



SEPTEMBER 14, 2016

**TEST HARNESS**  
**OPERATIONAL CONCEPT DOCUMENT**

KUNAL PALIWAL  
INSTRUCTOR: JIM FAWCETT  
SUID: 961472585

### CONTENTS:

1. Execution Summary.....	1
2. Introduction.....	2
3. Use Cases.....	5
4. Modular Structure.....	8
Tester	
GUI	
Test execution	
XML parsing	
Loader	
Display	
Logger	
Indexer	
Status	
Client Query	
XML security library	
Exception handler	
Authorization	
5. Activity Diagram.....	12
6. Critical Issues.....	17
7. Conclusions.....	18
8. Appendix.....	19
9. References.....	20
10. Diagrams.....	
Activity diagram.....	12
Relationship diagram.....	16
Package diagram.....	8

## 2. Test Harness

### 1. EXECUTIVE SUMMARY:

This document constitutes the Operational Concept of a Test Harness. The first company founded to provide software products and services was Computer Usage Company in 1955. Since then, software has become an intrinsic part of business over the last decade. Business in almost all sectors depends on it, to aid in the development, marketing and support of its product and services. According to industry analyst Gartner, the size of the worldwide software industry in 2013 was US\$407.3 billion, an increase of 4.8% over 2012. As in past years, the largest four software vendors were Microsoft, Oracle Corporation, IBM, and SAP respectively.

This has led to reliability becoming an important factor alongside costs while designing software. Most bugs arise from mistakes and errors made in either a program's source code or its design, or in components and operating system used by such programs. For example, bugs in code that controls the Therac-25 radiation therapy machine were directly responsible for patient deaths in the 1980s. In 1996, the European Space Agency's US\$1 billion prototype Ariane 5 rocket had to be destroyed less than a minute after launch due to a bug in the on-board guidance computer program. This makes testing an important tool to prevent such unexpected outcomes as well as increasing reliability of the software. Many critical issues will be discussed in this document which will summarize the key issues and their solutions in the design of the test harness. This system will be implemented in C# using the facilities of .Net framework class libraries and Visual Studio 2015. Test harness will be divided into separate modules in which each module performs specific tasks. The project should provide capability to:

- Execute test requests after obtaining them from a XML file.
- Handle each test request by isolating it with an app domain.
- Create a log file with time stamp for each activity that takes place in the test harness.
- Support all client queries from the log file.
- Fetch DLL files from the specified directory and inject it to the app domain.
- Assert module which displays the status of the test harness.

Important issues to be solved during implementation of this system are:

- Unloading exception – solved by making sure unloading happens b4 loading a new DLL.
- Stub doesn't work – solved by providing exception
- App domain plugging
- Third party dependencies – solved by accessing meta data

Thus Test Harness can handle almost all kinds of tests and such a system can be implemented using the available resources provided by the .Net framework.

## 3. Test Harness

### 2. INTRODUCTION:

Usage of Test Harness leads to increased productivity due to automation of the testing process, increased quality of all components and repeatability of test runs. Automation, test management, data, conditions and results become an integral part of the Test Harness. This makes it suitable for testing functions regardless of their complexity.

#### 2.1 APPLICATION OBLIGATIONS:

The main responsibility of a Test harness is to handle test requests such that it can offer performance, flexibility and security. The primary obligations of the system would be:

- Support any kind of integration tests performed by developers, QA and managers.
- Should support a display of all DLL files available in the specified file location.
- Should be capable of loading and executing that simulated test library and displaying the results.

#### 2.2 ORGANIZING PRINCIPLES:

The organizing principles are to demonstrate key functionalities of a test harness through File Manager, Loader, App domain creation and logger while other functionalities besides test processing are handled by other modules by communicating with the File Manager.

#### 2.3 KEY ARCHITECTURAL IDEAS:

The key idea in Test Harness architecture is to perform tests on user's code. Each test driver and user's code it will be testing is built as dynamic link libraries (DLL) which in turn is loaded by an executive. In this application, we will be using C# language with .NET framework 4.6 and Visual Studio 2015. To handle test requests, we will use file manager. Test request will be run in the App domain which isolates it from Test Harness processing. Objects in the same application domain communicate directly. Since we use a child app domain for running test executions an unhandled exception in test execution will not affect Test Harness processing. We will be using reflection class to systematically call a discoverable set of API's defined on a class, to ensure a high level of code coverage in a test suite.

## 4. Test Harness

### 3. USE CASES:

#### 3.1 CODE DEVELOPERS –

Developer's will use Test harness to perform integration tests and construction tests. "Construction tests are implemented by writing few lines of code and continuously testing them, until class meets all its requirements. Integration tests are aggregated over the course of development of a large project and are run frequently to ensure that new code doesn't break the existing baseline." – Jim Fawcett.

Code developers can get the output or the status of the test harness by performing simple queries and with the help of Assertion package.

#### 3.2 TA AND INSTRUCTOR–

Teaching Assistants and Instructor will perform testing tasks on this system. They will formally witness that all the requirements of the project are fulfilled. They are also responsible for identifying faults in the system, if there are any, and to report them to the developers of the system.

#### 3.3 PROJECT MANAGER–

Output from the log details can be shared in the form of an XML file. This information can be used to write detailed reports by project manager.

#### 3.4 QA–

QA's can perform quality analysis by running all test cases and checking to see if there are any unhandled exceptions or interrupts. Execution time can be saved to check performance. This is a final demonstration of a large software system, and is used to demonstrate that the system meets all of its obligations to the customer as laid out in a system specification. "Qualification testing is legalistic, not technical. It should be very organized, detailed, and boring. Qualification testing uses both test automation, like regression tests, but also uses hands-on probing and demonstrating to the customer the system's operational details." – Jim Fawcett.

#### 3.5 OTHER APPLICATIONS:

##### 3.5.1 EXTENDING TO FUTURE PROJECTS:

- We will further explore the concept of accessing and using a remote Test Harness. We will further improve our GUI and work towards finding ways to effectively communicate between local clients and Remote Test Harness. We will be implementing a message-passing communication system using Windows Communication Foundation (WCF).

## 5. Test Harness

### 3.5.2 PRESENT DAY –

- Present day scenario: Today software organizations are incorporating multi-tier architecture, OOD and programming different kinds of interfaces, client server and distributed applications, data communication and huge relational database, which all contribute to exponential growth in software complexity. Test developers are unable to duplicate these complexities before a project progresses, making it impossible for them to resolve the problem they present in advance. Innovation continues to improve tech and add complexity to soft projects. Tools can't predict these developments. For these reasons, it's imperative that each software organization develops its own tool to solve its unique testing issues.
- Scope of a fully automated test harness: In order to assure quality products within the aggressive timeschedule, the organization must develop an effective test approach with a scope in focus. Determining a testing scope is critical to the success of a software project. It is always desirable to test the software thoroughly, but that's not always possible due to timeline and budget constraints. Without a fully automated testing tool, test engineers usually determine critical requirements and high-risk areas by analyzing the requirements that are most important to the customers and the areas within the application that receive the most user focus. Decisions are often made to do no testing at all in non-critical areas, which can cause problems and even disasters in the future and turn these noncritical areas into critical ones. Thus, the scope of a fully automated testing tool should be every part of the code, all the code should be tested continuously day and night until most of the defects are found and the product is released. After the testing scope is determined, we usually need to plan the testing phases, including the timeline of tests to be performed in both the pre and post release stages.
- Test harness can be used testing wide range of applications. One such example is for testing embedded software for companies which supply large diesel engines for ship propulsion systems, stationary power supply and rail traction. A marine diesel engine is composed of several parts ranging from mechanical, electrical to electronic. In recent years, the working of a marine diesel engine is controlled using electronic control unit (ECU). This is an electronic device, basically computers, and some networks that reads several sensors and uses the information to control the actuators for example fuel injection in a diesel engine, allowing greater fuel efficiency, better power and responsiveness, and much lower pollution levels than earlier generations of engines. Test Harness can perform tests on such software's to verify that it satisfies specified requirements and to detect errors.

## 6. Test Harness

### 3.5.3 FUTURE SCOPE:

There have been many proposed test harness frameworks. Here is a future usage which can be considered with the available resources:

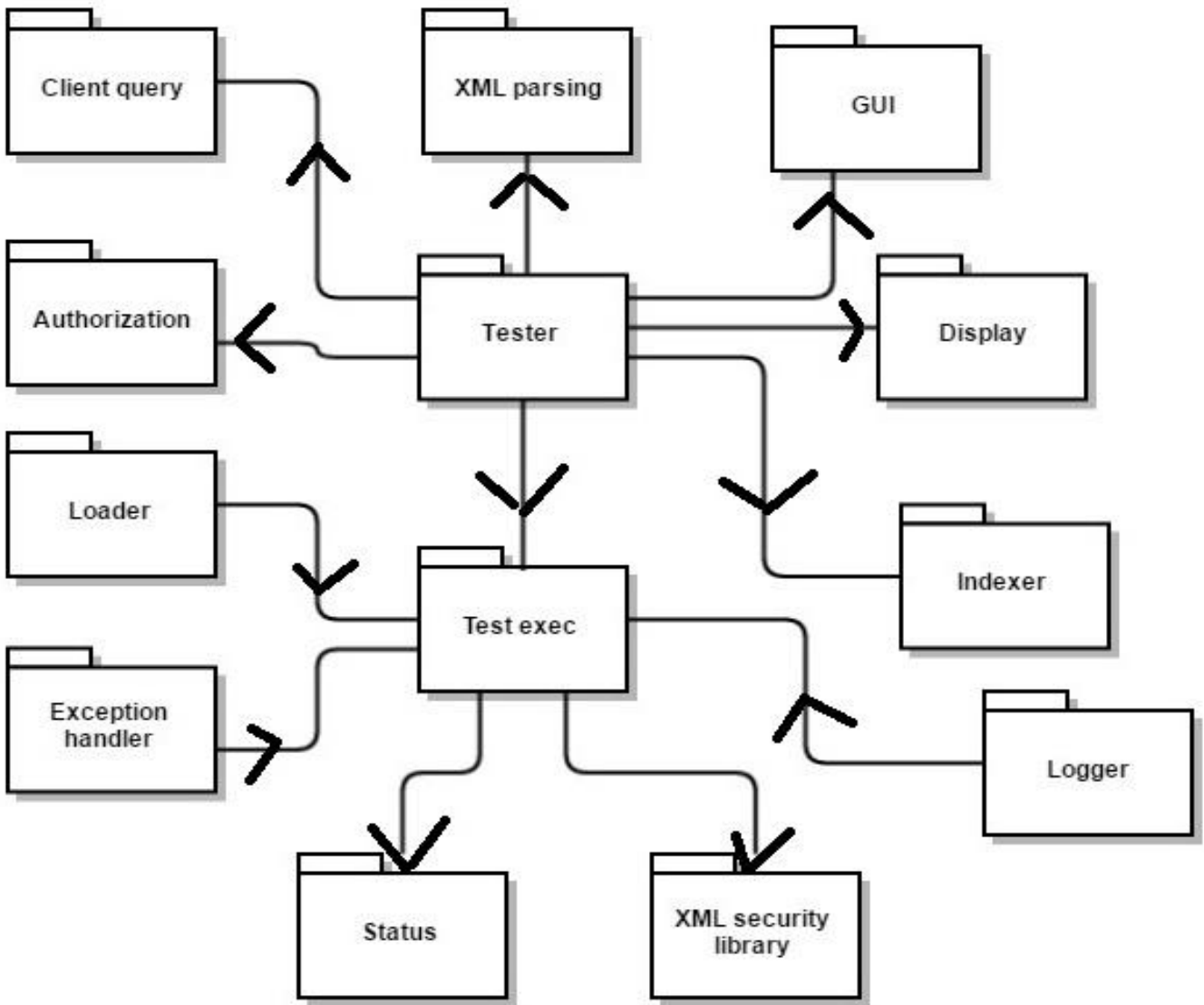
Web service testing:

- The proposed test harness is a configurable XML-based framework; it can be used for the functional testing of Web services and to measure the performance of Web services at runtime. The test harness will support performance testing, load testing, stress testing, and CHO (continuous hours of operation) testing for Web services. The test harness is designed in such a way that it treats each 'test' as a task to be executed. Each task can accomplish different functions, depending on the configuration parameters it is supplied with. The configuration parameters for individual tasks are provided by the XML files. So, the test harness starts with the loading of configuration files, getting the tasks and its parameters out of configuration file, and finally running the task with the desired properties. The task in this process will invoke the Web service to be tested, get the results, and if the task is doing a performance test, it will measure the execution time, also.
- The harness will have options such as simulating multiple users using threads for loading a Web service. Simulation of multiple users will be useful for performance/stress testing to get real-time data. The harness can also have an option of setting the number of iterations (and sleep time in between) for tests to average out the performance results. Depending on the type of Web service, the harness will use the appropriate XML file for configuration. The tester can provide the list of test groups (tests to be run in one group), or individual tests to be run for a particular Web service in another XML file. This tests list file will also have the pre- and post-conditions to check the success or failure of the test. In either case, the test harness will report the successes and failures with a report manager. The report manager comes with an option to produce output on a file format, console, or both.

## 7. Test Harness

### 4. MODULAR STRUCTURE:

Package diagram and explanation of each package:



Package Diagram



## 8. Test Harness

### 4.1 TESTER:

This is where the main method of the system is placed. This package will take input as an XML file and decode it using XML parser package. Test execution for each request will be done in a child app domain that isolates test processing from test harness processing. This package will also use following packages to implement required functionalities:

- Display package to show status of the test harness.
- Loader package used to inject application domain with the test driver.
- Indexer to queue the test requests which are then executed serially in a dequeued order.
- Query processor is used to support client queries from the log storage.

### 4.2 GUI:

Here we can use a simple interface that allows client interaction with the most important parts of the test harness functioning. These include:

- List of all test cases / names of the DLL files obtained after parsing the XML file.
- Listing out all the passed and failed test cases.
- A client query interface.
- Status display that shows the current state of the test harness. It can be used to display output and input values.
- Report log is also an important part of GUI and is very useful for project managers and QA in performance testing.

### 4.3 TEXT EXECUTION:

This is the package which executes each test request in a sequential order. After completion of each test case, a new test case is loaded until all of them are executed. This package interacts with the tester, logger, loader and other important packages to perform all of its functions. This process takes place in a separate child domain from the primary domain. This makes sure that any test execution interrupt doesn't stop the working of the entire test harness. Each test request is loaded with a test driver and is handled in isolation. This has many other benefits such as security and versioning. After completion of each test request the app domain is unloaded. This is key in order to maintain the processing speed or performance of the test harness.

App Domain provides these key benefits:

- Isolation.
- Control of type visibility.

## 9. Test Harness

- Fine-grained configuration of loading, versioning, remoting, user settings.
- Programmatic control of Library loading and unloading.
- Marshaling services to access type instances in another domain.

### 4.4 XML PARSING:

This package is used to read the input from the XML file by parsing it and then passing it to the test exec package in a readable format. This is an easy way for the client to request for DLL files to be loaded by naming them.

### 4.6 DISPLAY:

This package is a display package which searches the directory tree, rooted at a specified path, and displays the name of the DLL files encountered. Display package will interact with client querypackage to show the output of requests made on console. For example, if a command is run to display the name of the author, test name, time stamp or test driver output will show as below:

Example of some client queries are:

- Display current state: Running tests.
- Display author name: Jim Fawcett.
- Display time stamp: 17<sup>th</sup> September 2016.

### 4.7 LOGGER:

This package supports client queries about the test requests from the log storage. Catches every activity performed by the test harness with a timestamp. It logs the current and before states of the test harness. This information is used to perform client queries and regression testing. Logging makes it a much easier process. Ex; display total test requests: 5.

### 4.8 INDEXER:

This package handles queuing of the test requests. These requests are executed serially in a dequeuer order. After every test run an activity log is made which will show the timestamps and hence we can find out total execution time of each test run. This is helpful while doing performance tests. Indexer makes use of queue data structure and follows the first in first out approach.

### 4.9 LOADER:

## 10. Test Harness

Loader plays an important role during test executions. It injects child domain with the test driver. This way test harness processing is kept separate from test execution. The primary app domain only knows about the loader type not all testing types. Each test driver has a test stub which is used to check if the driver is working or not. After making sure it's working, the DLL file is loaded to the app domain.

### 4.10 STATUS:

This package is used to display output after finishing all the tests. The status can be either pass/fail or a general summary of the test results can be displayed in the GUI.

### 4.11 CLIENT QUERY:

This package is used to perform client queries. Client queries are performed here using the log files that have been stored after completion of the tests. Client requests may vary from requesting a display of current test case being executed or just the name of the user and test results.

### 4.12 EXCEPTION HANDLER:

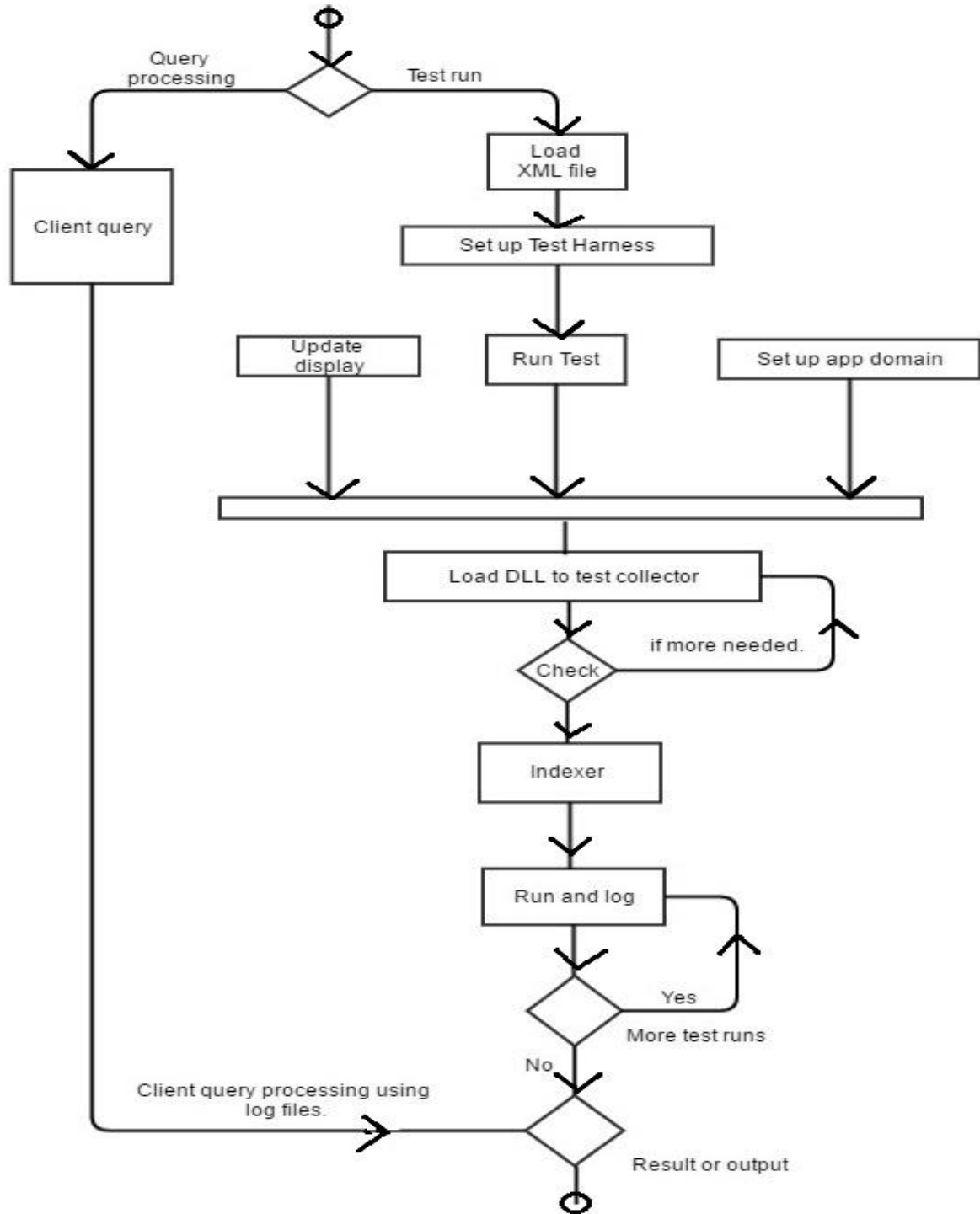
Exceptions are things that happen which are not expected in the system as a part of normal process. An exception tends to occur in large systems and if the system does not handle such exceptions, it can lead to system failure. To handle this kind of exception, we have introduced an Exception Handler package which will control the process flow if any exceptions are detected. Typically, whenever exceptions are detected, control is passed to a special exception handling method from that point or it will return to the last known state of the system. This package will interact with all other packages through Tester package.

### 4.13 XML SECURITY LIBRARY:

This package is used to ensure more security for test outputs. This library supports major XML security standards, including XML Signature, XML Encryption, Canonical XML, and Exclusive Canonical XML, usually an XML file, and a code file to read properties from that XML file which will include user credentials and privileges.

## 5- ACTIVITY DIAGRAM:

## 11. Test Harness



Activity diagram

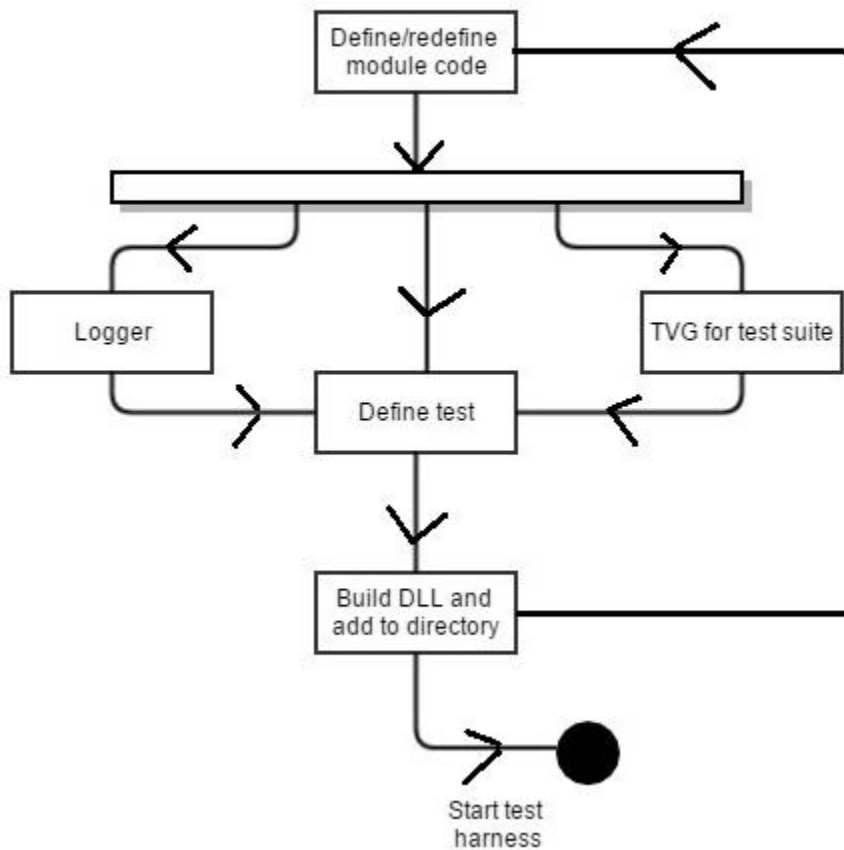
## 12. Test Harness

### 5.1- READ INPUT FROM COMMAND LINE AND GET TEST CASES:

For constructing test requests initially, the user feeds XML file as input to the application which contains information on test name, author name, code to be tested and test driver associated with it. All test cases are fed to the indexer where they will be queued and then later will be executed serially in de que order.

### 5.2 – REDEFINE TEST CODE AND BUILD DLL FILE:

This is done when before starting the test harness and code is redefined in case of failure of test or a need to modify. The test is redefined and built in to a DLL file which is then loaded to the test harness.



Activity diagram

## 13. Test Harness

### 5.3- START TEST REQUEST PROCESSING:

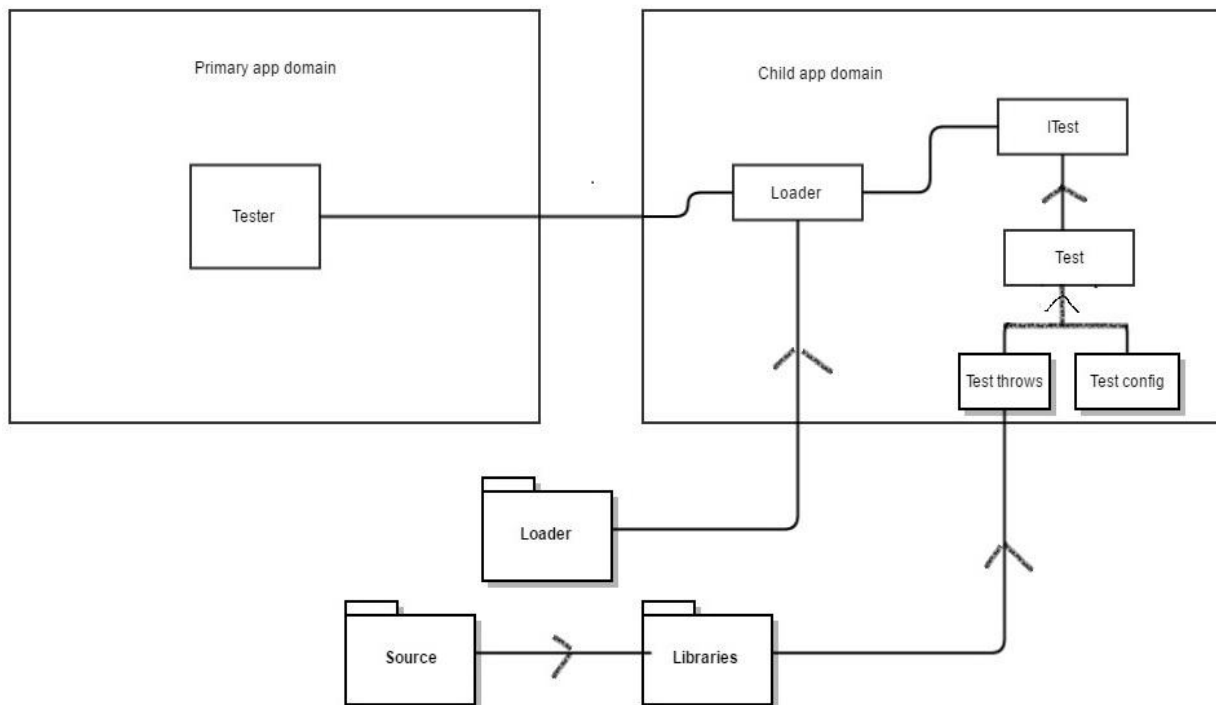
This task is split into three parts:

- Display.
- Set up app domain.
- Start test execution.

Each testing process will be isolated to a child app domain which executes it using the test driver it derives from the loader. Each test driver derives from an ITest interface that declares a method test () which takes no arguments and returns the test pass status(Boolean true/false). It also has a getLog () function that returns a string representation of the log. Each DLL file can have structured meta-data listings of all the dependencies of the test run. The ITest interface might look something like this:

```
public interface ITest {private string testDriverName;  
private string author;  
private bool test(){}; private string getLog(){};}
```

### 5.3 APP DOMAIN AND CHILD APP DOMAIN FUNCTIONING:(Relationship diagram)



## 14. Test Harness

The primary domain coerces a child domain to load a “loader” into the child domain and marshal back a reference to it. The primary domain then, using the loader reference, instructs it to load a collection of test assemblies for processing. This results in the following:

- The primary App Domain only knows about the loader type, not all the testing types.
- The tester or test manager, running in the Primary App Domain is isolated from failures of the test and tested code.

Also, each application domain in a process can be independently configured either programmatically or with a configuration file (Test configuration. file). After finishing the test execution, the DLL file is unloaded from the app domain and the next test case is added.

### 5.4 EVENT TRACKING IN THE APP DOMAIN AND LOGGING:

The AppDomain Type supports a handful of events that allow interested parties to be notified of significant conditions in a running program. These include:

- Assembly load
- Assembly resolve,
- Type resolve,
- Resource resolve,
- Domain unload,
- Process exit
- Unhandled exception.

A summary of these events can be logged and stored with time stamp. Clients can access these log files to perform simple queries.

### 5.5 INSPECTION/ RESULT:

This is a key area, where developers can use the status of the test execution to check if the code has any unhandled exceptions. QA can perform performance tests by comparing time taken for each test executions. The response after executing the tests can be displayed in green along with the test request which will be displayed in yellow. If there is a failure or modification client has to redefine module code and take care of the cause for the failed test.

### 6. CRITICAL ISSUES AND ANALYSIS

This section discusses about critical issues which are of crucial importance in relation to successful development of the system. If the required actions to address the issues are not taken, it might lead to development of defected system. Some of the critical issues and their potential solutions are discussed below.

- **Automated test case generation** is a critical issue; for any real program manually automated tests will never cover enough cases. In addition, they are tedious to prepare. Hence the work on automatic test case generation, which tries to produce as many test cases as possible, typically working from specification only (black-box). Manual tests which benefit from human insight remain indispensable. Automatic test generation needs a strategy for selecting test cases. It requires more initial development time, higher level of skill in team members, increased tooling needs (test runners and framework).

**Solution:** These problems can be minimized by outlining key issues and educating the key members for a test-focus approach. Such automated testing requires a solid multi-structure to ensure for example that if a test run causes a crash the testing process doesn't also crash but records the problem and moves on to the next test.

- **Additional deployment costs:** We can't just run unit-tests as a developer on our own machine. With automated tests, we want to execute them on commits from others at some central place to find out when someone broke our work. This is nice, but also needs to be set up and maintained.
- **Exceptions** can cause test execution to fail, hence should be handled appropriately;

#### 1) Recovery from failure:

Checking validity of application's data integrity after restarting an application. For the large systems, the system is supposed to verify and validate the inputs as well as it needs to handle the exceptions if any process results in undefined state which can eventually crash the whole system. Solution: For handling recovery scenarios, the application has one package – ExceptionHandler – which will handle all the exceptions that are resulted due to undefined reasons such as wrong type of input, segmentation faults, invalid syntax of queries and many more.

#### 2) Other cases:

When DLL file is not available in the specified directory, it can lead to an error which has to be resolved by providing an exception.



## 16. Test Harness

- **Programming Language Binding:** One of the critical issues is how other programming languages can connect with this system successfully to perform tests. The solution is to create a Language interface which will work as communicator between all the components of the test harness. This component can provide platform independent framework to test any code.
- **Data security & versioning:** Security is provided by isolating the test execution from the test harness process in a child app domain. In the issue of versioning, it is always better to have only one version of the code under test available in the repository. This is done by updating and storing the most recent DLL file is later fetched from the repository.
- **Test stub for test driver:** This has to be done before building a DLL file. This is done to make sure that the test driver works.
- **Meeting obligations:** How can we test all the requirements using one input XML file on console display? Solution: Demo client package will handle this issue by careful automation of series of tests and supported by processing the output on console using a display package.

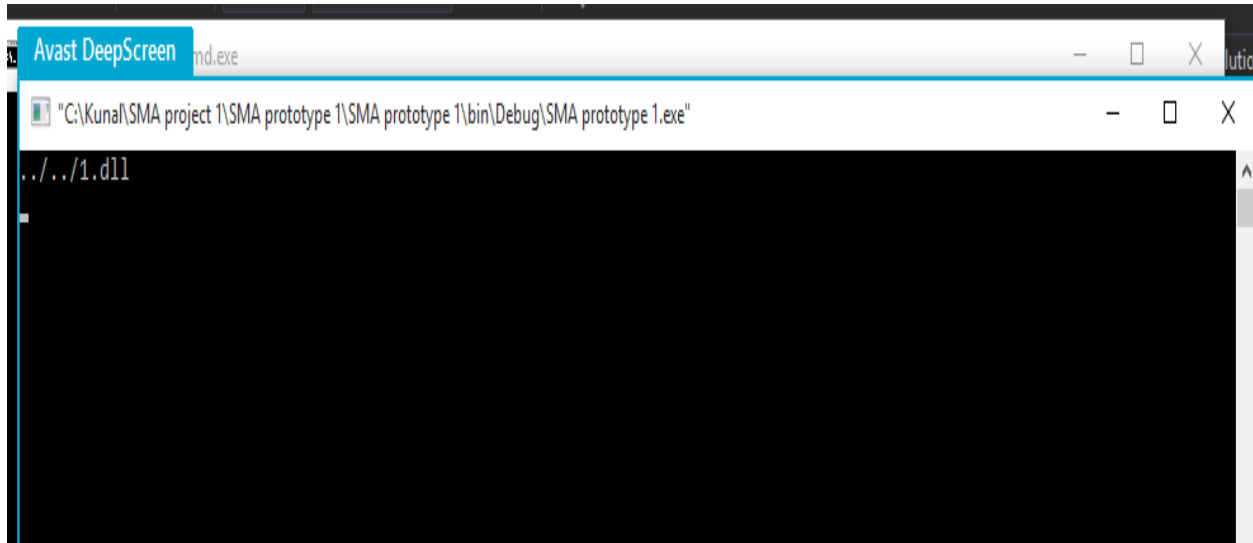
## 7. CONCLUSION:

In conclusion, Test harness can be used to perform tests independent of their complexity, input type with a simple interface. It provides users reliability, flexibility and safety to perform tests while developing code. This OCD lists out the different users and how they can use the test harness. We have discussed different use cases where Test harness will be most useful and the future scope of this design. The system's cohesive packages are partitioned in a manner with which they can interact with each other to perform all tasks defined in requirements. We also covered all the activities being carried out by the system using two activity diagrams in section 5. It describes how the events take place in the system and makes it easier to understand the system. We also discussed few critical issues which can result in serious design flaw if not provided and their proper solutions mentioned in section 7. It can be ensured from this OCD that the system can be developed in considerable period of time and with available resources

### 8. APPENDIX:

#### 8.1 FILE MANAGER PACKAGE

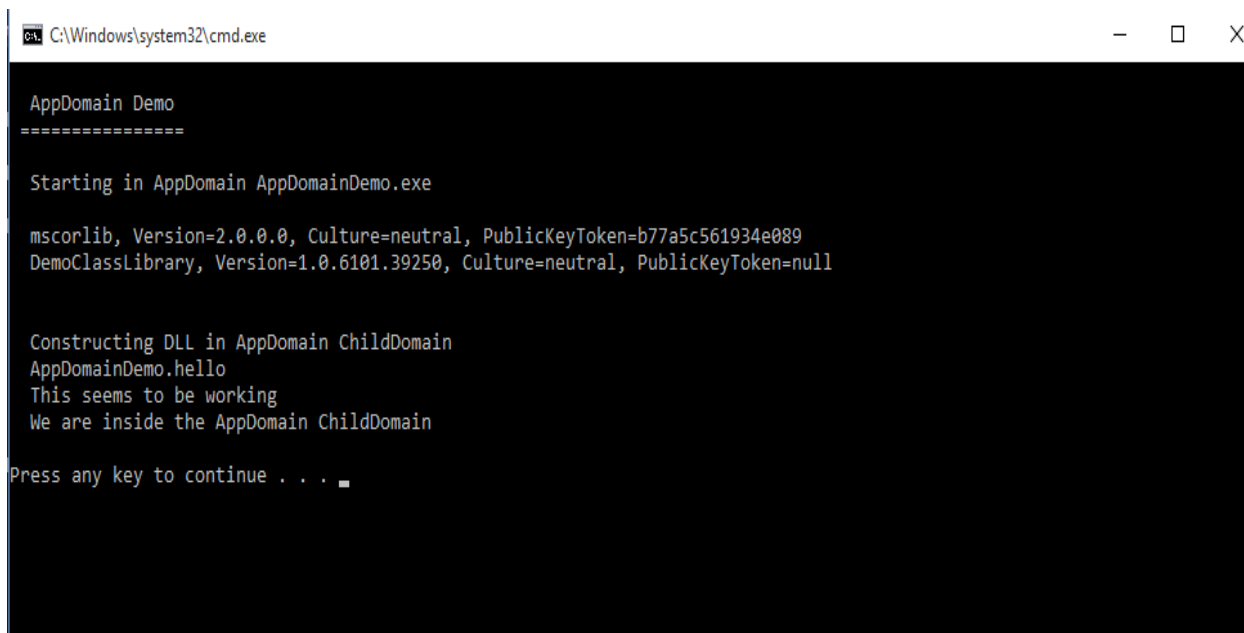
- Description: Package that searches a directory tree, rooted at specified path and displays the name of the DLL files encountered.
- Input is the specified directory path and file type. Name of the DLL files encountered are listed in the output.
- Using directory class we are able to get files by calling method `GetFiles(string path, string extension)`. This will return all the files with the specified extension type in the specified directory/path. We can create a new method called `DisplayFiles` method which can be used to display (by using `Console.WriteLine(name)`) for each DLL file returned by calling the `GetFiles` method.
- The method call will look something like this:  
`string[] dllFiles = Directory.GetFiles(path, "*.dll")`



## 18. Test Harness

### 8.2 CHILD APP DOMAIN DEMO:

- Loads a simulated test library, executes it and displays the result.
- Input is the path to the DLL file which is then executed in an app domain and its result is displayed.
- From this demo we can gain insight towards the inner functioning of the test harness where we create an app domain which is injected with a DLL file and later executed. The output is displayed as test result.



```
C:\Windows\system32\cmd.exe

AppDomain Demo
=====

Starting in AppDomain AppDomainDemo.exe

mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
DemoClassLibrary, Version=1.0.6101.39250, Culture=neutral, PublicKeyToken=null

Constructing DLL in AppDomain ChildDomain
AppDomainDemo.hello
This seems to be working
We are inside the AppDomain ChildDomain

Press any key to continue . . .
```

### 9. REFERENCES:

- <http://www.ecs.syr.edu/faculty/fawcett/handouts/>
- <http://www.ecs.syr.edu/faculty/fawcett/handouts/Webpages/BlogTesting.htm>
- [www.developer.com](http://www.developer.com)
- <http://stackoverflow.com/>
- <https://en.wikipedia.org/wiki/Test>