CSE-681|Software Modeling and Analysis

Project #1- Operational Concept Document (OCD)

TEST HARNESS

Instructor- Dr. Jim Fawcett

SAHIL SHAH (SUID- 654427435)

14th September, 2016.

Table of Contents

CSE-681|Software Modeling And Analysis 1

1. Executive Summary 4

2. Introduction 5

2.1. Automation Testing 5

2.2 Integration Testing 6

2.3 Organization 7

2.3.1 Organizing Test Data 7

2.3.2 Integration testing organization 7

2.4 Prototype Implication 7

2.5 Architectural Ideas for Test Harness 7

3. Users 9

3.1 Software architect 9

3.2 Quality Assurance Manager 9

3.3 Teachers Assistants 9

3.4 Developers 9

3.5 Test Manger 10

3.6 Testers 10

3.7 Manager 10

3.8 Uses 11

3.9 Impact On Design 11

4. Application Activity 12

4.2 Activity Diagram for Testing Process only 14

4.3 Flow of data: 16

4.4 Queuing and testing of data 17

5. Partitions 18

6. Critical Issues 20

6.1 Easy To Use 20

6.2 Performance 20

6.3 Deadlock Situations 20

6.4 Managing Regression Testing 20

6.5 Overloading 21

6.6 When to stop testing? 21

6.7 Service Level Agreement 21

6.8 Catching Exceptions 22
6.9 Security 22
7. Conclusion 22
8. Prototype 23
8.1 File Manager package that searches a directory tree, rooted at a specified path, and displays the names of all DLL files encountered: 23
8.2 Child App-Domain demo that loads a simulated test library, executes it, and displays the results. 24
9. References 25

1. Executive Summary

Today in the world of heavy loaded applications, testing becomes an important factor to reduce the bugs and make an application run perfectly. Manual testing for such big projects is outdated as concurrently testing hundreds of thousands of cases concurrently is a slow and tedious task to be carried out manually. Furthermore it may lead to faulty results. Hence, now we build the new age automated testing environment known as test harness.

Test harness has an advantage as they can carry out tests in two ways:

- Automation test
- Integration test

In automated test a test harness runs several cases under varying conditions and notes the output. It also monitors the behavior during several test conditions. The test harness then compares these outputs to the desired output of the test. In integration testing, there is a combination of two or more modules and tests are run to check whether the results are as expected that is, desired output or not.

Therefore in this project we expect that the test harness should have the capabilities to perform a number of desired tasks. A few of them are listed as:

- It provides logging facilities.
- Create a test harness that can run tests concurrently and also run subsequent tests.
- We can make use of communication methods between different modes of the system, to take proper inputs and supply proper test results, and smooth integration of these modules.
- It should have a queue for input data to manage high incoming load.
- Possibility of stubs and drivers should be available for testing purpose, whether individually or both together.
- The test harness should be able to process both, automated testing and continuous integration.
- It should be able to support regression testing.
- It should be able to record the result of every test and have a detailed report of it.
- We can make use of multithreading to run concurrent projects.
- It should have an App-Domain manager.

The possible users of this operational concept document (OCD) are developers, testers and quality assurance managers. Quality assurance managers as the name suggests take active role in the proper functioning of the test harness and maintain quality. Developers can use this document as a guidance to build the test harness project in the later stages. It is important for tester to be an active part so that there is no gap of communication between developers and testers as to what is being made and what is the demand of the testers.

There are some critical issues that might need to be managed during the building of the test harness who's solution will be referred to in a later section of the document, a few of these issues are:

- The testers might not be able to understand and use the test harness build by the developers.
- The testers might have not received the features that they need in the test harness.
- A few of the functionalities might not work during demonstration as expected.
- Deadlock issues in the testing cycle.
- What if there is a failure or lag of communication during integration testing of different modules?
- Maintaining a detailed report of all the test failures might be an issue, even for the programmer to understand.

2. Introduction

Software testing is a process to measure the correctness and quality of a developed software or application. To fulfill this task we make use a test harness. Test harness is an environment that runs various tests, automated and integrated, which gives us the measure of completeness of the software. To check whether there are defects or not the output of the tests are compared to the desired outputs and what changes are to be made to the software are thereafter decided.

Now one might say that to make software bug-free we can test each and every permutation and combination possible, but this increase the cost and time factors drastically. What would we do in this case is assess the possible risks in an application and test those. An experienced developer can catch such faulty sites and test it. Care must be taken that similar kind of cases are not tested repeatedly, because the same test cases will not lead to new bugs. Another important factor in the test harness is that it requires continuous reviewing and save detailed reports of test cases, especially the faulty test cases. Reviewing different type of test cases will bring out other type of bugs and hence contribute to the completeness of the system. These are the reasons why test harness is important and a brief overview of how they should work. The working of a test harness for different types of soft-wares can be subjective.

2.1. Automation Testing

Manual testing is a very primitive type of testing method and requires a skilled tester to carry out its operations, adding to that it is pretty time consuming and repetitive. Our test harness should support complete automation and integration testing. With proper communication of what are the needs of the tester and what the developer is going to build, a test harness can be created to run tests continuously without human intervention.

Automated tests are easier to manage as they create reports of the test and leave out the tedious task of manual testing. In fact, it supports offline testing too. Automation testing is the primary function of a test harness and is built for complex applications and for applications that need to run test cases concurrently and on a continuous basis. A few areas where we can implement automated tests are:

- Tests that need various data sets
- Tests that can cause human error.
- Applications that use certain functionalities that can be risky.
- Severely reduce time and cost factors.

Now, while building a test harness we need to populate the test data correctly to run the automated tests properly. One such method is when the testers have complete control of the testing environment. Here data loading (in any type of storage facility) is much quicker than any other method. A problem arises when the testers might have to manipulate the existing data structures into a more desired type of data structure.
Another method is to create test scripts. These test scripts run and help populate the storage facilities with the data required for automated testing. This method is slower but has less dependency on data structures.

2.2 Integration Testing

In a testing environment it is important that we have two cases to validate. Naturally one is where it is made sure that valid input is parsed and in the other scenario a false input is passed and an exception is raised. This method should be followed throughout as we write small or large chunks of code. The reason behind it is that the newer code is tested and integrated to the existing code. This is one of the other main functionality of the test harness, continuous integration and integration testing.

A software is made up of multiple modules that actually might have been made by separate teams. It is important that when these separate modules pass their individual automated tests and integrated, the data communications amongst these modules are perfect for proper functioning of the application. During unit testing the modules might work perfectly but different developers have different style and concept of building a piece of code, the important part is that when these modules are brought together as a single unit, they should work without nay erroneous effects.
Another important point to note is that the clients might change their requirements during the course of developing and these might not be unit tested and hence integration testing becomes even more important.

Steps to follow for continuous integration in software are:

- Maintain a single source repository
- Automate the build
- Commit should be built on an integrated machine
- Automate integration testing for testing the links between two modules.

Integrated testing provides completeness to the testing factor of an application because here it checks the flow of data between modules and not unit wise. Usually integration testing takes place logically that is, it checks the modules that are logically related and incrementally adds modules until the entire software has been tested.

2.3 Organization

If the data is correctly managed and tested in a well directed manner, the process of testing becomes easier, faster and causes less confusion in the methodology and collection of data, both input and output.

2.3.1 Organizing Test Data

Organizing test data is an important factor in test harness. There might hundreds of thousands of test inputs to be tested for a particular software. The important factor is that the test data should not be dependent on each other. Tests cannot rely on what happens prior to them or they create confusion on whether the test is faulty or the code. Each test should have it's own setup such that is should have the setup for a particular test to run on it's own. Each test should leave the system in it's initial state after completion. Hence isolation of test cases is important.

2.3.2 Integration testing organization

Testing can be done in two ways, bottom up and top down approach. In the bottom up approach all the lower level modules are tested with the higher level modules until all are tested. Similarly, in top down approach all the higher level modules are tested with the lower level modules until all modules are exhausted.

2.4 Prototype Implication

Our entire project is based on the test harness, and an important part of the code is the App-Domain. The App-Domain it isolates executed software applications from each other such that it does not affect other applications. The code prototype for this app-domain gives us a brief idea about how would we implement the parent and child app-domain for each child for each user. It would load the given assemblies from a child app-domain. It also used to display each dll file separately in the app-domain. The other one is the file manager which is important cause we need to load each of the user files correctly for the process to start using Directory.getFiles(parameters).

2.5 Architectural Ideas for Test Harness

The idea of a test harness is to implement App-Domain where we can run multiple assemblies to run within a single process but it also prevents assemblies from accessing each other memories. We can make use of a single App-Domain for a dynamically loaded DLL from a folder, this will not allow different tests to interfere with each other. For this we will be coding in C# using visual studio 2015 and .Net framework.
We have three block architecture for a test harness which consists of the input layer, the domain layer and last is the test logic layer.
The diagram of the architectural layers of the test harness are shown on the next page.

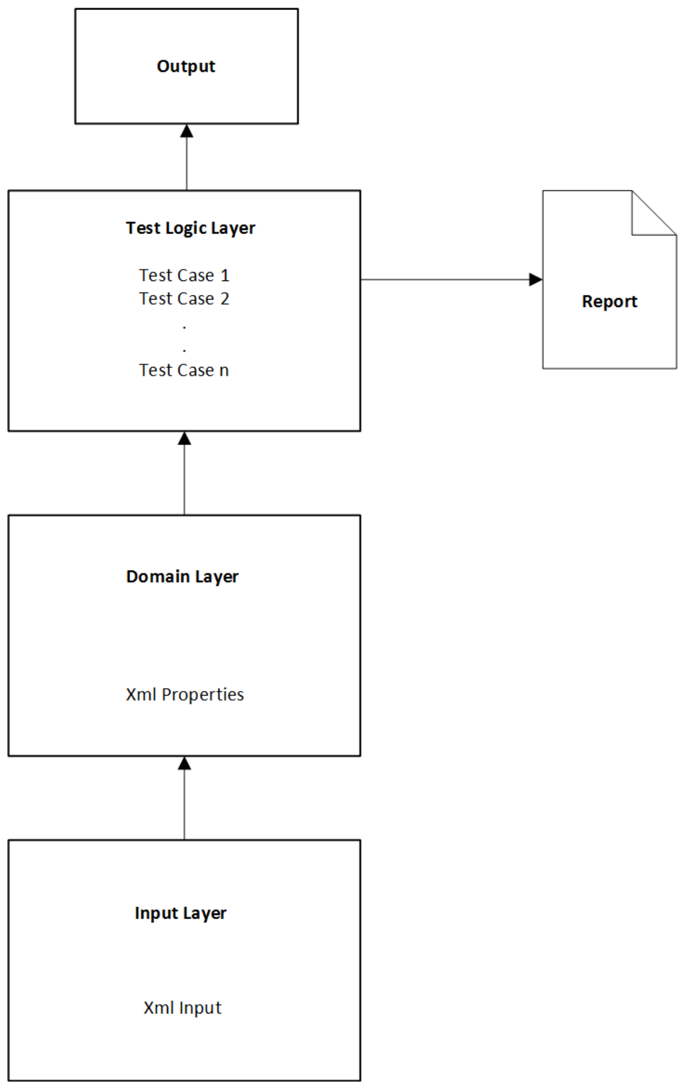


Figure 1: Architecture layers of a test harness

The three layers are explained as:

- **Input layer:** These are basically the test cases that need to be examined. As seen in a few examples by our instructor, Dr. Jim Fawcett, These cases are like

- **Domain Layer:** This layer has the task of getting the Xml properties from the Xml input from the user. Rather than X-path or Xml it represents the test cases in domain language.

- **Test Logic Layer:** These consist of all the test cases and expresses the test logic with the help of the domain layer below and provides the test results.

- **Report Generation:** One important job that the test harness performs is generate reports of the performed tests, whether successful or unsuccessful.

3. Users

The key users of this test harness can be software architects, teaching assistants, testers, developers, test managers, managers and quality assurance managers. Most of these users are a part of the entire life cycle of this test harness. The way in which each user uses the system is different and described in detail below:

3.1 Software architect

As the name suggests the software architects manages the systems architecture and it's attributes. The ultimate aim of the architect is to make sure that the built system accomplishes the goals it has been built to achieve. Namely a few other functions of the architects are :

- Define the document of the system.
- How to make advancements in the system for betterment.
- Make sure the integration process takes place in overly fashion and proper functioning of the system.
- Make sure the functionalities are properly understood by the developers and testers.

3.2 Quality Assurance Manager

Quality assurance managers basically are interested in the smooth functioning of the system and assure quality of service to all it's clients. They manage the following activities:

- They review the project requirements and quality criteria.
- They design test scripts and mange it.
- They manage and control the automation of tests in the test harness.
- Maintain that the functionalities required by the customer can be performed by the system.

3.3 Teachers Assistants

Teacher's assistant works closely with the developer to check whether the system works in an expected fashion without any issue and also make sure the code is clean and simple.

3.4 Developers

The developers of the test harness have to work closely with the testers. This is important because there shouldn't be a wide gap in what the developer has produced and the actual needs of the tester. They define the requirements of the system and start the coding process. They are also responsible to create test cases that are used to test the modules they have created.

3.5 Test Manger

Software test manger is the team leader of the software testing team and responsible for all the testing teams functioning.

- They delegate work to the testing team.
- They are responsible for communication with clients when needed.
- Responsible for the election of the right tools.
- Responsible for the integration of tests.

3.6 Testers

Testers should have the ability to understand usability issues and generate test cases. They should also have knowledge of the execution methods. A tester should be in constant communication with the development team to ensure the right system is built. A few functions are:

- Carry out testing according to the procedure.
- The most essential job is to create reports of the tests that are carried out.
- Responsible for designing theist scenarios and also knowing what to test, as testing all permutations and combinations is an impossible task.

3.7 Manager

Manger is the main coordinator and work delegator in this system. A few roles of a project manager are listed below:

- Define activities for testers, developers and other project groups like maintenance teams etc.
- To make sure the required resources are available to all the teams.
- They make the overall report of the system.
- Make sure that all the processes are carried out in strict order and in line with the guidelines of the project.

3.8 Uses

Test harnesses have multiple uses in a testing environment and along with the automated testing environment it's uses increases because of the time efficiency. The main uses of a test harness are listed below:

- Increased effectiveness : Since we are using an automated and continuous integrated environment the cost and time factors both decrease exponentially as compared to manual testing.

- It supports regression testing. Hence it also makes sure that the integration of software is done correctly or not, and what changes to make before further testing.

- A detailed report of the test results are made, especially for the failed tests. Using these reports a track of the test cases and test cases outputs are kept, keeping the system organized and it's smooth functioning.

- Since we start testing at an early phase, the system tends to be more reliable. Also, these systems run repeatedly and without human intention.

- It supports logging. Logging is perhaps a very strong tool in the developers hands. It gives data about what the code is doing an handles complex situations well and quick debugging.

3.9 Impact On Design

Ideally, testing should be designed such that it takes minimum input from the user, and displays the results of the test effectively in a pretty concise manner. It should basically be a well crafted code with simplicity. Precaution must be taken that we do not stuff up everything into one package. Continuous integration support is one of the main features of a test harness can have and all the packages should be cohesive with proper naming conventions should be followed. Having all this in mind we need to create proper test packages that can demonstrate the functions of the test harness. Here we also use loading testing, in this where the performance of the system is tested in both normal conditions and peak or high load situations.

4. Application Activity

This shows all the major task of the system that are broken down into activities. It is represented in the form of an activity diagram as shown below.

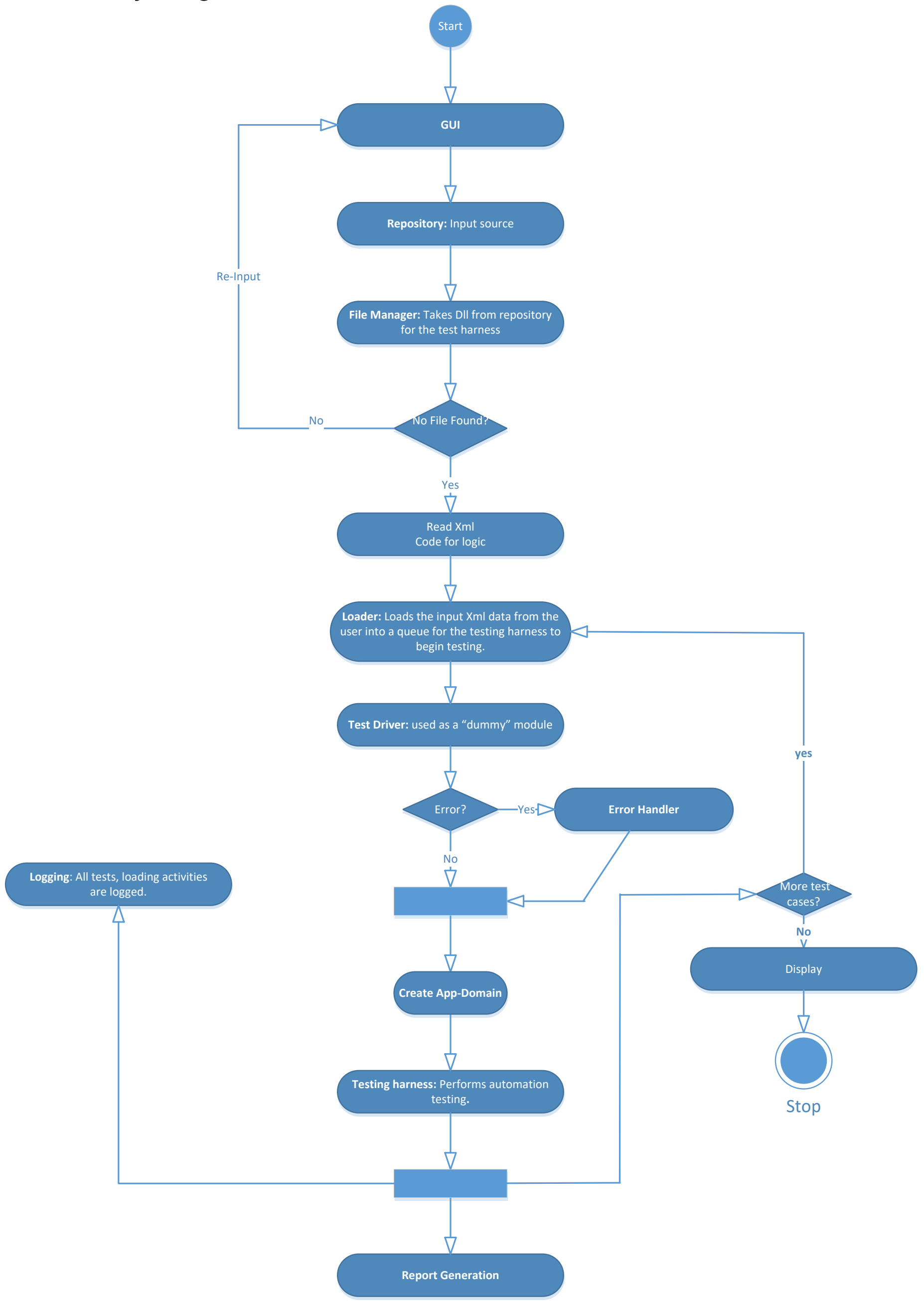

Start
GUI
Repository: Input source
File Manager: Takes Dll from repository for the test harness
No File Found?
No
Re-Input
Yes
Read Xml Code for logic
Loader: Loads the input Xml data from the user into a queue for the testing harness to begin testing.
Test Driver: used as a "dummy" module
Error?
Yes
Error Handler
No
Logging: All tests, loading activities are logged.
Create App-Domain
Testing harness: Performs automation testing.
Report Generation
More test cases?
yes
No
Display
Stop

Figure 2: Activity diagram for the test harness system

The flow of the activities in the system is very important, such that all the process should run in order for proper functioning and not even the smallest activity is left out. The activities that take place in this system are as explained:

- **4.1) GUI**: We start with the Graphical user interface or GUI, which is the interface of the system for the user and takes input from the user that marks the beginning of this test harness system. In our project we focus less on the GUI aspect and more on the coding and documentation aspect of the application.

- **4.2) Repository**: Repository is what contains all the data link library (dll) and logs, it needs to be connected to the loader or test harness. It basically contains the testing code from the user input that needs to be tested.

- **4.3) File Manager:** The file manager itself does not contain any data, it helps in connecting the repository to the Test harness functionality. There is a decision box there that checks if the file is actually present in the file manager that has to be tested, if yes it goes further to the next activity. If not then the process is sent back to the GUI phase to make sure the file is present for testing.

- **4.4) Read Xml code:** Here the logic for the Xml code is retrieved that tells us what testing operation we need to do for a particular input.

- **4.5) Loader:** As the name suggests the loader loads the file to the test harness environment for the testing activity to start. There is one more step here after the loader is the test driver phase is used for integration testing. Loading of file details are also logged.

- **4.6) Test Driver:** Here we make use of a “dummy” module to complete integration testing process. It is used when the calling function is needed for execution by the lower module function to complete the testing. It is more complicated than stubs.

- **4.7) Error Handler:** If there is an error during integration or any exception error like data structure error is empty or queue is empty we have to cats these errors and make them test ready. We can also use try-catch block for exceptions.

- **4.8) Queues**: The input from the user is put into a queue for the test harness to pop and run tests on. This avoids overloading of input data and unnecessary complexity in the system. Basically the user queues the Xml input data and the test harness de-queues the data for testing.

- **4.9) App-Domain:** This is an important part of the code here we use App-Domain such that if one data is running a test then it’s presence or result should not affect any other test that is or will be conducted. One single user has one App-Domain for it’s test cases. So each user has a different App-Domain for it’s test cases.

- **4.10) Test Harness:** Here is where the crux of our code lies and also the central idea of our project. Test harness can compute different types of automaton testing be it regression, unit

Testing or integration testing. It computes test cases concurrently and gives test results for each case whether positive or negative.

- **4.11) Report Generation:** This is for the generation of a statement about a test that has been conducted. It is a detailed summary about the test cases from the input to the output. Irrespective of the test result it reports it and maintains the data of that particular test case.

- **4.12) Logger:** The way to a proper coding application is that a developer must log very possible activity. If any error or exception occurs the first thing a developer does is checks the log files. Hence we log everything form the test output to the loading process.

- **4.13) More test cases?:** This is a decision box that needs to run so that the test harness ca make sure that all the test cases in the queue have been completed that were requested by the user. If not it display's the result to the user. If there are test cases left in the queue then it goes back to the loader to mark the start of the testing of the next case as shown in the activity diagram above.

- **4.14) Display:** This simply display's the result to the end user notifying it about the success or un-success of the test.

The stop symbol suggests the end of the testing process.

4.2 Activity Diagram for Testing Process only

The diagram below shows the main process of the testing process in the application.

- Testing is the core area of our project where we process test cases to be tested in an automated way.

- Once the initial phases of the repository and file manager are understood as explained in the earlier activity diagram the next step towards testing is when the App-Domain is created.

- The child App-Domain has two roles:

1) Isolate different test cases from separate users so that the test running does not affect another running test of a second user by the intervening of unhanded exceptions.

2) It enables the test harness to unload libraries, which means one has to unload the entire library and not possible to unload only one. So, if we do not have any child App-Domain then the main App-Doman has to be unloaded which stops the working of the entire test harness. Hence it is clear that we need to unload libraries on a regular basis to avoid the memory from getting over used.

- Once all the inputs from the queue are popped and executed the test harness moves onto the test execution of anther user inputs after logging the results of the process.

The diagram of the above explained part is shown below:

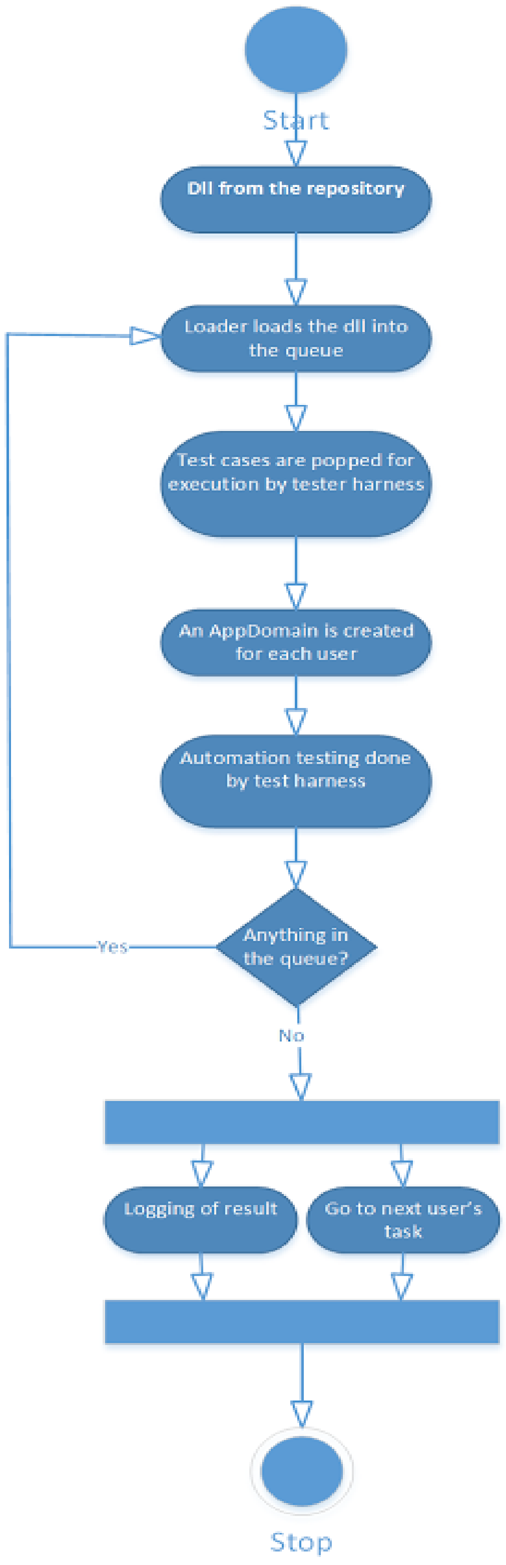

Start
Dll from the repository
Loader loads the dll into the queue
Test cases are popped for execution by tester harness
An AppDomain is created for each user
Automation testing done by test harness
Anything in the queue?
Yes
No
Logging of result
Go to next user's task
Stop

Figure 3: Activity diagram of the testing process

4.3 Flow of data:

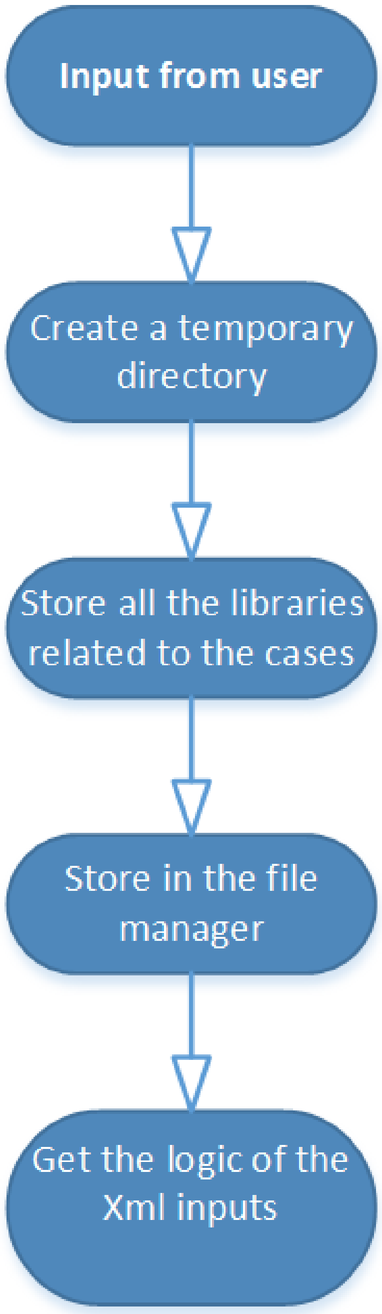

Input from user
Create a temporary directory
Store all the libraries related to the cases
Store in the file manager
Get the logic of the Xml inputs

Figure 4: Flow and reading of data before reaching the testing area

Knowing the flow of data is an important factor in developing an application. It has to be known precisely so that correct data can be monitored correctly and exceptions can be handled in an easier fashion. In fact that is why we even log the entries of when the data is loaded or queued or tested. The diagram above shows the flow of data before it reaches the testing area. It also shows what information is extracted from the data that is, the Xml input.

4.4 Queuing and testing of data

The diagram below shows how:

- The input data from the user is stored in queue after the initial processes on the data are completed.
- The data is stored in the queue for the test harness to test and display the result.
- The test harness de-queues the data one at a time and runs the testing processes on them. It continues this process till the queue is completely empty.
- After this it displays the result.
- An important point to note is that all the inputs for a single user have one App-Domain, and separate users have a different App-Domain so that test results are not affected.

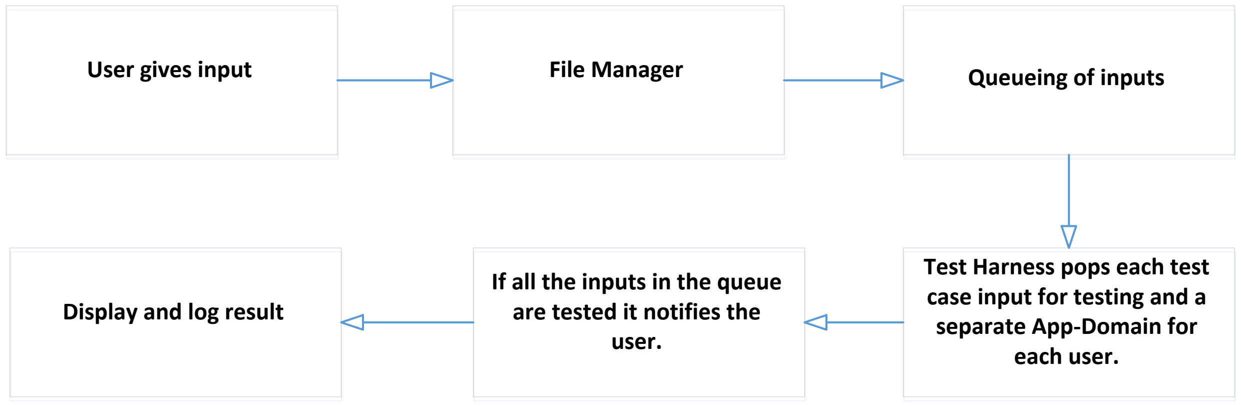

User gives input
File Manager
Queueing of inputs
Test Harness pops each test case input for testing and a separate App-Domain for each user.
If all the inputs in the queue are tested it notifies the user.
Display and log result

Figure 6: Queuing Process

5. Partitions

In this each task becomes a package candidate. In this we decide the name of each package and explain the function of each package and also the communication between them.
This figure below shows us the package diagram of the test harness system.

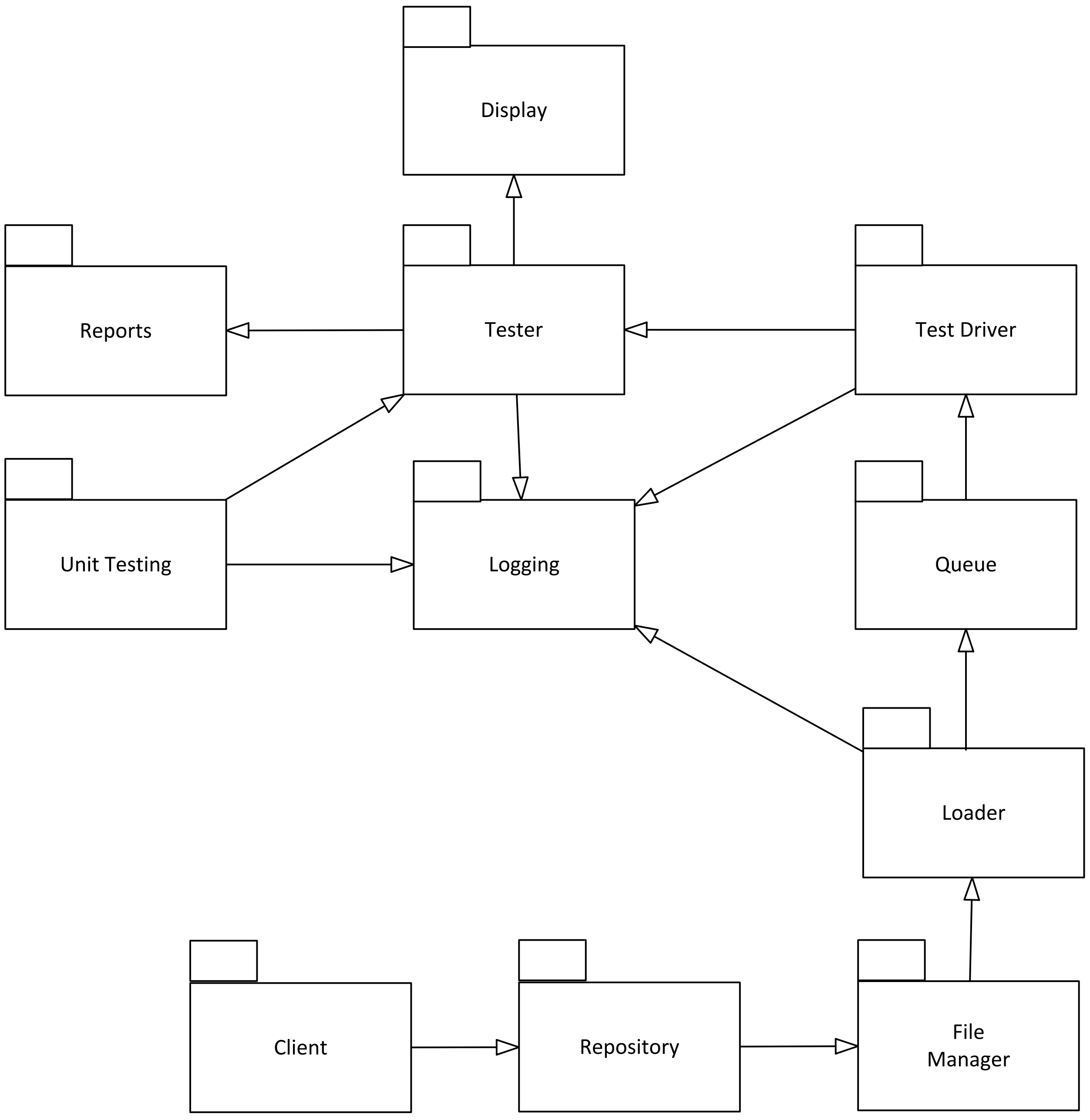


Figure 6: Package diagram for the test harness application

The working of these packages is as explained below:

- **5.1) Repository**: In a single line it is a central location in which data is stored and managed in an organized way. In software testing it is a baseline of the test plans of the system and various types of testing. The repository is accessible to users and also a place where data and documents are found and used further in the system. Here for our purpose the repository consists of the Xml code that needs to be tested.

- **5.2) File Manager:** This is a connection that connects the repository to the loader. It itself does not consist of any data but from the repository.

- **5.3) Test Driver:** This an important part of the test harness system and make as the "dummy" function for a calling module. They are used during bottom up integration testing in order to act as temporary module for the higher modules that have not been tested. They give the same output as the actual module. They are also be used when the software needs to interact with an external system. Hence they are the building up process for an automation testing environment.

- **5.4) Loader:** A loader is a component that locates a certain document or data in a storage area and loads it to a main stage area of the computer and gives the control power to this particular data. Here in our system the loader fetches the particular input Xml files and gives it to the test driver.

- **5.5) Tester:** This is the main component of our project, the tester. Here is where the input code from the user is tested for different tests and a result is received. These are automated tests and run multiple test cases concurrently. Each test is conducted in such a way that it's result does not affect the result or working of the tester for an another test. There is a single App-Domain for a single Xml request, which is generated by the test harness.

- **5.6) Reports:** Every test that has been conducted needs to have a test report whether the test was a pass or fail. Reports are a formal way of representing test results. They give us an opportunity to estimate test results before hand. They show the operation condition of the test conducted and help us to compare it with test objectives and desired results.

- **5.7) Logging:** Logging is keeping track of all the test results, whether the loader loaded the data correctly or information about any testing that was carried out. Basically logging increases efficiency of a system and code. It can communicate what the system is doing with the developer and help decrease complexity of the system. That is why as seen in the diagram logging has been used at every step.

- **5.8) Unit Testing:** Unit testing although preferred to be done via automation it can be done manually too. Unit testing is basically the testing of all the small modules of the system to ensure their proper working before we move onto the her testing phases of the system and input data. These results are logged too.

6. Critical Issues

These are issues that a test harness might face as discussed in the earlier section. Here we will give a detailed ideas and solution to each issue. The solutions to such issues are discussed in detail below:

6.1 Easy To Use

This is the most important critical issue in a test harness or rather any software project.

Solution: It is important that the testers, developers, managers and clients work and discuss the project details and work out the necessary requirements of the project. If this doesn't happen the software created by the developers wouldn't be of any use to the clients and the testers might not understand how to use it or haven't been provided with the functionalities they were looking for. This document can be very handy for all the users of the project. A strict agenda and protocol should be followed towards building the project and testing methodologies, in this way the project is clear and it becomes easier to use.

6.2 Performance

The system should be able to handle high load as well, when the flow of input is higher and the load increases it might lower the speed of the test harness.

Solution: For this way we can make use of a data structure,queue. This can hold the input in case of higher load and not affect the performance of the system.
Also, we can make use of messaging between modules, request and acknowledgment based for usage of resources and not affect unwanted situations.

6.3 Deadlock Situations

Since we make use of multithreading during programming on a test harness and although they provide us wit better performance the create a new issue, deadlock.

Solution: In this issue we can make use of thread pool. These thread pools reduce the number of application thread and provide management of worker threads. They also help in decreasing the complexity of the program.

6.4 Managing Regression Testing

When the project keeps in expanding, sometimes the the regression testing might go out of hand or become uncontrolled.

Solution: To check this uncontrolled problem of regression testing we can:

- Identify and label the cases to be tested.
- Manage the new modification of the regression tests to avoid confusion.
- Maintain a test log of the regression test cases result.

6.5 Overloading

As there can be endless testing options, there might be an increased load on the resources and hence the system. There can be high number of cases and this can affect the performance of the system.

Solution: There can be a check on which module is using the sources and for how long. Based on the urgency, requirement and priority a module can gain access of the resource required. The lock system can be used.

6.6 When to stop testing?

The possibility of performing tests are never ending as the permutation and combination of tests cases can be endless. This makes testing an infinite process. We need to have a check on the system to know when to stop the testing process.

Solution: Truly, there are three ways by which the testing process can be affected:

- Time factor
- Cost factor
- quality factor

Now depending in the budget and time factor testing need to be done smartly, leaving out the unnecessary cases that do not need to be tested. Manual testing can check the easy bugs that do not require automation testing. For the quality factor we can stop when the quality meets the requirements or further testing will yield no better results than that are already present.

6.7 Service Level Agreement

We need to set a bar on the time frame or availability for a particular test case to increase the efficiency.

Solution: For this we use Service Level Agreement or SLA is basically marked as a measurable requirement check. It can label test case as ‘pass’ or ‘fail’. Depending on the response time the tester can monitor and check for inefficiencies in the test cases.

6.8 Catching Exceptions

The app domain will crash if there is an exception at runtime.

Solution: .NET provides to handle an exception at runtime and also the developer must try to run exceptions by using try try-catch block.

6.9 Security

This system is being build for a class project and does not require any sort of security like authorization or authentication.

7. Conclusion

Hence in conclusion we have created a concept document that can be used by different users as mentioned in the earlier section of the OCD. It gives an idea of the working of the test harness because of the separate diagrams that have been drawn. In the next section there are a few prototype codes that explain the working of a few main areas of the project like loading using App-Domain and getting the list of all Dll files from the file manager in a specified directory. The document will be of great help to build the test harness in project #2 using C# and .Net framework.
(P.T.O)

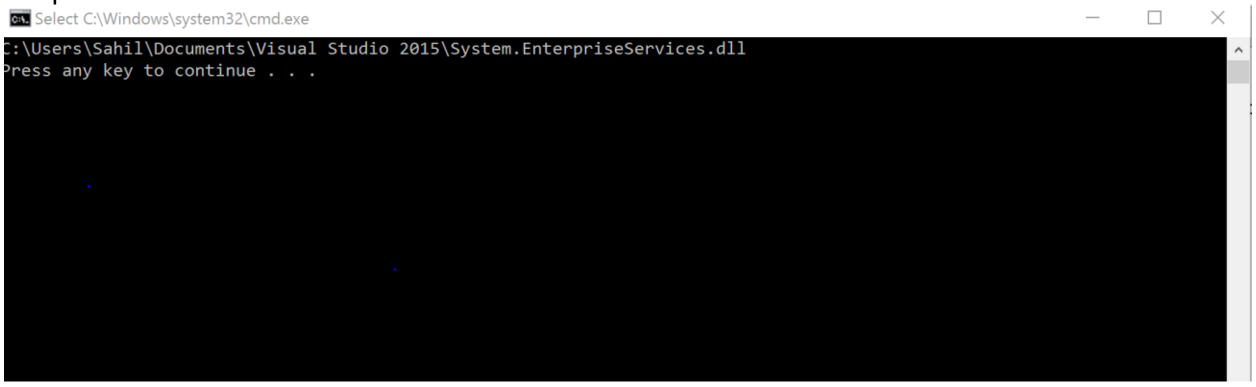
8. Prototype

Here are a few code prototypes to first get .dll files from a directory and second to load an App-Domain, execute it and display the result.

8.1 File Manager package that searches a directory tree, rooted at a specified path, and displays the names of all DLL files encountered:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace getFiles
{
    class getFiles
    {
        public static void Main()
        {

                // Get all the files ending with .dll in a specified directory
                //Using Directory.GetFiles(path, file ending in, search all directories)
                string[] filepath = Directory.GetFiles(@"C:\Users\Sahil\Documents\Visual
Studio 2015", "*.dll", SearchOption.AllDirectories);

                foreach (string file in filepath)
                {
                   Console.WriteLine(file);
                }

        }
    }
}
```

Output:



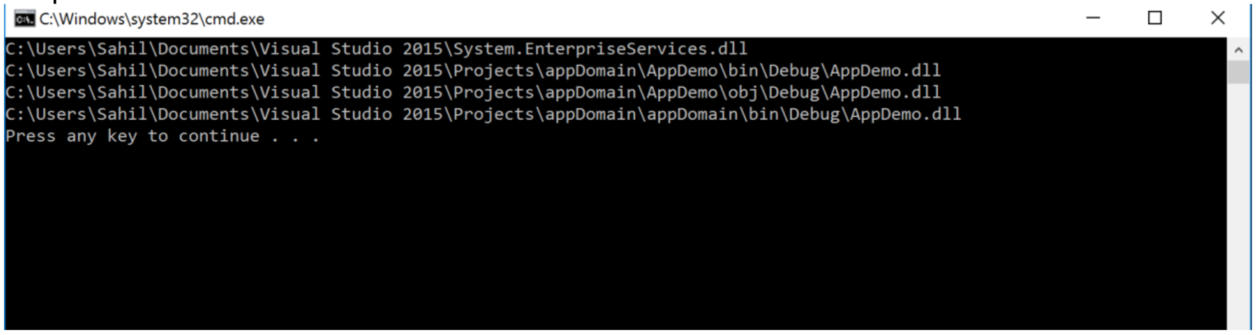
```
C:\Users\Sahil\Documents\Visual Studio 2015\System.EnterpriseServices.dll
Press any key to continue . . .
```

*We can use try-catch block, but it is not necessary for this code prototype.

8.2 Child App-Domain demo that loads a simulated test library, executes it, and displays the results.

```
//With the help of Dr. Jim Fawcett's code from his website.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Security.Policy;
using System.Text;
using System.Runtime.Remoting;
using System.Threading.Tasks;

namespace appDomain
{
    class Program
    {

            static void Main(string[] args)
            {
            //creeating child App-domain
            AppDomainSetup dinfo = new AppDomainSetup();
            dinfo.ApplicationBase = "file:///" + System.Environment.CurrentDirectory;
            Evidence evi = AppDomain.CurrentDomain.Evidence;
            //App Domain is created with this command with name Test
            AppDomain Test = AppDomain.CreateDomain("ToTestAppDomain",evi,dinfo);
            Test.Load("AppDemo");

            //give the path of the other .exe file to see if the app domain can run it
            //    Test.ExecuteAssembly(@"C:\Users\Sahil\Documents\Visual Studio 
2015\Projects\csdemo\csdemo\bin\Debug\csdemo");
            ObjectHandle handle = Test.CreateInstance("AppDemo", 
"ConsoleApplication4.Program");
            object obj = handle.Unwrap();
            ConsoleApplication4.ITinterface console = 
(ConsoleApplication4.ITinterface)obj;
            console.app();

            }
        }
    }
```

Output:


C:\Windows\system32\cmd.exe

```
C:\Users\Sahil\Documents\Visual Studio 2015\System.EnterpriseServices.dll
C:\Users\Sahil\Documents\Visual Studio 2015\Projects\appDomain\AppDemo\bin\Debug\AppDemo.dll
C:\Users\Sahil\Documents\Visual Studio 2015\Projects\appDomain\AppDemo\obj\Debug\AppDemo.dll
C:\Users\Sahil\Documents\Visual Studio 2015\Projects\appDomain\appDomain\bin\Debug\AppDemo.dll
Press any key to continue . . .
```

Another application in the project folder is called and executed by the current App-Domain

9. References

1) http://ecs.syr.edu/faculty/fawcett/handouts/csE681/Lectures/Project1-F2016.htm
   The entire crux of the project and OCD given by Dr. Jim Fawcett.

2) http://www.guru99.com/software-testing.html
   Basic information about different types of testing and users of software testing.

3) http://stackoverflow.com/
   Helped resolving small conceptual errors

4) https://msdn.microsoft.com/en-us/library/system.appdomain.aspx
   Gave definition and explained syntax for the prototype.

5) http://istqbexamcertification.com/what-is-test-harness-unit-test-framework-tools-in-software-testing/
   Entire basics of the software testing concept.

6) https://www.wikipedia.org/
   Small definitions about different components of the application.