

PROJECT #1 - TEST HARNESS: OPERATIONAL CONCEPT DOCUMENT

CSE-681 SOFTWARE MODELLING AND ANALYSIS

Instructor- Dr. Jim Fawcett

Sneha Patil

Spatil01@syr.edu

10th Sep 2016

TABLE OF CONTENTS

EXECUTIVE SUMMARY	4
INTRODUCTION	5
CONCEPTS AND KEY IDEAS.....	5
REQUIRED FUNCTIONALITITES	6
ORGANISING PRINCIPLES	6
USES	7
USERS AND IMPACT ON DESIGN.....	7
COURSE INSTRUCTOR AND TEACHING ASSISTANTS	7
DEVELOPERS	7
QA's.....	8
Manager.....	8
POSSIBLE EXTENSIONS	8
STRUCTURE AND PARTITIONING	10
DEMO CLIENT.....	11
CS BLOCKINGQUEUE.....	11
TEST EXECUTIVE.....	11.
APPDOMAINMANAGER.....	12
LOGRETRIEVER.....	12
LOADER.....	12
ITESTINTERFACE.....	12
LOGGER.....	12
MESSAGEPARSER.....	13
PARTITION.....	13
REPOSITORY.....	13
KEY APPLICATION ACTIVITIES	14
CRITICAL ISSUES	17
EASE OF USE.....	17
PERFORMANCE.....	17
EXCEPTION ON RUNNING TEST DRIVER.....	17

Project #1 Test Harness

SECURITY IN REMOTELY ACCESSED DATABASE USED FOR STORING TEST LOGS.....	17
DEMONSTRATION.....	18
INCONSISTENCY IN INPUT REUEST MESSAGES.....	18
REFERENCES.....	18
CONCLUSION.....	19
PROTOTYPING.....	19

LIST OF FIGURES

Figure 1: Package Diagram	10
Figure 2: Request Processing Activity Diagram	15
Figure3: Activity Diagram for the APP Domain Manager	16

1. EXECUTIVE SUMMARY

In earlier days of software development, different teams would work on different modules and once they are done the whole code would be integrated at once. This process was highly time consuming, and if the modules broke each other's code, it was detected only after integration. Continuous integration which works on the principle of "always keep it working" helps in earlier detection of module compatibility issues.

The aim of this project is to create an automated test tool i.e. a test harness that runs the specified tests on multiple packages. Each test execution would run the test driver on the specified test code from the repository. The results of the test would be logged for the purpose of analysis. The test requests are sent as messages which are processed by the child appdomain created by the test harness to isolate individual test requests.

The main users of the test harness we develop in project 2 are the graders, instructor and me. Thus the systems would be designed in such a way that the evaluation is easy and all the requirements are well demonstrated. The extended full-fledged test harness with remote access and multithreading facilities can be used by developers, Quality assurance team, and Manager.

There are certain significant issues that need to be addressed while developing a Test Harness. Some of these are:

- The test harness would be one of the busiest servers in the project. One of the most critical issues would be its performance. The time required to complete the tests would increase as number of test cases increase, especially the QA test which generally run all the test drivers to provide the proof of concept would need a huge amount of time. This can be partially managed by multitasking and sharing of resources.
- The test harness should be easy to use and thus providing a simple and easily understandable interface is crucial.
- Demonstrating to the graders that all the requirements of the project 2 are met without them having to manually enter data.

2. INTRODUCTION

CONCEPT AND KEY ARCHITECTURAL IDEAS

For building large systems successfully, we need to divide the code implementation into smaller modules. Before integrating these modules into the baseline code we need to thoroughly test it against the base line to see that the new code does not break anything. We thus might have run and re –run several tests on the whole baseline or the subset of it. Test harness automates the testing process. Manual testing is very expensive, as the code could have complex dependencies between the solution side and the application side. For Quality testing thousands of test cases need to be run. Test harness by automating the testing process saves time and increase efficiency. The Test harness supports continuous integration amongst multiple developers to build a baseline. All the code the module depends on can be tested against the baseline and we could be sure that our module does not break any code as soon as possible.

The client sends test request message which contains the details about the test drivers and the test code. The test harness runs the specified tests on multiple packages and logs the pass fail status and also the execution details.

The test request message sent by a client can be in the form of an xml file. It specifies the test developer’s identity and the test drivers and test code to be loaded. The test drivers and test code is in the form of dynamic link libraries.

The test harness runs each test request in an isolated container provided by the child app-domain. Every process by default has only one application domain, but this can be extended to create multiple app-domains. The key idea is to keep the test isolated, since an exception in one of test could bring the whole test harness down. Therefore after receiving a test request message from the client, the test harness creates a child appdomain, for this client which parses the message to decide the test drivers and test code required and loads it through the injected loader.

We can demonstrate we are meeting all the requirements to the Graders and TA's by making the logger write to the console as well. The logger would thus write its current status to the log file and the console. Giving insight into project execution.

REQUIRED FUNCTIONALITIES

Following functionalities of the test harness will be implemented in project 2:

- It will accept the test request message from a client in the form of xml file
- It will enqueue the message and execute it in the dequeued order
- Xml file would specify the developer identity, required test drivers, and test code
- Test harness will create a child appdomain and inject the loader which would load the test driver dynamic link libraries and test code dynamic link libraries. It will then run the respective drivers on the test code.
- The log of the pass fail status would be maintained.

Following are the extension of the functionalities that can be added to a test harness in real life scenario:

- It should accept the request from multiple clients
- It should simultaneously process requests from multiple clients
- It should provide remote access to the clients
- It should have an easy to use graphical user interface
- It should automate the test process.
- It should be able to run thousands of test within reasonable amount of time

ORGANISING PRINCIPLES

Most of the functionalities mentioned above will be achieved by splitting them into smaller tasks. Each task, or a group of related tasks, will be put into separate packages that make it easier to understand the operation the overall application. The various packages that make up the application and will include packages that control the program flow, test the various functionalities, process test requests, perform loading of dll's and log the results and execution.

3. USES

The current project focuses on designing and implementing a test harness, and as such, its primary uses will be restricted to running test drivers and logging the test results, and execution summary.

USERS AND IMPACT ON DESIGN

1. COURSE INSTRUCTOR AND TEACHING ASSISTANTS

These users will focus on testing the implemented test harness. This includes checking the various required functionalities and evaluating the overall performance of the application.

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 lucid, concise manner.

Keeping this in mind, the program should include a Test package that will be responsible for demonstrating each of the requirements by running step-by-step through a series of tests. The results should be displayed concisely to facilitate easy comprehension.

2. DEVELOPERS

Developers are the primary users of test harness. Integration tests are a fundamental part of continuous software integration and are used by each developer. They need to thoroughly test their modules against the entire baseline or some part of it, before integrating it into the software baseline. They may have to run several tests to check that the changes made by them do not break any part of the original code.

IMPACT ON DESIGN:

Ease of use is the major design impact for the test harness because if the UI is bad the developer may want to avoid using the test harness and it would remain as an unused resource. We therefore must provide a nice GUI but this would not be implemented in project 2. The client interacts with the test harness through the console in our implementation of project 2. Project 4 implements the GUI and also the remote access facilities which would be very useful to the developer.

3. QA'S

The QA's need to run thousands of tests to demonstrate that the project meets the requirements. These are automated tests that are run by the test harness. The QA's may need to run tests on certain part of baseline code or maybe the entire baseline. In a huge code with complex dependencies the test harness is a must for the successful working of the project. The QA's may start may put the entire baseline to test in the evening and then check the logs after coming to office in the morning. The automated process thus saves a lot of time.

IMPACT ON DESIGN:

Performance is the major concern for the QA's, since they run huge number of tests. The test harness should therefore be fast enough. Apart from that it should be easy to pick up working test set.

4. MANAGERS

Managers can look at the test logs before the test delivery dates to see if the project is in good shape. They can also view the test activity data to see if the deadline would be met.

IMPACT ON DESIGN:

Managers need the test activity data. Thus the test harness should provide really good logging facilities with proper time and date stamp. It should also provide facilities to retrieve graphs showing the amount of load on the test harness. These logs should be named, identify the test developer, the code tested including version, and should be time date stamped for each test execution.

POSSIBLE EXTENSIONS

1. EXTENDING TO PROJECT 4

In project 4 we will extend the system to handle multiple clients. It would also support remote accessing form different clients. One or more client(s) will concurrently supply the Test Harness

Project #1 Test Harness

with Text Requests. One or more client(s) will concurrently extract Test Results and logs by enqueueing requests and waiting for Test Harness replies.

A full-fledged test harness can be built from the prototypes designed in project 2 and 4 if we add GUI as well. It would work with the repository to provide a professional testing environment to facilitate integration testing, regression testing.

2. REAL-TIME ENVIRONMENT

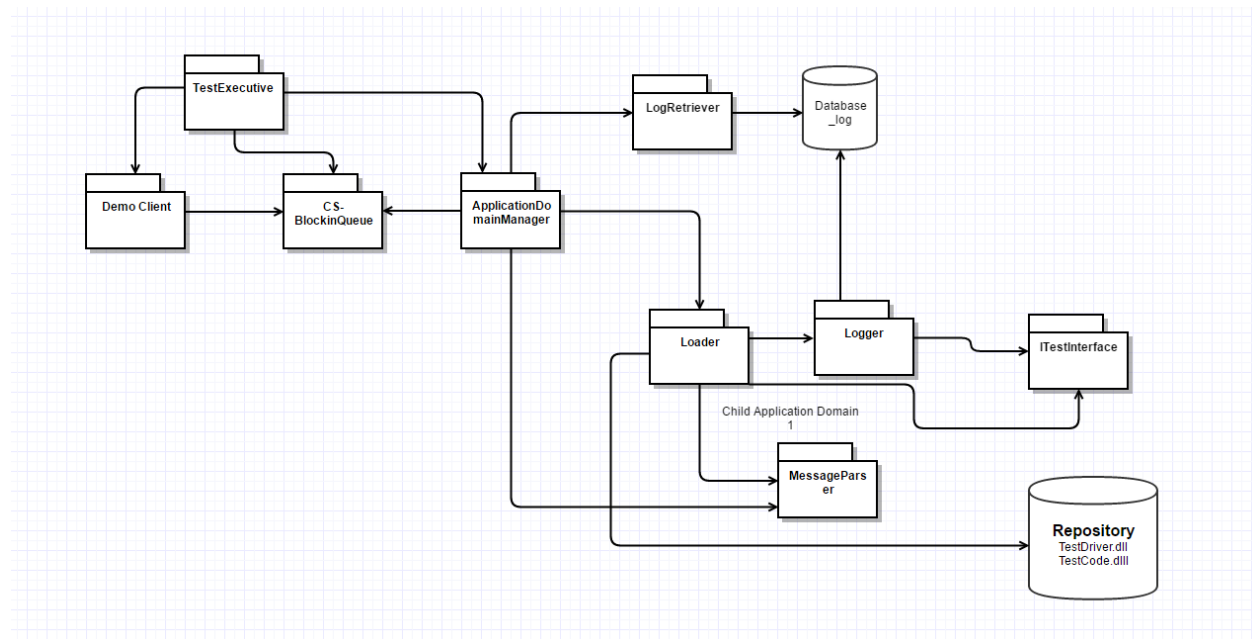
The test logs can be made to provide information of CPU usage, time taken by the events etc. Thus the test harness can be extended for use in real time systems.

4. STRUCTURE AND PARTITIONING

Most modern software applications are so large, that it is impractical to develop the program as one single source file. Instead, the application is developed as a collection of packages. The high level functionalities of the application are broken down into smaller tasks that must be carried out, in order to implement those functionalities. Logically similar sets of tasks are grouped together, into what we call packages. The various packages interact with each other, to provide the fully functional software. The chief advantages of such an approach are easier development, testing and debugging.

The Package structure of the Test Harness application is shown in Figure 1. The Package Diagram is followed by a brief description of each individual package, and its interactions with other packages.

Figure1 Package Diagram



DemoClient

Project #1 Test Harness

The client loads the queue with test requests then calls the test harness to begin executing. The DemoClient in our application sends a test requests, in the form of an XML file that specifies the test developer's identity and the names of a set of one or more test drivers with the code to be tested; it then enqueues the test request messages into the blocking queue. The .xml file path is passed to the application as a command line argument.

Following is the sample message enqueued by the DemoClient.

```
<?xml version="1.0" encoding="utf-8" ?>
<testRequest>
  <author>Sneha Patil</author>
  <test name ="First Test">
    <testDriver>td1.dll</testDriver>
    <library>tc1.dll</library>
    <library>tc2.dll</library>
  </test>
  <test name ="Second Test">
    <testDriver>td2.dll</testDriver>
    <library>tc2.dll</library>
  </test>
</testRequest>
```

CS BlockingQueue

The blocking queue package is provided by the instructor. It is implemented using the monitor and a lock. If the test harness tries to dequeue a message from an empty BlockingQueue it will block till a message is enqueued, thus providing an efficient way of waiting. Actually our application in project 2 can do with a normal queue since we are not implementing multi-threading. But as a preparation for project 4 we use blocking queue.

Test Executive

This is the package that controls the program flow, and contains the main method. The series of tests that must be performed to demonstrate the test harness functionalities will be run by this package. It also acts as an entry point into the program. The TestExec package will be responsible for creating the ApplicationDomainManager object, DemoClient Object and the blocking Queue which acts as the communication channel between client and AppDomainManager. We could think of it as a constructor for the test harness. It would kick off all the other activities.

ApplicationDomainManager

The application manager from the parses the header of the message from client. If the client request for log retrieval then it passes the message to LoRetriever package. If the testing facilities of the test harness are requested then the ApplicationDomainManager creates the child application domains for every test request dequeued from the blocking queue. The benefit of having separate application domains for each test is that it isolates them from each other. Apart from that if any of the test throws an exception, it would not bring the test harness down if it is running in the isolated appdomain. The ApplicationDmainManager injects the loader into the app domains. Creating separate AppDomains is also necessary for unloading of dynamic link libraries (DLL's). The whole AppDomain is unmounted for unloading DLL's. If separate AppDomains were not present then we would have had to dismount the entire test harness.

LogRetriever

This package parses the message to learn which test log it needs to retrieve. It then queries the database to get the respective logs. It displays the logs to the console.

Loader

Loader is responsible for loading the test files from Repository. (The DLLs). The client specifies the test driver dynamic link libraries and the test code dynamic link libraries, it needs, in the test request message he sends to the test harness. The test request message is nothing but the .xml file. The loader loads the required dynamic link libraries of the test drivers and test code. The loader also calls the test() function from the ITestInterface. After the testing is complete the logger is called by the loader.

ITestInterface

The test drivers are derived from the ItestInterface. It declares methods such as test() which does not take any arguments but returns status such as pass or fail example a Boolean true or false value. It also has a getLog() function which returns the string representation of the log.

Logger

The logger package logs the pass fail status of the test case. It can store the logs document created in a database like mongoDB. The logger makes note of the tester, the test driver executed, and the version of test code, the pass/fail status and the time and date stamp.

MessageParser

The test request message has a header which tells the port number, the type of message whether its retrieval request or test request and the body which is nothing but .xml file. The MessageParser package is used by AppDomainManager to know the type of message and it is used by the loader to get the list of test Driver and Test code DLLs. This package thus has methods like getHeader() which returns the header , getBody() which gives the .xml file . It further supports parsing of .xml file to give its attributes like list of test Drivers, list of Test Code.

PARTITION

Repository

The repository holds the baseline code, test drivers, and test code. We also assume that it has a way to keep a track of dependencies. In project 2 repository is just a folder with not much functionality.

Test Driver:

We have to design the sample test driver for demonstrating the requirements to the graders and TA's. The testDriver derives from the ITestInterface and implements the methods like test(), getLog() . In real life scenario the test drivers are designed by the developers for their respective test code.

Test Code:

Sample test codes are provided for demonstrating the requirements to the graders and TA's

KEY APPLICATION ACTIVITIES

The Figure 2 Activity Diagram shows the processing of a single client request. Each client request will be similarly handled. For every test request by the client new child Appdomain will be created

We give the xml file paths as the command line input. The DemoClient enqueue's the test request message (xml file) in Test Harness's queue. The application Domain manager package in the test Harness decides from the header of the message whether the client needs to retrieve the old test logs or it wants to run test drivers on its test codes.

Retrieval of logs:

If the client wants to retrieve the old test logs, the test harness Queries the database to return the test log. The client request xml file in this case would contain the details of the developer name, code tested including version that would help the test harness to identify which log is to be returned. If the log was found our program would display the logs on the console for making the demonstration of requirements to the graders and TA easy. If the log was not found the function would return Boolean false and inform the client that log were not found.

Run Test Request:

Test request comes in the form of xml file which contains the following details:

- List of testdrivers DLL's to be run
- List of test code DLL's
- The tester's name

If the client wants to run the tests on his test code, the test harness creates a new child application domain and injects a loader in it. The loader loads the required test driver DLL's and test code DLL's and executes the test .It then commands the logger to log the test results in the database. The next activity diagram describes in detail the working of the child application domain.

We assume that we already have a code repository that holds the current project baseline and can deliver test driver dynamic link libraries (DLL's) and test code DLL's on demand. The test driver derives from an ITest interface and supplies a factory class with a static method to create instances of all the classes used by the test. We also assume that the repository provides the facility of keeping a record of the dependencies of the code.

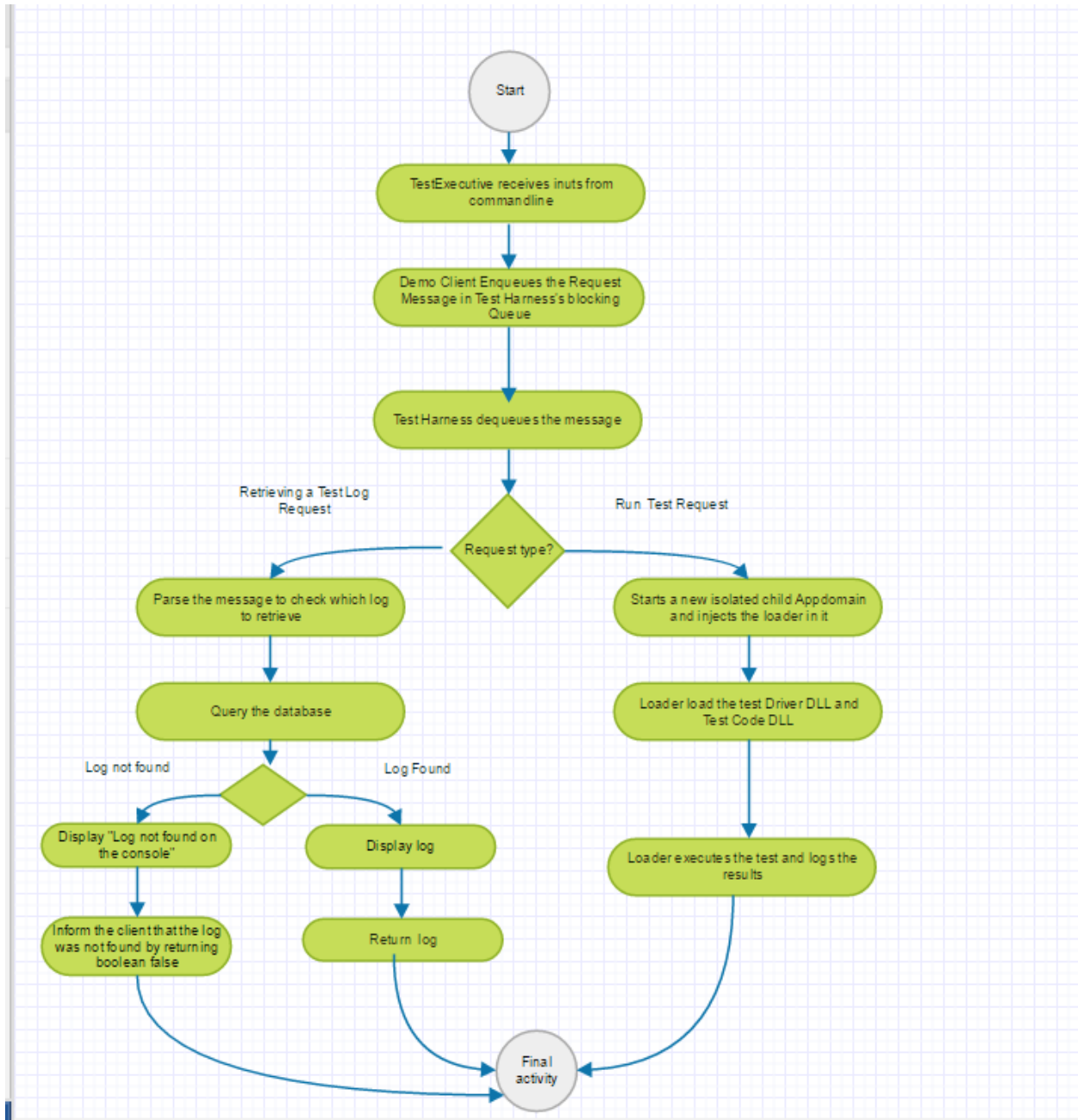


Figure 2 Request Processing Activity Diagram

AppDomainManager and Loader:

The Appdomain manager creates a child APPDomain and injects a loader in it. The loader parses the .xml file to get the list of test driver DLL's, test code DLL's.

It loads these DLL's and executes each test driver on the test code. The test driver derives from the iTest Interface which has the function test() that executes the test and returns the pass/ fail status. It also has a function getLog which returns the string version of the log.

The logger then calls the getLog() function to get the results. It creates a log of name, identity of the test developer, the code tested including version, and time date stamps for each test execution. After the tests are executed the loader informs the application manager that it is done satisfying the test request. The application manager then unloads the child appdomain and thus the DLL's.

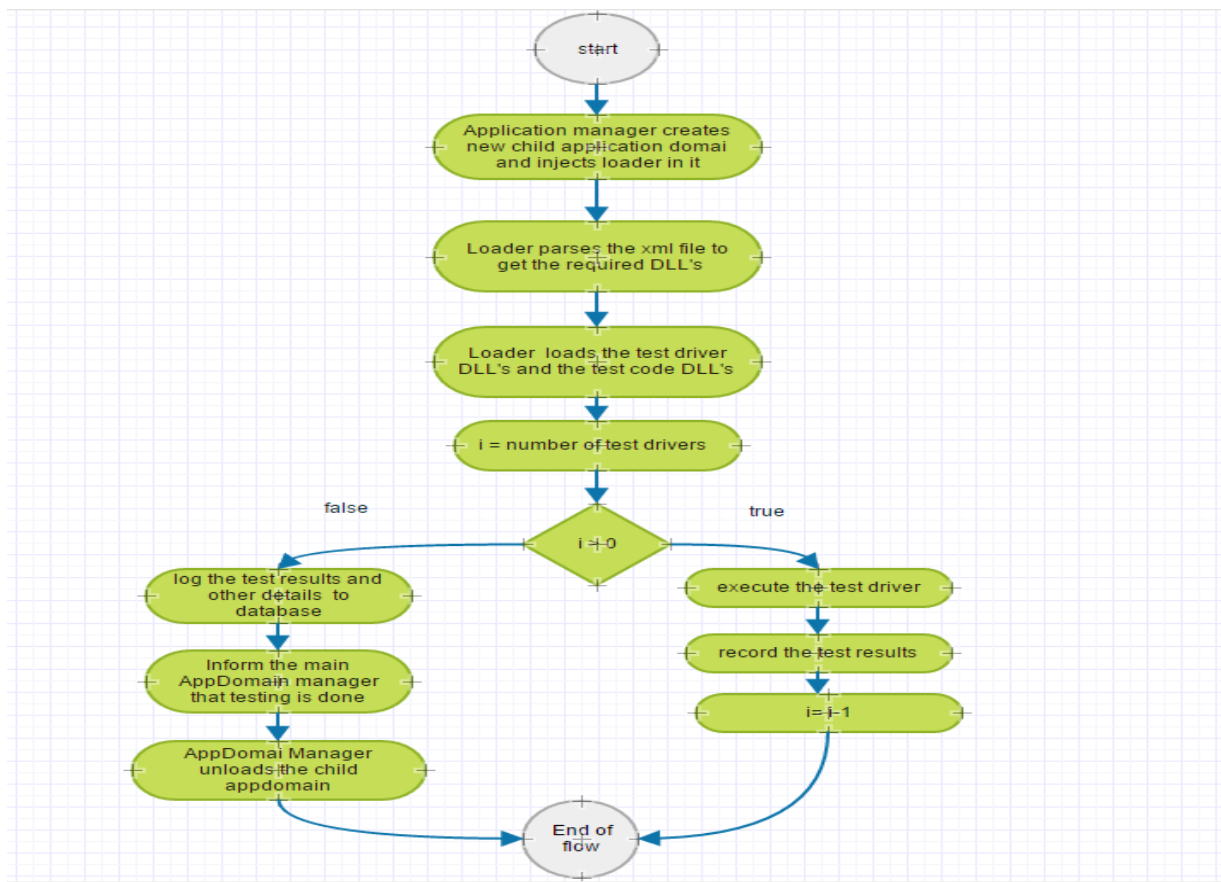


Figure 3. Activity Diagram for the APP Domain Manager.

CRITICAL ISSUES

Ease of use

Ease of use is one of the major concerns for the developers. We should take care that we do not build a very rigid application, which only works for certain cases. Also overly complicated user interfaces discourages the largescale use of the systems. Thus provision of an interface which is simple to use and easy to understand is crucial.

Solution: The code structure should be well designed, various use cases and their impact on design should be considered beforehand. A well designed Graphical user interface which provides the facilities to select the test drivers and the test code is desirable. But in project 2 we are not building a Graphical user Interface. User inputs will most probably be given as command line arguments to automate the testing process for the ease of graders.

Performance

The test harness is going to be one of the busiest servers during qualification testing and final delivery. The qualification testing runs thousands of test cases against entire baseline. The performance of the test harness is therefore a critical issue. Apart from this the test harness should also support multiple clients.

Solution: Multi-threading and sharing of resources would decrease the time required to process a test request. Without multithreading the test might have had to wait in a queue for a long time. The loading of only required dll's also helps in improving performance.

Exception on running test Driver

There might be unhandled exceptions in the test code, which would be encountered while running the test driver. This might take down the entire child AppDomain. Thus the remaining tests would not run.

Solution: We should run each test driver on a separate thread. The test log would make a note that the particular test driver threw an exception.

Security in remotely accessed database used for storing test logs: If we use database which is remotely accessible, multiple user segments will use the application. Security is an issue that needs to be addressed while developing remotely accessible applications.

Project #1 Test Harness

Breaches of security could include damages to the system, loss or theft of data, and compromise of data integrity.

Solution: We must allow only authorized users to access our database. Multiple levels of accesses for managers, developers and QA's depending on their needs should be developed. Users should only be allowed to access as much functionality as they are authorized.

Demonstration

Demonstrating that all the requirements are met to the graders and teaching assistants is one of the critical issue.

Solution:

Printing the outputs to console, which show stepwise execution of the code can demonstrate the requirements to the TA. Thus the .xml message request content, the loaded testdriver and test code, the thread id running in the APPDomain (for project 2 it's the main thread) , The logs showing pass/ fail status , date and time stamp , tester name can be printed to the console to show that the requirements are met.

Inconsistency in input request Message

If the data sent in the xml file is inconsistent, that is it misses certain fields or has a value mismatch, then there might arise a situation where we have insufficient data or the application might not know what to do at all.

Solution: We should provide a schema which is specific to our input style. If it is not followed we should inform the problem to user and ask him to correct it.

REFERENCES

- <http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/pictures/Project4-BlockDiagram.pdf>
- <http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project1-F2016.htm>
- Helper Code provided Dr. Fawcett

CONCLUSION

We can see that a Test Harness is an extremely useful and important tool in the development of large software. But while designing it we should make sure that we address the critical issues and make the application robust and secured.

PROTOTYPE

FileManager

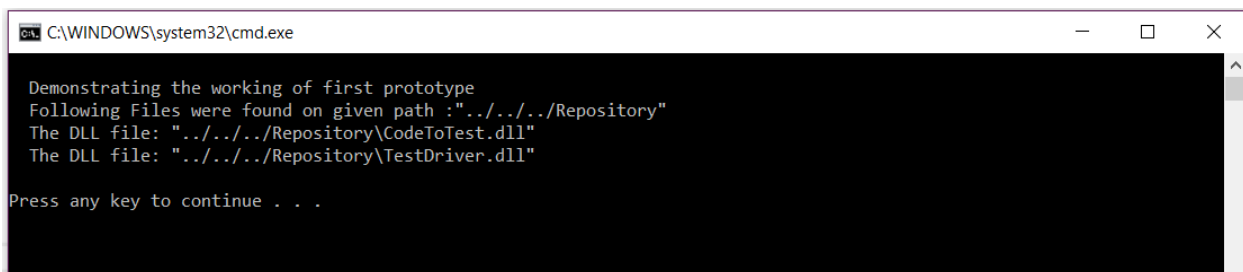
Requirement:

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

This Prototype searches the .dll files from a directory and displays it to the console. This package is just a simple proof of concept. The code developed here would be used by the Loader to search the directory and get the test driver and test code DLL files.

The file manager searches for .dll files in a repository folder and displays the path and filename on the console.

Following is the output of the prototype.



```
C:\WINDOWS\system32\cmd.exe
Demonstrating the working of first prototype
Following Files were found on given path : "..\..\Repository"
The DLL file: "..\..\Repository\CodeToTest.dll"
The DLL file: "..\..\Repository\TestDriver.dll"
Press any key to continue . . .
```

Thus the prototype satisfies the requirement.

Project #1 Test Harness

TestLoader

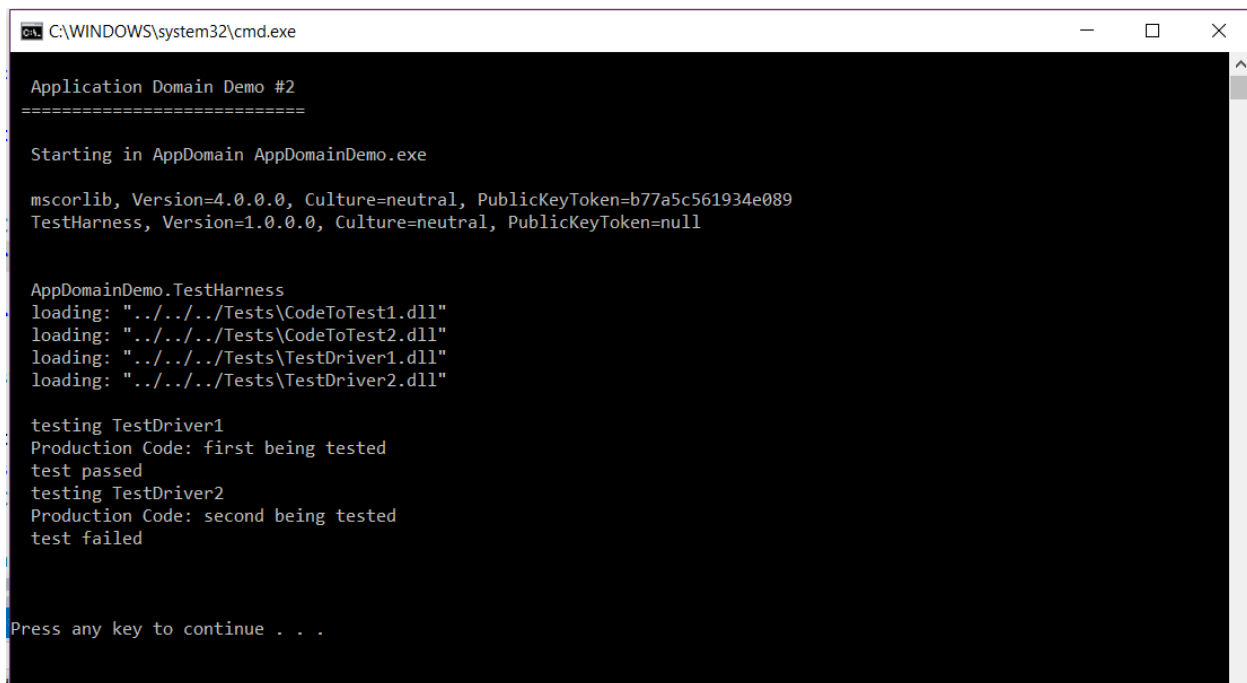
Requirement :

Child AppDomain demo that loads a simulated test library, executes it, and displays the results.

This prototype shows the loading of test driver and test code .dlls and execution of test drivers on the test code in the child Appdomain. This code prototype would be the basis of implementation of the AppliationDomainManager and the loader packages in the test harness.

Here the AppDomainDemo starts a child APPDomain and injects the testHarness in it. The TestHarness is derived from ITestHarness interface. The testHarness() method is called to load and execute the test driver.

Following is the output of the prototype.



```
C:\WINDOWS\system32\cmd.exe
Application Domain Demo #2
=====
Starting in AppDomain AppDomainDemo.exe

mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
TestHarness, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null

AppDomainDemo.TestHarness
loading: "../../../Tests/CodeToTest1.dll"
loading: "../../../Tests/CodeToTest2.dll"
loading: "../../../Tests/TestDriver1.dll"
loading: "../../../Tests/TestDriver2.dll"

testing TestDriver1
Production Code: first being tested
test passed
testing TestDriver2
Production Code: second being tested
test failed

Press any key to continue . . .
```

Thus the prototype satisfies the requirement.