BIO
PRESENTATION
SUPPLEMENTAL MATERIALS

T15

November 18, 2004 3:00 PM

# TEST HARNESSES FOR API TESTING

**Michael Sonshine**
**Intuit Inc**

International Conference On
Software Testing Analysis & Review
November 15-19, 2004
Anaheim, CA USA

# Michael Sonshine

Michael Sonshine has been involved in QA and Test Automation for more than 14 years. At Intuit he holds the position of Principal QA Software Engineer in the Shared Development and Services Division where he has wrote the testing standards document and created the current Microsoft API Test Harness and associated tools which are used for automated testing. He also serves as the chair of the Test Automation Special Interest Group that works to standardize testing and solve current testing problems across the company.

Prior to working at Intuit he was responsible for the development and implementation of an automated testing program at a company that had no automated testing in place when he arrived. He built the API test tools, coordinated testing between multiple sites and was active in creating both API and GUI level tests as well as building custom test tools for those QA personnel who were not engineers.

Prior to starting in that position he was actively involved in creating automated tests and test tools for the Unix environment and served as the corporate representative to the testing subcommittee of X/Open. He has been involved in software development in and out of the testing area for 20 years.

# API Testing and Use of a Test Harness

- Michael Sonshine
- Principal QA Software Engineer
- Intuit, Shared Development & Services
- San Diego
- (858) 525-8987

# API Test Harness

- **Basic Set Of Functionality**
- **Portability**
- **Usability**
- **Standardized Inputs**
- **Extended Functionality**

# API Test Harness

- What must a test harness do?
  - Gather Input Information
    - Data files
      - Externalize data from test
      - Allow test cases to be added without code modification
      - Contain control information
      - Contain pseudo-data
    - Configuration files
      - How to run
      - Test run specific information
      - General
      - Test specific

# API Test Harness

- What must a test harness do?
  - Modify Setup information
    - Where are the inputs?
    - Where are the outputs?
    - How do I act?
  - Control Test Execution
  - Provide visual evidence of operation
    - What is executing?
    - How is the status?
    - What has completed?

# API Test Harness

What must a test harness do?

- Provide Configurable Logging
  - Trace to Summary
- Provide Data Tools
  - Generate Descriptions
  - Renumber (if needed)
  - Unique Ids (if needed)
  - Validate Format

# API Test Harness

What must a test harness do?

- Provide Reports
  - Summary reports
  - Timing reports
  - Coverage reports
  - Test Case Status - Pass, Fail, Blocked, etc
- Client/Server functionality
  - Microsoft
    - Web Service
    - .NET Remoting
  - Java
    - Web Service
    - RMI

# API Test Harness

- What must a test harness do?
  - Automated Install
    - Microsoft deployment
    - InstallShield
    - Wise
    - Zip

# API Test Harness

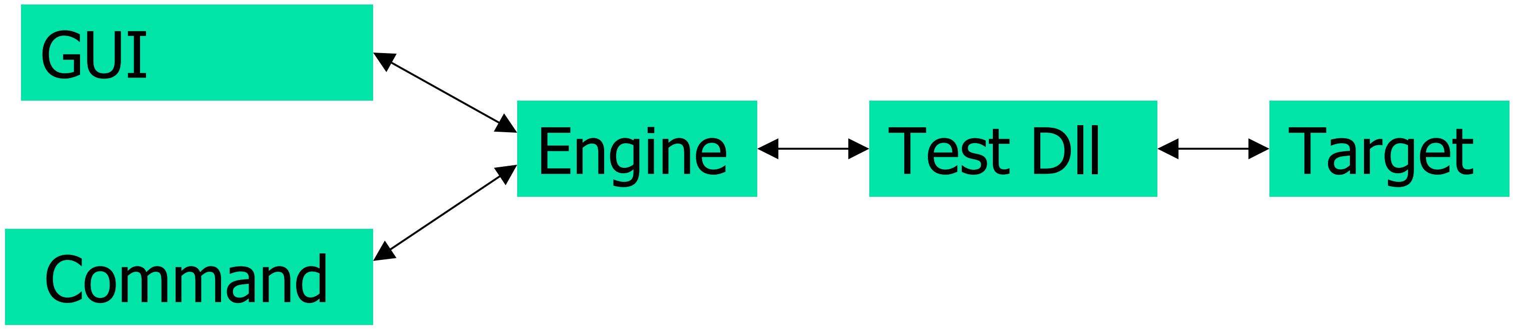
- How do we provide for functional portability?
  - Standardize Data File(s)
    - Generalize format so it can be used for multiple projects
    - Provide control information for Test Harness to use
    - Create set of meta-data values for use in testing
    - Xml
    - Make the format suitable for any language
  - Standardize Configuration Information
    - Standardize format
    - Allow hooks between the data file and config file
    - File for Test Harness
    - File for Test dll/assembly

# API Test Harness

- How do we provide for functional portability?
  - Standardize log file
    - Standard set of test case results
      - Pass, Fail, Blocked, Untested, Inspect, Incomplete
    - Standard error tag format
      - Uniquely identify errors
        - Alpha-numeric tag
        - Tag extension (test dependent)
    - Provide configurable logging
    - Easy to identify errors
      - NOTE, FATAL, ERROR, →, INSPECT

# API Test Harness

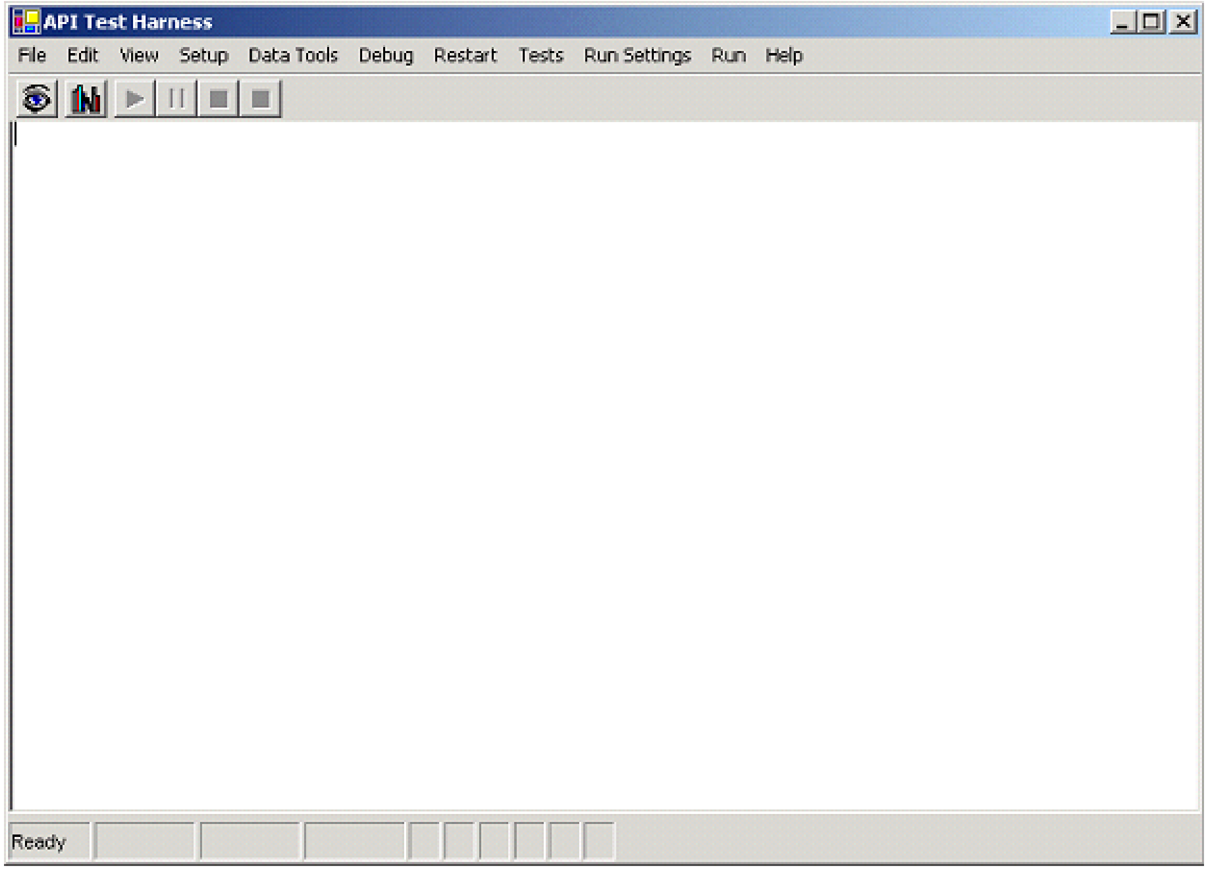
- How do we provide for functional portability?
  - Separate the GUI/CommandLine from the Engine
  - Separate the Engine from the test dll



# API Test Harness

- How do we make it user-friendly?
  - Many types of users
    - Test Developers
    - Testers
    - Application Developers
    - Managers
  - Testers

# API Test Harness



# API Test Harness

## How do we make it user-friendly?

- Testers
  - Provide for auto install
    - Microsoft deployment
    - InstallShield
    - Wise
  - Insure setup is as easy as possible
    - Comment all entries needed
    - Provide documentation for installation
  - Provide Visual Progress
    - Number of passes, fails, etc
    - Summary information

# API Test Harness

- How do we make it user-friendly?
  - Provide a way to
    - stop the tests early
    - run single or multiple tests
    - select test cases to run
    - user tools
    - Visual confirmation of actions

# API Test Harness

- How do we make it user-friendly?
  - Test Developers
    - Template for code
    - Template for data
    - Complete libraries
  - Managers
    - Automatic Reports
    - Summaries

# API Test Harness

```
<TestSuite>
  <TestCase nbr=“1” id=“Sample-A1B2” />
    <Description value=“Sample test case” />
    <Blocked value=“false” />
    <Untested value=“false” />
    <Dependencies>
      <Dependency value=“” />
    </Dependencies>
    <PrevRefs>
      <PrevRef value=“” />
    </ PrevRefs>
    <Attributes>
      <Attribute value=“” />
    </ Attributes>
```

# API Test Harness

```
<Obj name=“object1” />
   <Mthd value=“method1” />
     <Params>
       <Param name=“param1" type="string">value1</Param>
       <Param name=“param2" type="string">value2< /Param>
     </Params>
     <Return value=“false” />
     <Exception value="false" />
     <Sequence start="true" />
   </Mthd>
   <Mthd value=“method2” />
     <Params>
       <Param name=“param1" type=“integer">value1</Param>
     </Params>
   </Mthd>
 </Obj>
```

# API Test Harness

MetaData

- Tokens
  - #T:TranslateValue
    - Translate value based on config entry
    - Done by Test Harness prior to call to test dll/assembly
  - #R:N:MinLen:Maxlen
    - Generate random numeric string value of random length between MinLen and MaxLen characters
    - Done by Test Harness prior to call to test dll/assembly
  - #R:A:MinLen:MaxLen:U
    - Generate random uppercase alpha string value of random length between MinLen and MaxLen characters
    - Done by Test Harness prior to call to test dll/assembly

# API Test Harness

- MetaData
  - Tokens
    - #Prev
      - Return the object used in the previous test case.
      - If done by Test Harness requires object be serializable
    - #Prev:ParamName
      - Return the parameter used in the previous test case
      - If done by Test Harness requires object be serializable
    - #S:15

# API Test Harness

Test dll/assembly

- Code needed for testing is in test dll/assembly
- Only 5 method calls are necessary
  - preTestSetProcessing
    - Perform any test setup requirements
  - preTestCaseProcessing
    - Perform any test case setup requirements
  - testCaseProcessing
    - Process test case
  - postTestCaseProcessing
    - Perform any test case cleanup (tear-down)
  - postTestSetProcessing

# API Test Harness

- Other items
  - Method/Property coverage information
  - Timing Statistics
  - Timing Report

# API Test Harness

- Basic Functionality
  - Specify Inputs
  - Specify Outputs
  - Specify Control Info
  - Display Progress
  - Display Completion Information
  - Data Tools
  - Same interface for MS and Java
  - Start/Stop
  - Helper Libraries (xml, database, etc)

# API Test Harness

- Extended Functionality
  - Pause/Restart
  - Schedule delayed start
  - Results Summary
  - Dependency verification
  - Multiple executing copies
  - Database test case storage
  - Windows service control
  - Max run times for test set and/or test case

# 1. Introduction

Test automation has become an increasingly important part of the QA process in many companies. The reasons are well known to all QA professionals – the ability to create a set of repeatable tests, the ability to create a trail of what has been tested and what has not been tested, the need for quick turn-around during product releases and the constant pressure of finding and training people for large-scale manual testing efforts.

The easiest path to test automation has been through the use of GUI based testing tools. These tools seemed to offer what most organizations felt they needed. Users could be trained quickly, tests could be created through the "capture-playback" functionality and later modified to act in a more repeatable fashion. Companies like Mercury, Rational, Segue and others produce sets of tools for functional and load testing and many testing efforts rely heavily on them. While the ability to create tests quickly using these tools is a help, the tests themselves often suffer from well-known problems. They tend to be fragile, when something breaks tests often do not know how to get back to a well-known state and consequently hang, GUIs change repeatedly and the tests usually have to be revised just as often and, since it is difficult to construct these tests before the GUI is available, there is a time lag between product release and test. Still, these tools have often proved useful for many companies in shortening the testing time and the associated scripting language for the tools allows the use of sophisticated test control code to, at least potentially, take care of some of the problems mentioned above.

One of the limitations of these tools is that they are designed for testing GUIs and hence did not seem to address non-GUI applications. Some QA organizations have addressed this limitation by requesting that development organizations create simple API test applications which themselves would call the APIs, but would present a GUI front-end to the tester and hence could be automated through the use of these tools.

A second path to test automation centers on direct API level testing. Initially this type of testing seemed applicable only to non-GUI based applications but allowed QA organizations to create very thorough test sets to validate the functionality of such applications. However some organizations have also applied this type of testing to GUI based applications reasoning that GUI-based software only ties a set of GUI actions into calls to underlying events and method calls. In this view text box data, button and checkbox settings, list data and the like become nothing more than parameter data for some event or method call which can just as easily be made by a test as by a GUI. Given this view API testing can be used, and used very effectively, for testing both GUI-based and non-GUI based software.

Regardless of the type of application, testing at the API level offers many advantages when compared to GUI-based testing.

- **Much more thorough testing can be done.**

It is easier to create a large set of possible data at the API level than through the GUI. Special data values such as very long fields or nulls can easily be used.

- **Testing can be done more quickly.**

  Testing of a specific call through a GUI may require a relatively long time for the test to get to the window where the call will actually be made. Testing through the API takes very little time and hence many such calls can be made repeatedly.

- **APIs tend to be more stable and hence fewer changes are required**

  GUI screens often change with every release. Screen and object names change, items move from one location to another, new buttons or text boxes are added, existing ones are removed and new intermediate screens are created. Changes like these and others not listed often cause havoc in GUI-based tests. The underlying APIs often are largely untouched and tests written for them will remain valid. If they do change the required test changes are often minor.

- **APIs tend to be defined earlier in the process and thus test cases and test code can be created earlier.**

  A frequent problem with GUI-based tests is that it is difficult to create the test without the GUI itself. While there are approaches to address this problem it typically remains a major roadblock to the creation of a functional set of tests. Since one of the primary goals of any test effort is to get done as quickly as possible, API testing can be a big help in this area.

- **Changes in APIs are generally known in advance and thus test case changes can be made earlier.**

  Test changes can be made as soon as changes are known and published and do not need to wait for the release of a GUI. Often development organizations will notify QA or proposed changes well in advance of the actual software release.

This discussion will center on API level testing and on the use of a Test Harness to help in the effort. The term Test Harness is used widely through the industry and is generally understood to refer to a software package that controls the running of automated tests and often provides a set of tools to help create the tests themselves. Given this definition many tool sets quality as "Test Harnesses" and the term has been used to refer to both GUI and API level test frameworks as well as storage tools used to launch tools.

Exactly what set of functionality should a "Test Harness" make available to be useful? If we consider this to be a normal software project, what are the requirements? At a more basic level, exactly what should a "Test Harness" do? In this discussion we will address three main questions.

- **What must a Test Harness do?**
- **How do we construct such a tool in a way to make it usable in different projects?**
- **How do we design a user interface so the tool will be easy to use for the wide range of possible users – developers, domain experts, contractors and testers?**

## 2. What Must A Test Harness Do?

In general it is easy to construct a high level list of needed functionality. As a minimum a Test Harness should be able to perform the following actions.

### 1. Gather Configuration Inputs From the User

Automated tests vary considerably. Some are batch files running on Unix or Microsoft systems calling command line functionality, some are full GUI presentations running under X/Windows on Unix or on Microsoft or Apple systems. All such tests require data to function. While it used to be common for tests to wrap this data in code, the modern set of automated tests generally try to center only test functionality in the test itself and to externalize test data, control structures and configuration information in either a set of data files or a database. These externalized files (or database entries) consist of both inputs and outputs and both are essential to the running of the tests. The Test Harness must allow users to specify where this information is for any particular set of tests. There is no hard and fast rule about what constitutes a complete set of such information, and it varies considerably from Test Harness to Test Harness and from GUI interface to command-line interface.

The API Test Harness that Intuit's SD&S is currently using provides external inputs for the data files, for the Test Harness configuration file, for the test configuration file, for test execution information and, of course, for output log information. In addition the test may require the actual location of the test dll/assembly itself for use with reflection and information regarding remote access such as IP Addresses.

A file, which is in standard xml format, provides the actual data (or, in some cases, pseudo-data) used to specify both test cases for execution and the parameters and results expected. The Test Harness configuration provides information about how the Test Harness should run. The application configuration file provides control information for the test itself and thus would provide such things as target system IP addresses for this test, system file paths, translation information that is subject to change, the location of test meta-data and the like.

In addition to test inputs the user should be able to specify the locations for test outputs. A test, to be useful, must write a log file summarizing test execution and results and the name and location of this file should also be able to be set by the user.

The SD&S API Test Harness also generates execution timing information when calls are made and the user is also able to set the location of this file.

Additional control information such as logging levels, test repetition counts, startup and run times may be deemed important and should be considered.

## 2. Allow User Modification of Existing Settings

A Test Harness needs to be easy to run and the above is a fairly long list of inputs for the user to enter every time a test sequence is to be executed. Further, even if all of the information already has been input, some may require modification prior to a run. A user, having entered and run a set of tests, may wish to re-run the tests, but specify a different output log file. He or she may wish to modify the current log level to include more or less logging information in a subsequent run. Or link to a different dll/assembly to test. Frequently a test set may be re-run, but the user only wishes to execute a subset of the tests that were executed the first time and hence must be allowed to set the test cases which will be executed.

Of course the manner that this is to be done is dependent upon the Test Harness and its user interface. For the Intuit SD&S API Test Harness this involves opening a window and either modifying a text value or accessing a list box and setting and unsetting various entries.

## 3. Start, Stop And Control Test Execution

Clearly the user must be able to start test execution, but that alone is insufficient in a general test tool. Users should be start a test run, schedule a test run at some later date and time, schedule how many times a test set should be executed, stop an executing test set, pause an executing test set and, of course, restart it.

While GUI-based test harnesses are generally used to start tests immediately, they should also allow the user to start a test at some later time. The ability to start a set of tests at some scheduled later time should be a basic component of most command-line Test Harnesses since often such tools are intended to be used in connection with nightly builds to provide for automatic testing of new releases. Functionality available through a GUI should be a super-set of that available from a command-line and this implies that the ability to schedule later testing should be part of any GUI based Test Harness.

Users should be able to pause a set of tests that is executing. This need may be planned (perhaps they know that prior to execution of a specific set of tests they need to connect/disconnect some modem connection) or unplanned (perhaps they find during test execution that something needs to be taken care of and, due to the length of time the test set runs, they do not wish to stop and restart the tests). In either case the ability to pause an executing set of tests is important. And clearly if a set if paused, the user must have the ability to restart it.

In addition the user may wish to terminate an executing set of tests and the functionality should be present to allow test termination gracefully rather than having the user terminate the process itself. This allows the Test Harness to do any normal test cleanup and reporting.

While the above features are a basic set of required functionality it is often useful to be able to provide extended functionality as well. The user may find it helpful to be able to specify a repetition count for test sets (for example, run this set of tests 20 times or run this set of tests for a total of 4 hours). Because QA is often running tests against immature software there is always the problem that a test case may hang and hence it is useful to provide a way to insure that a test case does not run for longer than a specific period of time. If it does, the harness should kill the thread running the test, make a log entry and continue and thus avoid the problem of someone starting a test at 6 pm and walking in the next morning at 9 am and finding that the Test Harness is still running test case 5 of 1207. Dependency testing is another area that should be considered. Test cases are often not discrete sets of tests but form some sequence. If some object needs to be created so that it can later be updated, then if the creation test case fails the update test case should not be run. This type of dependency checking can thus prevent hours of meaningless test runs.

## 4. Provide Visual Progress Information

As testing proceeds it is important that a Test Harness provide some type of progress information to the user. Tests may run for hours and it can be extremely important not only to know that the test is actually executing, but what the current results summary is. A user who has just run a test set on a 2.8 GHz system with 1 GB of memory and knows that the test took 30 minutes to complete may terminate a test running on a 200 MHz system with 128 MB of memory because it has been running for 45 minutes and has not yet completed if there is no visual display that the test is successfully executing test cases.

It is also important that the Test Harness provide real-time summary information of passes, fails and other results. A user who knows that a test has completed on one machine with no failures may have cause to terminate a test and recheck the settings if it is running on a different machine and all the test cases are failing.

## 5. Provide Configurable Logging

Testing is usually done throughout the test cycle and the need for detailed information early in the cycle is often more important than later in the cycle. The need for detailed information for test failures may be more acute than for test passes. The information a manager may wish to see is almost certainly different from the information an engineer may need to see. All of these are reasons for the user to be able to configure the logging level before a test starts, for specific test cases or for passes and fails. The usual solution is to allow the user to specify an input log level number and for the test to log only if the specified logging level for any particular entry meets or exceeds that value. One alternative is for each test case to specify its own log level and another is for the test to automatically log passes at some low level and fails at some high level.

Any of these solutions need to be changeable by the user at run time since any particular test run may require more or less logging.

## 6. Provide Data Tools

One of the issues that we will discuss in the next section is the need for a standardized format for Test Harness inputs and the advantage of standardized outputs. Thus data files should be constructed from a standard format so the form of the data only has to be learned once by a user. Once that form is learned and understood developers can construct data for any test set from it and domain experts, who may have the responsibility of constructing new test cases, may do so with relative ease. And the same standardization, which allows ease of use by users, also allows the Test Harness to provide a set of tools that can work on that format. Thus tools can be created which automatically renumber test cases when new ones are added, generate unique Ids for test cases, display data sections, extract specific test cases and assist in the adding of new data.

## 7. Provide Reports

After test runs there is often a need for specific types of reports. Managers may want a list of the test cases actually run. Project leads may want a report of test case progress – how many test cases ran, how many passed, how many failed, how many were blocked and so on and developers may want a set of reports on failures only.

As we will discuss later in this document it is also often helpful to produce timing information. Test Harnesses are in a unique position to generate this information and subsequent timing reports can be immensely helpful as a project progresses. Timing information on any particular run may seem unnecessary, but timing comparisons between consecutive releases can be extremely helpful in determining if fixes or architectural changes have had any measurable impact on response times.

All of this can be done automatically by a Test Harness by providing hooks to the custom test dll to provide such information and then, if it has been provided, generating a report.

### 8. Provide Client/Server Functionality

Today’s testing environment often requires the ability to perform tests across system boundaries. The client portion of the Test Harness may reside on one machine, the test dll on a second and the target on a third. Any Test Harness, to be useful, must be able to fit such a test requirement. Fortunately today’s software tools and operating systems make such an installation relatively painless. Microsoft provides both web services and .NET remoting in the latest OS versions as well as COM and COM + in earlier versions. Java provides remoting software and Unix also provides the ability to test remotely.

### 9. Automated Installation

One of the advantages of a Test Harness is that it provides an environment in which testing may easily be done by QA personnel who are not themselves developers. A standard GUI or command line interface allows them to know how to run tests, how to control test execution and to assess test results. However none of this is of much use if they cannot install the Test Harness and associated software and thus it is important to provide automated test installs. Since often testing is done on multiple platforms QA labs are often in the business of copying standard images onto systems and then running tests on them. Since these images likely cannot contain the test software, that software has to be able to be installed with a minimum of trouble. The easiest way to do this is through automated installs. Indeed, since most software packages are delivered through such installs QA personnel are used to using them and providing a test in the same manner makes the entire process more consistent.

The above presents a reasonably complete list of general required functionality although some testing environments may have additional specific needs. The question now turns to how to implement this tool so that it is portable between projects.

## 3. Portable Construction

No matter how useful any tool may be for a particular project it is unlikely to be used in another project if there is too much work needed to make it suitable for testing for that project. At the heart and soul of such portability is the need to develop a single format for both inputs and outputs. Standardizing inputs provides a template for the test developer for the creation of data for a new project. Standardizing outputs allows the Test Harness to produce the necessary reports from the logs and statistics and thus the test developer has only to concentrate on writing the custom code for testing. Code templates make it easy to create custom tests that live and work properly in the confines of a Test Harness.

## 3.1 Standardize Data File(s)

The data file is probably the most important item to standardize. If its format is known the Test Harness can load and parse the data, verify that it is in correct format and hand one test case at a time to the engine. The engine can handle the known overhead information calling the test dll/assembly only when necessary and keep control information about how testing is proceeding. Then only the necessary test cases are actually called and the test dll can use a pre-existing set of classes to parse the data and extract the necessary information. Given its current common use in the industry xml appears to be the most useful data format and it can be successfully used even when the data for tests actually resides in a database. While many different formats may serve as templates Appendix 1 contains one possible sample. A variant on this format is in use at Intuit for all of its automated testing. It has been general enough to serve for a multitude of projects.

## 3.2 Standardize Configuration Information

Configuration files allow a great deal of flexibility in defining how tests will run. The information they provide can be accessed and used to control what the test will do and how it will do it, can be translated as data input or data modification and can be updated as conditions change and force test conformance to new environments without worrying about re-building.

Two pieces of software are generally running during a test. The Test Harness itself and the custom dll/assembly that does the actual testing. Both should have configuration files.

## 3.3 Standardize Log File

The format of log files should also be standardized enough to insure that the start and end of test case log entries are done by the engine so that the engine and/or front end is able to parse through the data and extract subsets suitable for reports. In our QA organization we provide the ability to extract any subset of a log by test case result so we can provide reports of only passes, only fails, only blocked and so on. Log entries themselves are often relatively free form although some standard entries are provided for special handling. One of these in particular deserves some special discussion.

One of the advantages of automated testing, especially at the API level, is the ability to run many test cases in a relatively short time. It is not unusual for automated tests to be able to execute thousands of test cases in relatively short time periods and thus it can become a problem to compare test results. Usually this is done by comparing the failures in the logs. The basic assumption is that if the same test cases failed, then the test ran as before. Unfortunately this is not always the case. If the same test case failed as before, but failed in a different way, it is not the same failure and must be reported as a new defect. How can we automate the test log comparisons?

One way is to mark each failure in the log with a special token unique to that failure. While this may sound complex it is, in reality, a relatively simple process. In my tests I label each possible error condition that will result in a Fail with a 2-letter code starting with AA and ending with ZZ. If the failure can be produced by any of a series of actions such as comparing data fields in a record with their equivalent in a database, then the token is expanded to include the field being compared. If the failure results from an exception, then the token is expanded to include either the error number or the error message. If the failure comes from a single field comparison, then I may expand it by including both the expected and actual field contents in the log file. These entries are then extracted from the log and form a set of failure information for that test run. New failure sets can then be compared with old ones and it is possible to tell if failures have changed.

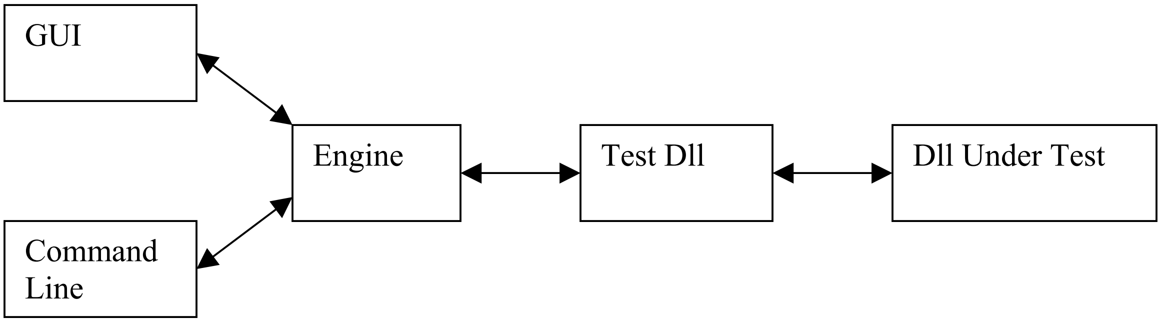
Other such approaches may be used. The important thing is to provide a standard way of doing this so tools can be developed within the Test Harness and used to help ease the testing effort.

Another approach to providing portability in data file creation is through the use of pseudo-data (or meta-data) entries. Such entries, if standardized, can be used in multiple tests and the work to replace them during actual test execution with the real data can be done by the Test Harness. We use this process at Intuit through the use of well-defined data tokens that have specific meanings. All domain experts who work with the data files know these token values and how they work and they can be (and are) used across tests. A list of some of these “tokens” is given in Appendix 2.

## 3.4 Isolate Test Dependent Functionality in the custom dll

Test Harness functionality should be completely isolated from testing functionality. The Test Harness has a specific set of functionality to implement, but none of that functionality should involve any testing. The Test Harness may read and verify the format of the data file, access and pass along configuration information, log entries and other such requests, but should know nothing about the dll/assembly being tested. That is, it verifies the format of the data, not the data itself. That responsibility lies with the test dll alone since it alone knows what specific data entries mean.

Typically the architecture of a Test Harness is something like the following



Separating functionality provides many advantages

- Since the Test Harness is separate from the test functionality it can provide functionality for any test environment.
- Separating the GUI functionality from that of the engine allow the construction of a command line interface paralleling the GUI functionality
- Separating the GUI/Command Line from the engine and that from the test dll allows remoting to be used for any interface.

## 4. How Do We Make It User Friendly?

A Test Harness is only useful if it is used and only used if it is easy to use. For the test developer it should provide a simple test template and library and, since most QA organizations would like automated tests to be able to be run by everyone - domain experts, testers and contractors as well as by test developers, it should provide a familiar and easy to use visual interface, GUI or command line.

From the test developer's prospective test execution is relatively simple. In general there are only five methods that need to be provided. One to be run prior to any test execution (test set setup), one after all test cases have been run (test set cleanup or "tear-down") and then one prior to each test case, one after each test case and one for the actual test case work. Of course much other work needs to be done during a test case, but these five methods constitute the complete set of needed Test Harness – test dll interface methods. Since the defined interface is relatively simple the Test Harness can, and should, provide a template implementing this interface that the test developer can then use as a starting point for automated tests. For SD&S these methods are named preTestSetProcessing, preTestCaseProcessing, testCaseProcessing, postTestCaseProcessing and postTestSetProcessing. It also provides a complete set of library routines to assist in xml handling, database access, logging and other normal processing and validation needs.

For the Tester the single most important thing the harness can do is provide an easily understood and easily used GUI. The GUI should look and behave like any normal piece of software. If it has the same look-and-feel as any other normal piece of software it is more “approachable”.

Provide the simplest setup possible. If configuration information is needed, determine if it can be automatically generated. Create all necessary folders during the installation. Make references relative rather than absolute. Use any existing environment variables to determine where needed resources are.

Provide automatic installation of the tests. It may be an install package such as the Microsoft Deployment Package, InstallShield, Wise or a simple zip exe. The easier it is to use, the more likely it will be used.

Other areas that help testers: Supply visual confirmation of progress. Display test cases executed, passes, fails, etc. Provide a way to stop the test without terminating the process. Provide a way to run multiple sequential tests rather than a single test set. Provide for scheduled test execution. And make the functionality of a GUI an extension of the functionality of a command line interface.

While Managers may not directly use tools like a Test Harness, the output from such tools can form a core of information they need to make appropriate decisions about project timelines and progress. Simple reports such as test run summaries can often be very helpful in determining how well a project is progressing.

Application developers, whose products are actually under test, benefit from log files containing detailed information about both successes and failures. Success entries, if sufficiently detailed, give them information about exactly what type of data is being properly processed and failures, of course, give them detailed information about what is not being handled properly. Thus they often need far more detailed test information than the similar reports given to Managers.

# Appendix A: Data File Format

```
<TestSuite>
  <TestCase Nbr="1" Id="Sample-A1B2">
    <Description>description 1</Description>
    <PrevRef value=”none” />
    <Dependencies value=”none” />
    <Untested state="false" />
    <Blocked state="false" />
    <Attributes>
      <Attribute name="" value="" />
    </Attributes>
    <Obj name="object1" clazz="" instance="" depends="">
      <Mthd name="method1">
        <Params>
          <Param name="param1" type="string">value1</Param>
          <Param name="param2" type="integer">5</Param>
        </Params>
        <Exception value="false" />
        <Return value=”void” />
      </Mthd>
    </Obj>
    <Obj name="object2" clazz="" instance="" depends="">
      <Mthd name="method2">
        <Params>
          <Param name="param1" type="boolean">True</Param>
          <Param name="param2" type="long">125</Param>
          <Param name=”param3” type=”string”>Null</Param>
        </Params>
        <Exception value="false" />
        <Return value=”true” />
      </Mthd>
    </Obj>
    <Obj name="object1" clazz="" instance="" depends="">
      <Mthd name="method3">
        <Params>
        </Params>
        <Exception value="true" />
        <Return value=”false” />
      </Mthd>
    </Obj>
```

# Appendix B: Tokens

While this is not a complete set of tokens it does provide a sample of what is used by SD&S.

**#T:Value** – translate the specified value based on entry in configuration file. This translation is automatically done by the Test Harness before the data is passed to the test dll/assembly

**#R:A:minLen:maxLen:U** – generate a random uppercase alphabetic string of random length with minimum minLen and maximum maxLen

**#R:N:minLen:maxLen** – generate a random numeric string of random length with minimum minLen and maximum maxLen

**#R:AN:minLen:maxLen:L** – generate a random lowercase alphanumeric string of random length with minimum minLen and maximum maxLen

**#R:Num:minVal:maxVal** – generate a random number with value between minVal and maxVal

**#Min:A:FieldName** – generate an alphabetic string of minimum length for the specified field name. Length information for the specified field must exist in a file (or a database reference must exist)

**#Max:A:FieldName** – generate an alphabetic string of maximum length for the specified field name. Length information for the specified field must exist in a file (or a database reference must exist)

**#XMin:A:FieldName** – generate an alphabetic string of one character less than the minimum length for the specified field name. Length information for the specified field must exist in a file (or a database reference must exist)

**#XMax:A:FieldName** – generate an alphabetic string of one character more than the maximum length for the specified field name. Length information for the specified field must exist in a file (or a database reference must exist)

**#Prev** – return the object generated by the previous test case.

**#Prev:paramName** – return the value from the previous test case for the specified parameter name

**#X** – return the object generated by the specified test case ID

**#X:paramName** – return the value from the specified test case for the specified parameter name.