker (MemCheck, NuMega, or such.) This should be used with the test tool and during manual tests.

## Visual Cues

A positive visual cue that can be easily checked via the test tool should always be used to signify the completion of any long-running process. Examples of a positive visual cue

---

<sup>8</sup> This mode causes the system to generate many more logging messages as it runs. It is often useful to have switchable logging modes: both a terse mode and a verbose mode.



Innovative solutions to maximize your testing investment.

include a “completed” message in the status bar or the enabling of an object in the window. Audio and cursor-change cues are insufficient for good testability.

## Communications

Testers should be able to simulate server messages if needed. This would allow the tester to test front-end reactions to error messages and specific server conditions.

The test tool (or custom DLL) should be able to insert and monitor messages in the messaging system. Messages used in the application to interface between objects, modules, and systems should be well documented.

## Processes

There are no such things as minor code changes. Any change should be sized to determine the impact on testing. Strict version control ought to be used; all changes should be reviewed by the test group (especially the automation group) as well as the design, documentation and development groups.

Testability bugs should be included in the bug-tracking database, and taken seriously by all users of the database.

Testers need to understand the version control system being used. The testers should put all testing artifacts into the version control system.

Testers need to be active participants in requirements and design meetings.

Unless there is already an active QA group who owns it, the test team should own the inspection processes for requirements and design meetings.

## Documentation

It is essential that testers have access to all documentation that the developers use in creating their designs and code. This includes all requirements and design documents.

All changes that are authorized should be documented thoroughly; this documentation should be available and updated with each build. This includes bug fixes and planned enhancements. Changes to any file (including DLLs) in the system should be documented in the change notes.

Mandatory build release notes for every build ought to be made available to the test team. Error tracking tools should have a category for release documentation shortcomings.

All module interfaces should be documented thoroughly. “Development by Contract” should be the rule. This documentation should be supplied to the testers.



Innovative solutions to maximize your testing investment.

The database schema should be completely documented for all developers and testers. All changes in the schema should be also documented. A where-used catalog for all database entities should be generated and made available to the testers.

Each error message that exists in the system should have documentation fully explaining the action or conditions that triggers it.

Transaction flow across components should be documented.

Testers should have access to all documentation that covers actual use of the system by users. This includes requirements and design docs, feedback from customers, use cases, usage patterns, etc.

All the different types of users of a system should be completely documented. This includes both authorized and unauthorized users and what it means to be either.

Testers should have access to the raw customer service logs.

All co-existing applications should be documented, including the required versions. All DLLs should be documented including the required versions. This includes the system DLLs needed to make the system run.

## **Data**

Testers should be provided copies of the data files used by developers. These should be critiqued by the test group to provide feedback to the developers for unit testing.

Testers should have ownership of the test database and the test environment. At no time should changes to either be done without test team authorization.

Processes should be developed for the easy reloading of the test database to ensure repeatability of automated tests. This process should be under the control of the testers to ensure it can be performed as needed. If the database cannot be easily reset, then a scheme can be developed to allow unique data be used for each test series. Alternatively, a scheme to reverse all of the changes performed by a test series may be created.

If there are no direct ways to access process data, a back door may be requested. This is especially true when an error may corrupt data in such a way that the gross failure does not show up until later.

Under some circumstances, it would be useful to have a notification message passed when certain data is changed. For example, suppose a financial application was being updated asynchronously by another process. The tester would be well served to be able to track any such changes when reviewing a failed test case.



Innovative solutions to maximize your testing investment.

## **Test Environment**

The test environment should closely mimic the expected production environment wherever possible. Formal risk analysis should be performed for any substantial differences.

### **Summary**

The time has passed when developers could simply “throw the code over the wall” to the testers and expect that all of the bugs would be ironed out. In today’s fast-paced, high-tech software development environments, all participants should work together to ensure success. It is essential that designers and coders, testers and managers cooperate.

Planning for testability during the design and coding phases of a software product can provide significant efficiency in testing the product and has a very small cost and schedule impact on the development processes. In net, the ROI for considering testability early in a project is positive.