SEPTEMBER 14, 2016

TEST HARNESS OPERATIONAL CONCEPT DOCUMENT (OCD)

BENJAMIN KRAINES
SYRACUSE UNIVESITY
SMA681 Project 1

Table of Contents

1 Introduction

The following is an Operational Concept Document describing the basis for development on a comprehensive automated test harness. The executive summary will provide you with the key takeaways from initial prototyping and architecture work, and the path moving forward. Beyond that, the document delves into the specific system design elements and details which will guide project execution. If you have any questions, comments, or concerns regarding this document, please contact the author Ben Kraines via email at bjkraine@syr.edu.

2 Executive Summary

In general, the Test Harness concept was found to be viable and ready for transition into the implantation phase. The following are the key takeaways from this document:

- Nested .Net Application Domains will hold each test execution in isolation. This technique was successfully demonstrated in a proof of concept application called "ChildAppDomainDemo".
- Test Libraries stored as DLL's can be located and utilized programmatically. This technique was successfully in a proof of concept application called "DLLFinder".
- Critical Issues exist, but solutions to each are detailed here.
- Top level packages are the following:
    - Client/Main Executive
    - AppDomain Manager
    - Messaging
    - Logger
    - Loader
    - Client File Manager

  Each package is described in detail in the system structure subsection.
- Methodologies are in place for development, testing, and deployment

The test harness design is realistic, and ready to be implemented! This document has laid out the path forward in detail towards a completed application.

3 Concept Statement

Testing is a critical piece in the development and lifecycle of a software system. As changes in the code base occur, developers, quality assurance, and managers all have a vested interest in ensuring that the baseline continues to function as these changes are integrated. In large commercial projects where many developers contribute to a single set of interdependent packages, it quickly becomes necessary to automate the process of testing new contributions. This concept document lays out a test harness application which would serve to bootstrap test requests, execute them safely, and report the results back to the development team.

3.1 Requirements

The following are some of the key requirements and concepts for the Test Harness:

- **Automate Tests.** The most basic function of the test harness is to simply automate tests. It must accept requests for tests in the form of an XML file, and queue them up for completion in order. Once the user submits a test request, the harness completes all intermediate steps to provide the users with logs recording the results.
- **Support a continuous integration.** The test harness will be used to verify that changes to the baseline are integrated seamlessly, maintaining functionality in all dependency relationships, direct or indirect.
- **Use .Net Application Domains.** The test harness must sequester individual tests into their own .Net application domains. App domains are essentially an isolated execution space within the main process being run. The major benefit of this approach is keeping any given test's failures and exceptions from affecting the operation of other tests of the harness as a whole. Worst case, the app domain goes down and user gets only a partial log from the failed test, but everything else keeps running.
- **Demonstration Mode.** The test harness must be configurable to demonstrate all of its capabilities and requirements in a scripted demonstration for the customer. This allows for a quick encapsulation and exhibition of what we're aiming to achieve.

4 Guiding Principles

Several guiding principles in the architecture and development process are important to the successful and expedient implementation of the designs laid out in this document. Each one is listed below and briefly described.

4.1 Agile Process

Agile software development is one of the prevailing schools of thought in industry regarding evolutionary and responsive software design. Although the development of this test harness won't be completed in a team, there are many key lessons in the agile manifesto that can be mapped onto this effort. Small iterative cycles are key to a responsive and effective development process. As each package in the Test Harness is built, these mini cycles are an absolute must. The development must be sustainable, and well thought-out. The test harness will be used later in larger continuous integration applications, so it is very important that the design is future-proofed. On the whole, agile provides a set of principles that are highly useful to developers, and those should be rightly applied in this work.

4.2 Concurrent Testing

Testing during the execution of the "mini cycles" discussed above is crucial to making consistent progress on the Test Harness. Construction tests shall be used while writing packages to confirm functionality after each small addition. These must occur along the way – otherwise risk is incurred by adding large untested pieces of code with many potential modes of failure.

4.3 Eating Your Own Dogfood

As development on the test harness begins reaching a critical mass – the test harness itself should start being used to test new packages! There's no better way to validate usability and functionality than being required to use the product in its own development. This practice is dubbed "eating your own dogfood" in industry. You can bet that the developers of Visual Studio are using it as their development environment, and the same principle applies here.

5 Uses and Users

The paramount consideration in the development of any application are ultimately the users of the application. Without the use cases that these individuals provide, the software would literally be useless! Below is a list of the key stakeholders in the development of the Test Harness, and a description of their ultimate relationship with end product. Discussed with the critical user types are the sorts of control inputs they might need on the harness, the information they need to see, and the design implications of their use of the application.

5.1 Developers

Developers are one of the most prolific users of the test harness. They will be requesting tests every single day to make sure that their new changes to the baseline are compatible. In large code bases, much of the testing will occur overnight, so a normal use case might be developers queueing up test requests at the end of the day, and then letting them run overnight, before checking them the next day. In short, the developer needs to do two main functions:

- Submit Test Requests for Execution
- Query the Harness for Log Results

Control Inputs: Devs need an easy way to load in an XML test request. The steps to do so should be straightforward, and it should take a very short amount of time. Additionally, developers need to be able to pull up all the completed test logs, and be able to view them.

Information Required: The most critical information to developers is the results of the tests that they queued up themselves. However, it is easy to imagine a case in collaborative work with other developers or QA personnel in which they might like to open up the results from a test someone else submitted. Therefore, developer mode should feature easy access to query all logs.

Design Implications: Much of the design for this test harness will be centered around the developer uses. Queueing the test requests is critical since many developers means there will be a huge number of requests flowing in. Since developer workflows need to be highly optimized, it needs to be very quick and easy to queue something up or pull a result. This feeds into the design of the user interface, which will be discussed later in the document. Overall, developer uses are a huge driver in many of the design decisions.

5.2 Quality Assurance (QA)

Quality Assurance teams manage the baseline of a large software system. This means QA might run some very demanding integration and regression tests on huge swaths of packages. For this, the test harness needs to be very fast! The actual testing process will be very similar to that of the developer, they log in, queue up some tests, and come back later for the results. It also needs to be easy to load up tests, and they will probably end up running the same tests many times.

Control Inputs: Same as the developer! They need to be able to input XML test requests, and view results afterwards. It is important to note that these XML requests might be much larger than those used by the devs. Previous XML test requests should remain available since QA reuses these quite often.

Information Required: Again, the information needed by QA staff closely parallels that of the developers. They need access to the logs of completed tests. QA values more heavily some test metadata, like developer names and dates. These must be linked to each test for quick identification of code elements that are causing issues in the baseline.

Design Implications: Performance is a large design implication from the QA use cases. Large tests require lots of computing resources. Uptime on the harness should be near 100% so QA can have their biggest tests running around the clock. Parallelizing parts of the code will be crucial to performance. Although initial implementations will be single threaded, eventually the tests will each be running on their own threads – an implementation made very easy by the use of .Net application domains.

5.3 Program Manager (PM)

The program manager has a less direct relationship with the test harness than developers and QA engineers. The PM is more interested in the broad strokes – is the baseline stable, are developers being consistent in their testing, etc. For this reason, the information provided to PM's in the test harness will be filtered and more directly suited to their interest.

Control Inputs: Managers only control input will be to open manager mode on the harness. It will then display an automatically collated set of manager specific information.

Information Required: Metadata including the percentage of tests that have passed, numbers of tests submitted by individual developers, and state of the baseline will all be summarized in the manager view. There will be no additional interpretation needed, it should be a "one stop shop" for PM's.

Design Implications: Besides implementing metadata collection and a manager summary view, there aren't many major design impacts when planning for a manager role in the test harness implementation.

5.4 Test Harness Developer

The Test Harness Developer is just a unique form of the developer user type. As discussed in the "Guiding Principles" section, the developer will use a dogfooding approach to develop the test harness when possible. This means both implementing the test harness packages, but testing them using the harness. This will provide unique perspective on the usability and functionality of the harness.

5.5 SMA Instructor and Teaching Assistants

The SMA instructor and TA's will be using the test harness while evaluating its functionality. This will involve utilizing the demonstration mode, but also applying it to unique test packages in developer and QA roles. They may also evaluate the manager mode to view metadata generation and display.

6 System Structure

System structure is a critical part of developing the operating concept on a new system. This section will layout the technical breakdown of required functionality for the test harness and provide a largely instructional approach for the implementation phase of the project. The system breaks down into six major packages. The Client/Test Executive, Client File Manager, Messaging, AppDomain Manager, Loader, and Logger. Each of these packages will be detailed in depth throughout the system structure section. The following package diagram emphasizes the basic package relationships throughout regular program execution.

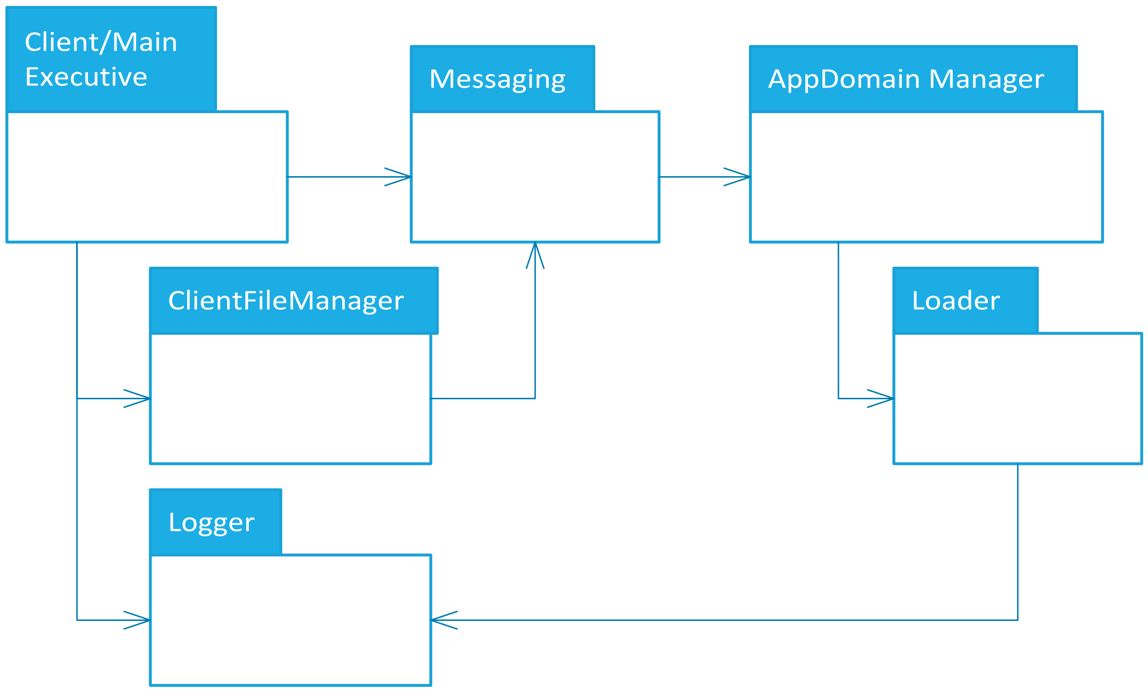

Figure 1 - Test Harness Package Diagram

6.1 Client/Test Executive

The executive package is the top-level code block responsible for bootstrapping everything that happens in the application. In the test harness, it will open a command line user interface, and perform functions using the other packages as directed by the user to meet the requirements and use cases discussed above.

The executive first references the Client File Manager package when starting up to initialize folders for Test Requests, DLL's, and Log files. After those steps are complete, it proceeds to display user modes. The initial actions of the client/test executive will be to direct the user to the appropriate functional space. This is where developers, QA, and managers diverge into parts of the interface specific to their needs. So the very first option the user is provided in the command line is to decide whether to enter developer, QA, or manager mode. Beyond that, the user continues to be prompted by the executive on how to proceed in each mode. This initial process conducted by the test executive is shown in the following activity diagram.

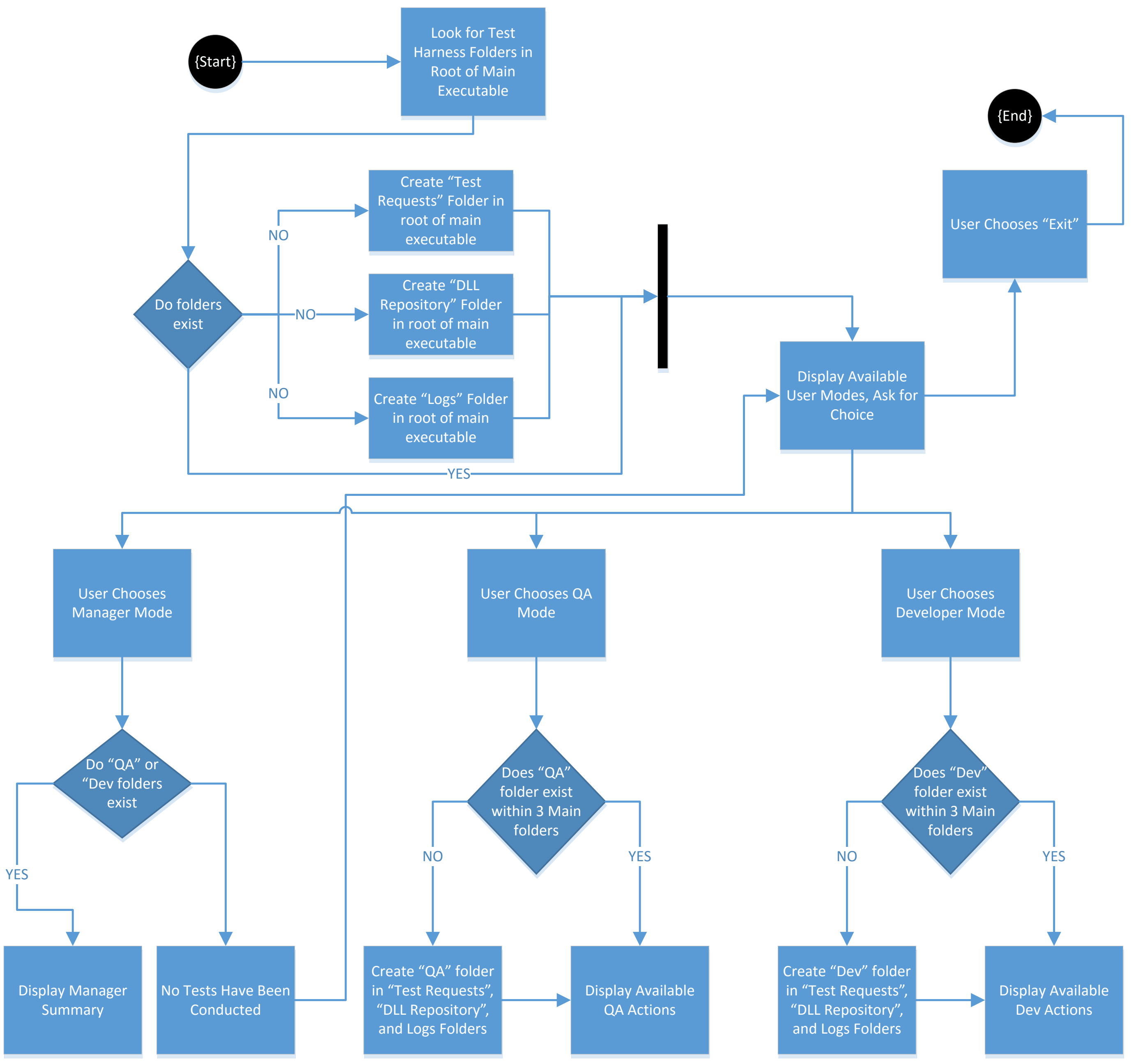
{Start}
Look for Test Harness Folders in Root of Main Executable
{End}
Create "Test Requests" Folder in root of main executable
NO
User Chooses "Exit"
Do folders exist
NO
Create "DLL Repository" Folder in root of main executable
NO
Create "Logs" Folder in root of main executable
Display Available User Modes, Ask for Choice
YES
User Chooses Manager Mode
User Chooses QA Mode
User Chooses Developer Mode
Do "QA" or "Dev folders exist
YES
Does "QA" folder exist within 3 Main folders
NO
YES
Does "Dev" folder exist within 3 Main folders
NO
YES
Display Manager Summary
No Tests Have Been Conducted
Create "QA" folder in "Test Requests", "DLL Repository", and Logs Folders
Display Available QA Actions
Create "Dev" folder in "Test Requests", "DLL Repository", and Logs Folders
Display Available Dev Actions

Figure 2 - Initial Activity Diagram

The user follows the procedures shown in **Figure 2 - Initial Activity DiagramFigure 2** to receive a list of available actions, which will be further detailed later in the structure section. Anytime in the activity diagram that file checks or creations are happening, the test executive is kicking over to the client file manager for handling those interactions.

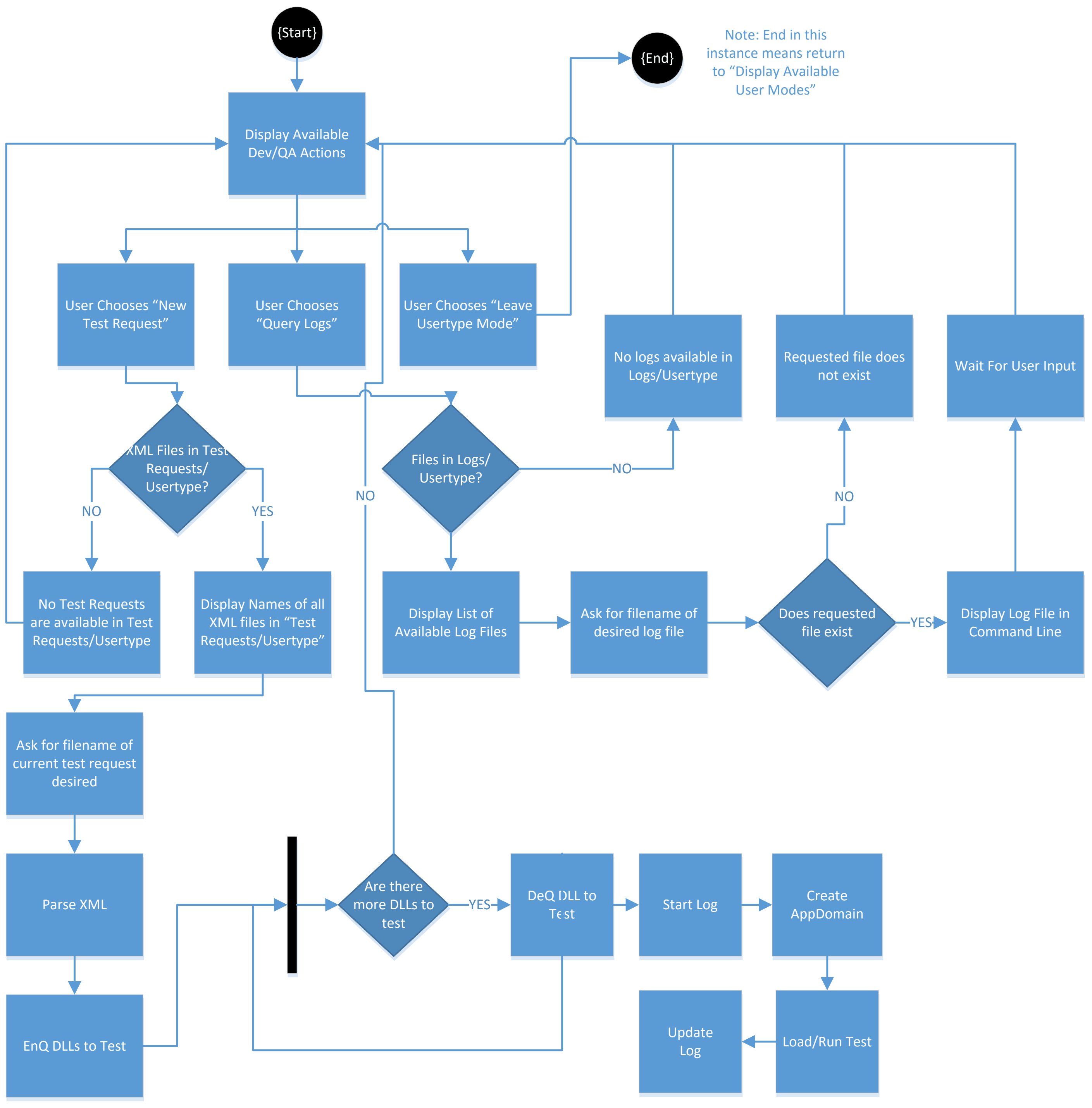
{Start}
{End}
Note: End in this instance means return to "Display Available User Modes"
Display Available Dev/QA Actions
User Chooses "New Test Request"
User Chooses "Query Logs"
User Chooses "Leave Usertype Mode"
No logs available in Logs/Usertype
Requested file does not exist
Wait For User Input
XML Files in Test Requests/ Usertype?
NO
YES
NO
Files in Logs/ Usertype?
NO
NO
No Test Requests are available in Test Requests/Usertype
Display Names of all XML files in "Test Requests/Usertype"
Display List of Available Log Files
Ask for filename of desired log file
Does requested file exist
YES
Display Log File in Command Line
Ask for filename of current test request desired
Parse XML
Are there more DLLs to test
YES
DeQ DLL to Test
Start Log
Create AppDomain
EnQ DLLs to Test
Update Log
Load/Run Test

Figure 3 - Developer/QA Activity Diagram

The activity diagram above in Figure 3 shows a more detailed breakdown past the initial activity diagram of how the flow of events occurs when actually queueing up tests and executing them.

6.2 Client File Manager

As mentioned above, the test executive utilizes the client file manager to conduct operations with regards to Test Request, DLL, and Log file storage. In order to complete these tasks effectively, the Client File Manager must exhibit the following functionality:

- Instantiate the following directories as needed:
    - Root/Test Requests
        - Root/Test Requests/Dev
        - Root/Test Requests/QA
    - Root/DLL Repository
        - Root/DLL Repository/Dev
        - Root/DLL Repository/QA
    - Root/Logs
- Query Root/Test Requests for a list of all .xml files
- Query Root/DLL Repository for a list of all .dll files

Other packages may use the query functions in their implementation. For example, the Messaging package should verify that all .dll files mentioned in the XML test request actually exist in the DLL Repository.

6.3 .Net Application Domain Manager

One of the prime requirements for the test harness is that it employ .Net application domains for test isolation. As such, a package is required to set up these application domains each time a new test is started. It also must unload the application domain once the test is finished.

Parsed Tests flow into the domain manager queue with the following:

- Test Name
- List of DLL's

The domain manager then performs the following steps:

1. Dequeue the First Test
2. Create new child AppDomain (using AppDomainSetup to define the basic parameters)
3. Pass off list of .DLL files and AppDomain to Loader
4. When Loader call returns, unload the domain

In the future, there may be additional steps in order to create a new thread for each app domain to enhance parallelism and performance. This functionality was prototyped in the child appdomain code discussed in the appendix.

6.4 Loader

The loader is a package designed to load DLL's into newly created AppDomains from the AppDomain Manager. It uses the concept of remoting to run tests based upon the ITest interface, before piping the output of those tests into the logger, and return back to the AppDomain manager for unloading. The loader must perform the following actions on an incoming AppDomain for testing.

- Take a list of DLL files and individually load them into the AppDomain
- Use remoting to call the test function on a specified DLL defined in the test request. Test function signature defined in the ITest interface that each package being tested is required to implement
- Send a test success Boolean to the logger
- Send any test output to the logger by using the getLog() function on the ITest interface
- Catch any exceptions that occur during the test and send them to the logs
- When execution is finished, return from bootstrapped function that was called by AppDomain manager so that it knows when to unload current AppDomain

Essentially, the loader package is doing the heavy lifting inside the child AppDomain of actually running each test that is sent in from the developers/QA.

6.5 Logger

The logger package is responsible for recording logs for each test executed by the harness. This is a key functionality as developers and QA need to review logs to make sure tests passed, and to troubleshoot packages when the tests do not pass. Logger must perform the following functions:

- Create a new log file given the name of the developer and the current time
- Append text to the log file
- Markup key metadata at the top of the file
  - Test Name
  - Pass/Fail
  - Developer Name
  - Date/Time
  - DLL’s Involved
- Close log file when test is complete
- Return only metadata from a log file
- Return all content of a log file

The logger will be notified of a new test when the loader begins to run, which is when it will create the new log file.

6.6 Messaging

Messaging is the package within the test harness that parses XML test requests from the developers/QA personnel into tests to queue up in the AppDomain manager. XML Requests from the devs/QA must include the following items:

- Test Name
- Developer Name
- List of DLL’s to load

The messaging package can then parse these items into a test struct that it can enqueue for the AppDomain manager to handle.

6.7 Demo Client

A demonstration client is necessary in light of the requirement listed in the concept statement section. Essentially, a demo client would be a scripted command line application that carries out a single test, explaining the package interactions and steps to load a test request, create the application domain, use the loader to bootstrap the test itself, and show the log outputs throughout the process. This would run outside of the normal command line user interface, since its purpose is to simply demonstrate what is happening without the user needing to interact in any way

7 Critical Issues

This section details the critical issues facing the development of the test harness which might pose risk to project execution and final implementation quality for the users. For each critical issue/sub-issue, there is a brief description, the design impact, and solutions for the issues.

7.1 Performance

Performance is a key consideration in the design of the test harness. It must be capable of processing huge numbers of packages, and it needs to be able to do so in a reasonable amount of time. Especially for the QA users who run large baseline-wide tests, performance is a critical concern.

7.1.1 Critical Path: Tests

Description: The critical path for this application must be the tests themselves. Instances of tests waiting on the harness for any significant delay must be eliminated.

Design Impacts: Code should be optimized to queue and start tests efficiently. Parallelism should be embraced anywhere it becomes feasible, especially in later stages of the project. This will be discussed more below.

Solutions: Harness performance can be examined through some simple observations in the AppDomain manager. A quick timer on execution inside vs. outside the child appdomains would provide some valuable input on whether there are significant delays outside of tests in comparison to test execution time. This would help guide some performance improvement work where it is needed if delays are experienced as well.

7.1.2 Threading

Description: Later in the project, threading will be implemented to increase parallelism and performance for each AppDomain. Threads will be initiated in the AppDomain manager at domain creation time.

Design Impacts: Thread safety will become a concern later on when threads are added. Any objects accessed by multiple threads become vulnerable to thread safety violations.

Solutions: Thread safe implementations will be essential, especially in key data structures like the Test Request queues, and AppDomain callback functions. Blocking queues, for example, will prevent multiple access issues with the queue.

7.1.3 Scaling

Description: Initial development on the test harness must account for the fact that the size of tests, number of users, deployment scenarios etc. might expand rapidly once in use.

Design Impacts: Ensure no hardcoded limits exist on size/use functionalities. Leave room for threading to enable parallel deployment later on when performance is needed to scale the number/size of tests running.

Solutions: Assess dependencies that might prevent smooth scaling of the system. Prepare for massively parallel deployment by allowing for threading appdomains. Perform stress tests on the system to gauge preparedness for scaling.

7.2 Time Costs

Description: One concern in development of the harness is the time commitment required to complete the actual package implementations. It will admittedly probably take a large chunk of time.

Design Impacts: None, design shouldn't be compromised in functionality/robustness due to tough time constraints in this case.

Solutions: Plan well ahead of the deadline, start package development immediately, work incrementally, test early and often.

7.3 Ease of Use

It is very important to consider the ease of use of the test harness to all users. A carefully crafted user experience that makes using the harness effortless should be baked in from the start.

7.3.1 User Interface

Description: A very important aspect of the test harness that will be tough to get right is the actual interface via which the users will control the test harness actions.

Design Impacts: In this design, the UI will be a simple command line menu interface which presents the user options. This design is driven by ease of implementation, and the application. Developers and QA users don't necessarily need a beautifully crafted GUI. Something simple and easy that lays out the options and allows them to work quickly and efficiently is preferable.

Solutions: Provide users the following in the command line menus:

- A List of available options, each indexed by a number
- Pressing the number given by the option will execute or select that item

See the example below for selecting an xml test request to run:

```
The following XML Files are available for use as Test Requests. Enter the index of the Test Request you would like to send to the Test Harness
=====================================================================
0. Test1.xml
1. Test2.xml
2. Test3.xml
```

The user would then simply press the number of the index on the keyboard, and the test harness would take it from there. This methodology will be implemented in each menu.

7.3.2 Information Filtering

Description: The distinction between data and information is a critical one for a test harness which could contain huge volumes of test data in the form of logs, metadata, and results. It is necessary to filter the data from the test harness in an intelligent way in order to make the information that the users see actually useful and easy to interpret.

Design Impacts: Information filtering was a key design driver in the manager interface. Devs/QA need lots of technical information from specific tests, and they might see unfiltered log files. However, managers really only need to see the metadata. Baseline pass/fail performance, who is running tests, and how many, etc.

Solutions: Intelligently filter information displayed in the UI. Let's use the manager as an example:

If the user selects manager mode, it automatically displays an at a glance summary of the current test results in the logs directory, which tests have failed, who is responsible, and which developers have submitted tests/how many. This removes all barriers from the manager quickly getting the information they need. It might look something like this:

```
========================================================================
=== Test Harness Status Summary ===
========================================================================
Baseline Status: ## out of ## Passed, ##.#% Success
Failing Tests:
----------------------------
FailedTest1.xml | Dev Jim Smith
FailedTest2.xml | QA Jack Ryan
----------------------------
Total Number of Recent Tests: ###
Team Activity:
Dev Jim Smith | ##  Recent Tests, ##.#% of  submitted tests
Dev John Doe  | ##  Recent Tests, ##.#% of submitted tests
QA Jack Ryan  | ##  Recent Tests, ##.#% of submitted tests
```

7.4 Incorrectly Formatted Test Requests

Description: One major difficulty that the test harness might experience is incorrectly formatted test requests. If developers get something wrong in the generation of these .xml files, the parser would fail to be able to enqueuer that request.

Design Impacts: The user needs to be notified immediately when parsing fails. Specifically, the messaging package should have a way to communicate to the user immediately that the file is formatted incorrectly, allowing them to fix the error and retry gracefully.

Solutions: The messaging package shall have a graceful escape sequence from XML parsing sequences, which carries with it a tag or information set on what part of the parsing failed. This will then be displayed to the user in the console, allowing them to troubleshoot the issue and try again without restarting the test harness application again.

7.5 File Conflicts & Organization

Description: Another major difficulty in the operation of this test harness is the management of files, specifically for testing and demonstrating the client this may become a difficulty. The issue is precipitated by the fact that we are simulating a repository throughout project 1, where a windows directory is serving as the repository. Versioning in this case is essentially unworkable – and multiple copies of the same code can't exist in the same directory without serious complication being introduced.

Design Impacts: Folders are designed to be separated by team: one folder for developers, one for QA. This will assist in separating tests into their respective use categories. Beyond that, file conflicts and organization will need to occur outside the purview of the harness.

Solutions: For demonstrations, some simple scripts to place the right .xml and .dll files in the right places might be helpful. This would expedite the process of setting up tests, and usage for the script could be given in a simple readme for the user of the demonstration code to run before starting up the harness demo.

8 Appendix I: Prototype Code

In support of this operational concept document, two code prototypes were built to prove that some of basic underlying principles of this design are workable. The first was an application that gets a root file path from the user, and displays a list of all .dll files in that path. The second was a program that spawned a child .Net application domain, then ran a simulated test within that domain. Each prototype can be found in the zip folder that this document is submitted in, and is discussed in detail below.

8.1 DLL Finder Prototype

A DLL Finder prototype was written to prove out the viability of searching the repository folder for test packages. This is a critical necessity, as users might like to see a list of available .dll files to test, and the loader needs a way to find these files when it is time to load them into the child appdomain. The implementation of the DLL Finder utilizes a .Net tool called FolderBrowserDialog, a helpful windows explorer extension that gives the user an easy GUI with which to select the directory they would like to search for DLL's. This dialog prevents any errors in inputting a root path. It either outputs a valid path or no path, eliminating a failure vector for the application. Once a path was specified, another c# function was leveraged to locate the dll's within the path. Directory.GetFiles() performs a recursive path search, creating a string array that holds the filenames of each DLL that is found. One challenge in building the DLL Finder was handling access restrictions. When searching a large directory (i.e. C:/), the recursive search would encounter folders that the user does not have access to, which throws an UnauthorizedAccessException. This was gracefully handled in a try/catch block, informing the user of the access restrictions in their specified root. The user can then re-specify the root and try again. Note that this issue won't really effect the test harness implementation of DLL searching, as the repository folders will exist in user directories that are not access protected. However, it might be prudent to plan exception handing anyways just in case.

Overall, the DLLFinder prototype was a success. It demonstrated a capability for searching and finding .dll files which will be critical in the Client File Manager package for providing DLL's to the loader as directed by test requests.

8.2 Child App Domain Prototype

A Child AppDomain Prototype Demo was developed in support of this operational concept document to prove the viability of building individual isolated AppDomains for each test being run. The prototype was set up much like the demonstration client in the test harness will be – a scripted command line execution of the key functionality, with description statements printing out along with the results explaining what is occurring. The prototype features three packages. One is a modified implementation of the ITest interface that will be implemented in the test harness, with a function added to get the AppDomain of execution. This will be helpful in proving that tests are actually occurring in a child domain. The second package is a simulated test library called CodeSpoofer which generates a simple string. This is the library that we are testing in the demo, and it implements the ITest interface. The final package is the ChildAppDomain package, which sets up the child appdomain, loads the simulated test library, executes the test, then unloads the appdomain, and displays the whole process to the user.

Overall, the Child AppDomain demo was a success. It showed that child AppDomains can be used to execute tests on a test library, a central concept to the test harness. The code can be found in the same directory as this document when submitted.