TEST HARNESS

Operational Concept Document for Project 1

Version 1.0
14th September 2016

CSE 861 – Software Modeling and Analysis
Instructor: Jim Fawcett

Sahil Surendra Gupta
SUID: 74146-3736
sagupta@syr.edu

CONTENTS

1. Executive Summary 1

2. Introduction 2

2.1. Objective and Key Ideas 2

2.2. Obligations 2

2.3. Organizing Principle 2

3. Users 3

3.1. Developer 3

3.2. QA 3

3.3. Manager 3

3.4. TA / Professor 4

3.5. Project 4 4

4. Structure 4

4.1. TestExec 5

4.2. UI 5

4.3. RequestQueue 5

4.4. AppDomain Mgr 5

4.5. AppDomain 6

4.6. Loader 6

4.7. XmlParser 6

4.8. DatabaseEngine 6

4.9. Logger 7

4.10. Scheduler 8

4.11. ReportGenerator 8

4.12. Test Request 8

4.13. TestDriver 9

5. Activity 9

5.1. Lifecycle of Test Request 9

5.2. Processing Test Request 11

6. Critical Issues 12

6.1. Ease of Use 12

6.2. Performance 13
6.3. Security 13
6.4. Reliability and Load Balancing 13
6.5. Language Interoperability 14
6.6. Versioning and Backward Compatibility 14
6.7. Demonstration 14
7. References 15
8. Appendix 15
8.1. Prototype - File Manager 15
8.1.1. Design 15
8.1.2. Conclusion 15
8.2. Prototype - Child AppDomain Demo 16
8.2.1. Design 16
8.2.2. Conclusion 16

FIGURES

Figure 1 - Package Diagram 4
Figure 2 - Sample Database 7
Figure 3 - Sample TestRequest 9
Figure 4 - Activity Diagram showing Lifecycle of Test Request 10
Figure 5 - Activity Diagram showing Execution of Test Request 11

1. Executive Summary

The following Operational Concept (OCD) Document illustrates the concept of a Test Harness. It explains the uses and design of a test harness and the various critical issues once can expect to encounter while implementing the test harness. The OCD also attempts to propose solutions to the listed critical issues. Finally, it describes two code prototypes, that helps demonstrate the working of the test harness.

The test harness uses application domains to keep test requests in an isolated environment. It uses queuing to serialize input requests and serially processes the tests requests one after another. Each test execution also generates logs summarizing test result. The test harness maintains a database which helps support querying of past test results. Users can schedule tests to execute at particular time intervals, event like build generation or dependent code check-in.

The key components of the test harness are:

- TestExecutive that accepts test requests and en-queues them
- AppDomain Manager that de-queues test requests and creates application domain
- AppDomain that executes individual test request
- Schedular that allows scheduling of test cases
- ReportGenerator that displays summary of test execution, its status & exception details if any

Following are some of the issues to be considered while implementing the test harness. Potential solutions to these problems are further discussed in the document.

- Ease of use of the test harness by users
- Performance of the system with respect to number of users and queries it can handle
- Security issues in controlling unauthorized access to logs and test information
- Reliability of the system with respect to server downtime
- Load balancing for teams in large organization
- Language interoperability to provide support for testing code written in other languages
- Providing backward compatibility to support future versions of the test harness
- Demonstrating successful completion of project requirements

The test harness can be used in all kinds of testing scenarios like unit testing, smoke testing, integration testing, regression testing, construction testing, etc. With the use of test harness, we can expect an increase in productivity of the team due to automation of testing. With a rigorous test plan scheduled, we can be assured that the software baseline is always in a working condition and ready for production.

2. Introduction

A test harness is a tool that allows automation in software testing. It supports continuous integration among many developers to build a software baseline.

The primary use of a test harness is to facilitate a user's testing process. The users of the system are mainly developers and QA teams that will continuously test modules to ensure functionality of code. In addition, Managers will use the system to monitor development progress. TA's and professor will use the system to grade the project.

The test harness will be implemented in C# using .NET framework class libraries. To isolate tests from each other the test harness will create and run each test in its own Application Domain.

2.1. Objective and Key Ideas

1. Automate testing process
2. Generate report on testing progress
3. Reduce repetitive testing by providing TestDrivers
4. Allow scheduling of tests at specific time intervals and events
5. Reduce time required for testing
6. Reduce staff required for testing

2.2. Obligations

The primary obligations of a test harness are

1. Provide well defined structure for test request
2. Accept test requests from client in XML format
3. Process each test request in an AppDomain
4. Provide a queue for serializing multiple test requests
5. Log results of each test execution
6. Support client querying of past test requests

2.3. Organizing Principle

1. Each test request must be executed in a new application domain
2. The version number in the test request XML file and the corresponding database entry indicates the version of test harness it is intended to execute on
3. Each test execution should generate a log summarizing the test results

3. Users

This section describes the various users of the test harness and explains their purpose and interaction with the test harness.

3.1. Developer

The primary user of a test harness is a software developer. The test harness allows a developer to automate his testing process by running test drivers. Test drivers are stubs or small programs that a developer writes to interact with the program being tested. They are primarily function calls which provide necessary arguments to functions that are being tested and validate output on return. Once these test drivers are implemented, the developer can then use them over and again to test the same piece of code, as long as the input and output signatures don’t change. This saves time to setup the test environment by testing functions with default arguments to ensure its functionality. The same test drivers can also be used during integration and regression testing.

Test harness is also useful to developers when a product is being developed as a part of a large organization. The developer can design the test drivers and commit them to a software repository. If there are any changes made to a dependent code in the repository, the test harness should trigger all test drivers which are linked to the dependent code. If any of the test fails, then the developer will be notified and can make suitable changes to fix the code. Thus the test harness will ensure that the software baseline is always working.

We, as graduate students, will be using the Test Harness as a developer to test code and prototypes that we develop during the course.

3.2. QA

The test harness can be used for quality assurance by the QA team. A test harness will help testers to design test cases and run them as a part of smoke testing. Test Drivers can be built to ensure that the most basic functionality of the program is working and QA’s can run these test drivers after every build is generated. Special test drivers to test the functional requirements of the system can also be built to help QA’s test the system.

3.3. Manager

Managers can use the test harness to look at the overall progress of development by analyzing test results. It helps them get an overview of how many tests are passed or failed for their team in a fixed time interval. This is particularly useful during product release, when managers would like to see majority of tests being successful nearing to the release date. If tests continue to fail, the managers can then take corrective actions. Managers can also see individual developers test status which will be useful for performance evaluation.

3.4. TA / Professor

The Teaching Assistants and Professor will use the system to check if all requirements of the project are satisfied or not. They can then accordingly grade the project.

3.5. Project 4

The project described in this OCD can be further extended to project 4 where we build a test harness with remote client. The logic to process a test request can be used as is and additional multi-threading capability can be added to improve performance.

4. Structure

The basic structure of Test Harness is shown below in package diagram. The main components are TestExec which controls execution of the program and the flow of test request and AppDoman which executes individual test requests. Almost every component either interacts with either of these two modules.

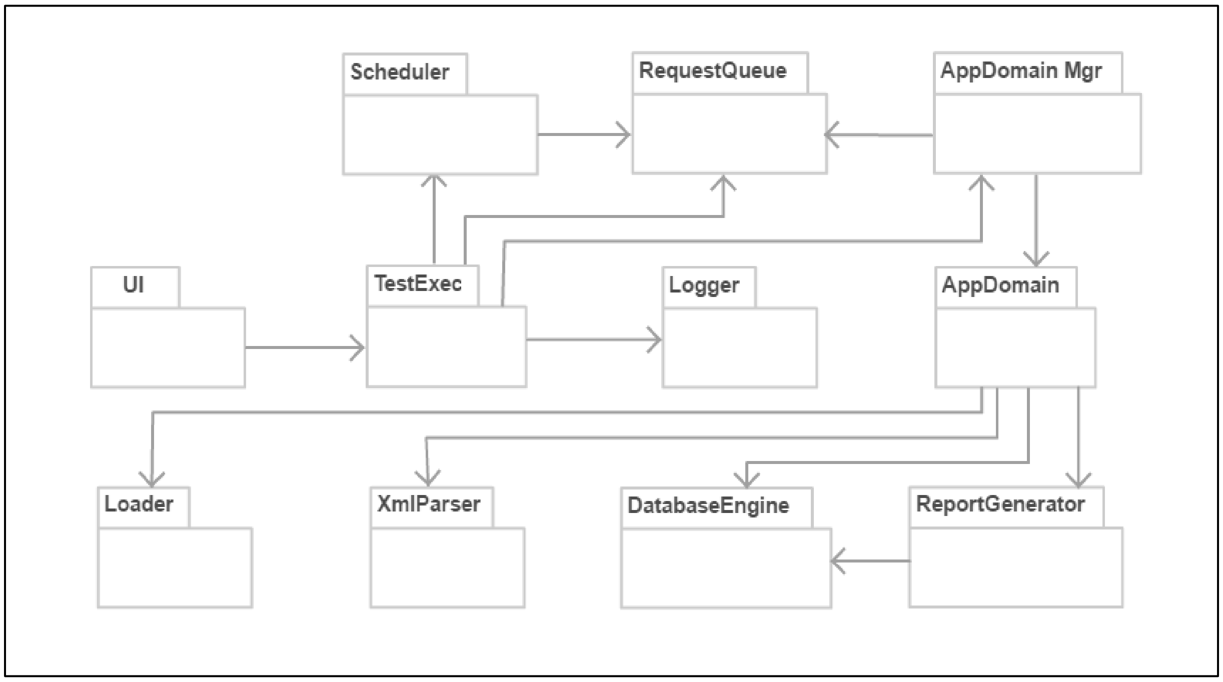


Figure 1 - Package Diagram

4.1. TestExec

The TestExec is the entry point into Test Harness. It controls the main components of the test harness by receiving input from UI and queuing them for the AppDomain Manager to process the TestRequest. It is responsible for controlling the main components in the program; namely the queue, AppDomain manager, scheduler, and the logger. The TestExec remains alive throughout the lifetime of the test harness.

4.2. UI

The User Interface module is responsible for interacting with users of the system and receiving input from them. Developers and QA's will be the main users of the system and will present inputs in form of TestRequest which is an XML file that contains details of the test to be performed. Managers would only monitor logs to check progress of passed and failed test cases.

In a real world Test Harness, we would want a Graphical User Interface to present available options to the users. For developers and QA, it should present a file explorer which allows them to browse and select multiple TestRequest XML files. It can also show status of his past tests executed and give him options to schedule tests. For Managers, it should present a dashboard which contains number of tests passed or failed for his team and customized options to view status for selected time duration, individual developers test status, and status for individual components of the project.

For this project we limit the scope of input to developers and QA. We will implement a command line interface to accept test request. Optionally we can configure a directory path from which the test harness will read TestRequest files.

4.3. RequestQueue

The RequestQueue acts as a buffer for incoming TestRequests. The TestExec accepts TestRequest from UI and adds it to the RequestQueue. The AppDomain Manager monitors the RequestQueue for pending TestRequests and spawns a new AppDomain to process the TestRequest. Ideally want the RequestQueue to be a blocking queue which blocks the AppDomain thread when there are no more requests to be processed.

4.4. AppDomain Mgr

The Application Domain Manager is responsible for reading TestRequest from the RequestQueue and creating new AppDomains to process the request in an isolated environment. It is also responsible for destroying the AppDomain when the test is completed.

Ideally we would want to have a multi-threaded environment, where the AppDomain Manager continuously reads TestRequest and creates new AppDomains to process it. Thus we will have multiple tests running in parallel under their individual AppDomains. This may create synchronization issues while writing to database which needs to be addressed when implemented. Currently we limit the scope of this project to running tests serially to avoid synchronization issues.

4.5. AppDomain

This is the isolated execution environment where TestDrivers are executed. The AppDomain manager creates an AppDomain for each TestRequest and passes the TestRequest into the newly created AppDomain. The AppDomain then calls the XML parser to decode the TestRequest and identify TestDrivers and TestCode. It then calls the Loader to load the assemblies into the AppDomain. All the TestDrivers from a TestRequest are executed in this AppDomain serially and status of each TestDriver is logged into database. The main advantage of running tests in AppDomain is that any exceptions generated form the test are not percolated to other parts of the program, particularly to the TestExec and AppDomain Mgr. This ensures that other tests requests running in the system are not impacted by exceptions generated while executing current test driver.

4.6. Loader

The loader loads assemblies required for execution of the test into the AppDomain. The names of the assemblies are extracted from the TestRequest by the XmlParser. The loader then uses reflection to load assemblies in particular AppDomain.

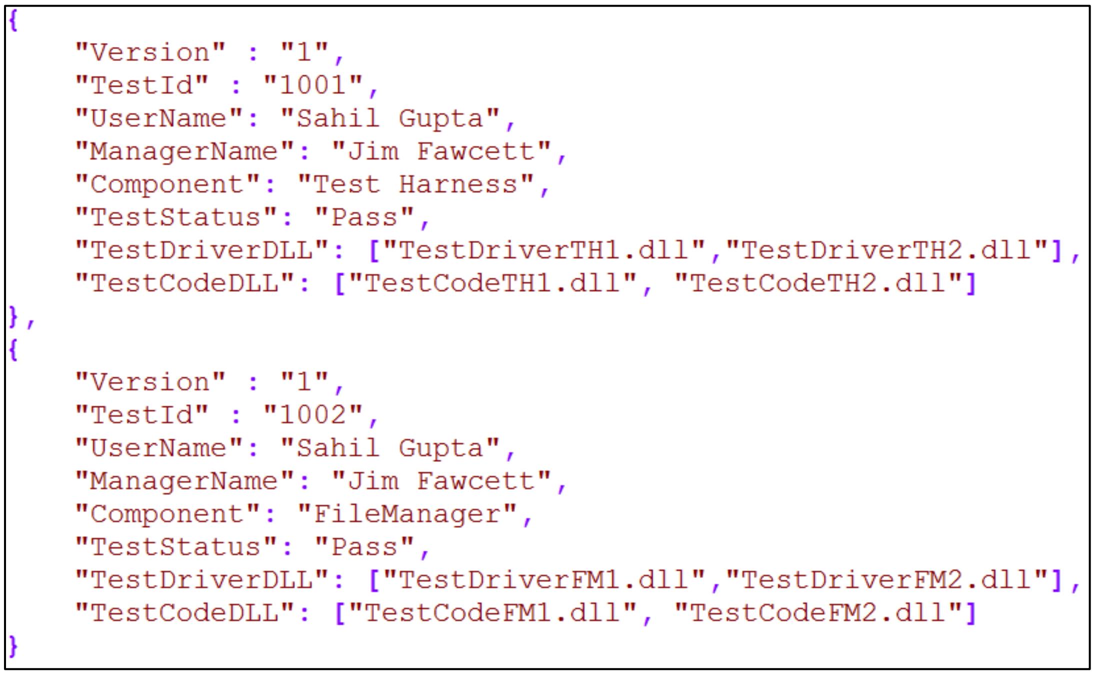
4.7. XmlParser

The XmlParser is responsible for parsing the XML file to extract user and assembly information from it. It creates an object which contains all information that the AppDomain will need to complete the TestRequest.

4.8. DatabaseEngine

The database engine manages a database which stores information of all past test requests executed by the test harness. Using this database, the managers can view status and progress of all users running tests on the Test Harness. The database is implemented using MongoDb.

The database may include fields such as TestId, User Name, Manager Name, Component, Status, TestDriverDLL and TestCodeDLL. The ManagerName and Component field can be used by Managers to get a summary of test results of his employees or status of progress on a particular component.



```
{
    "Version" : "1",
    "TestId" : "1001",
    "UserName": "Sahil Gupta",
    "ManagerName": "Jim Fawcett",
    "Component": "Test Harness",
    "TestStatus": "Pass",
    "TestDriverDLL": ["TestDriverTH1.dll","TestDriverTH2.dll"],
    "TestCodeDLL": ["TestCodeTH1.dll", "TestCodeTH2.dll"]
},
{
    "Version" : "1",
    "TestId" : "1002",
    "UserName": "Sahil Gupta",
    "ManagerName": "Jim Fawcett",
    "Component": "FileManager",
    "TestStatus": "Pass",
    "TestDriverDLL": ["TestDriverFM1.dll","TestDriverFM2.dll"],
    "TestCodeDLL": ["TestCodeFM1.dll", "TestCodeFM2.dll"]
}
```

Figure 2 - Sample Database

4.9. Logger

The Logger implements a logging mechanism to help debug the test harness. It categorizes logs into its level of importance to the developer. For our project we will have the below defined log levels

- Level 1: Audit

This will log all results of all major components of the Test Harness. For instance, starting of individual components, one log per TestRequest summarizing the results of the execution.

- Level 2: Critical

Any critical error encountered while the Test Harness operates will be logged here. These are errors that impact the functioning of the test harness.

- Level 3: Warning

Warning messages which do not impact the functioning of the Test Harness may be of importance during debugging the Test Harness.

- Level 4: Info

Any other info that the developer may wish to log for debugging individual modules.

4.10. Scheduler

The Scheduler is an optional component of the test harness which will be of use in a real world scenario. The job of scheduler is to look at changes in repository, and if there is a change in a module, to run all TestDrivers which are dependent on that module. For this, the TestCodeDll field of the database can be used to identify which TestDrivers are linked to a DLL.

With this, any new piece of code checked in triggers testing of dependent code to see if the new code breaks any existing functionality. If it does, it notifies all the involved resources of the bug which came out of the new code. Accordingly, either the developer of the new code or the author of existing code can accommodate the new change as soon as possible without affecting any other member of the team. Having a scheduler do this job eases the task of regression testing on the team and saves lot of time for QA and developers to find out bugs as and when they are encountered.

The scheduler also gives options to developer to schedule test cases to execute at various time intervals, so that test drivers are at fixed time intervals to ensure the code is always in working condition. QA's can schedule basic test drivers once every build is generated, to ensure basic functionality of the system is working.

4.11. ReportGenerator

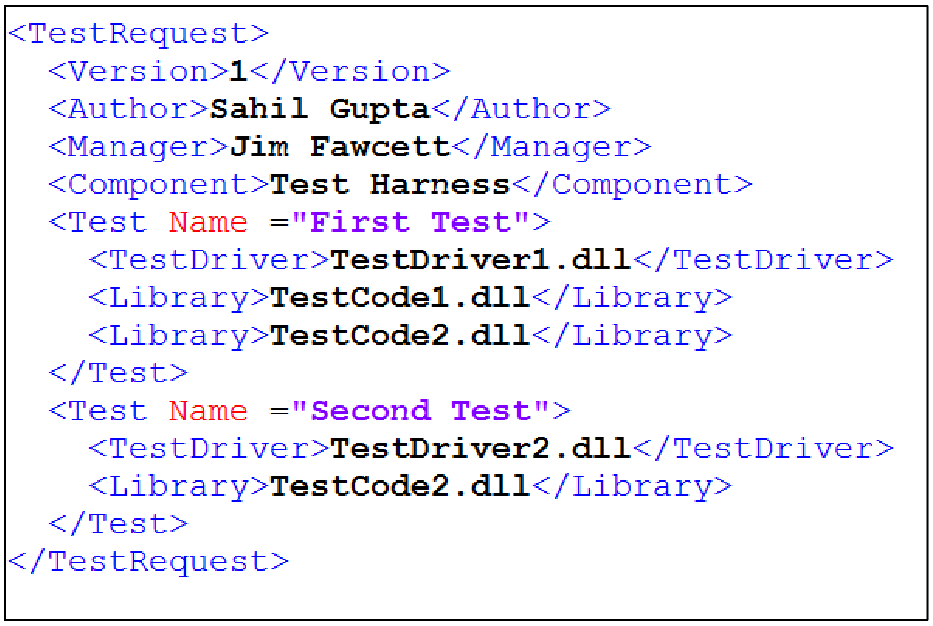
This module is responsible for displaying results of each test request to its users. For developers, it logs the date and time of execution, individual test case and results of the test case. The test cases are grouped by TestDrivers they run under. In case of errors, it also logs the exception details which can help developers fix their code.

For managers, it provides a report of its team members progress by number of tests executed and its status. They can also see tests executed in a component and status of tests in a particular time frame.

4.12. Test Request

A test request is an XML file that comprises all information that the test harness will require to execute a test. They are created by developers and QA's according to their requirements.

A sample test request contains the authors name and assemblies which contain test driver and test code. Optionally the developer can specify Managers name and component to allow better querying.



```xml
<TestRequest>
  <Version>1</Version>
  <Author>Sahil Gupta</Author>
  <Manager>Jim Fawcett</Manager>
  <Component>Test Harness</Component>
  <Test Name ="First Test">
    <TestDriver>TestDriver1.dll</TestDriver>
    <Library>TestCode1.dll</Library>
    <Library>TestCode2.dll</Library>
  </Test>
  <Test Name ="Second Test">
    <TestDriver>TestDriver2.dll</TestDriver>
    <Library>TestCode2.dll</Library>
  </Test>
</TestRequest>
```

Figure 3 - Sample TestRequest

4.13. TestDriver

A test driver is a program which is written by the developer of an application that wants to test his application using the test harness. Test drivers contain stubs to test individual or overall functionality of the application. The developer generates a library of the test driver and specifies the location in test request XML file.

The test driver should derive from an ITest interface and implement two methods, test() and GetLog(). The test method contains stubs for developers code to be tested and GetLog() method is a mechanism to obtain results of the test from test drivers.

5. Activity

5.1. Lifecycle of Test Request

The following activity diagram demonstrates how a test request will be processed in the Test Harness.

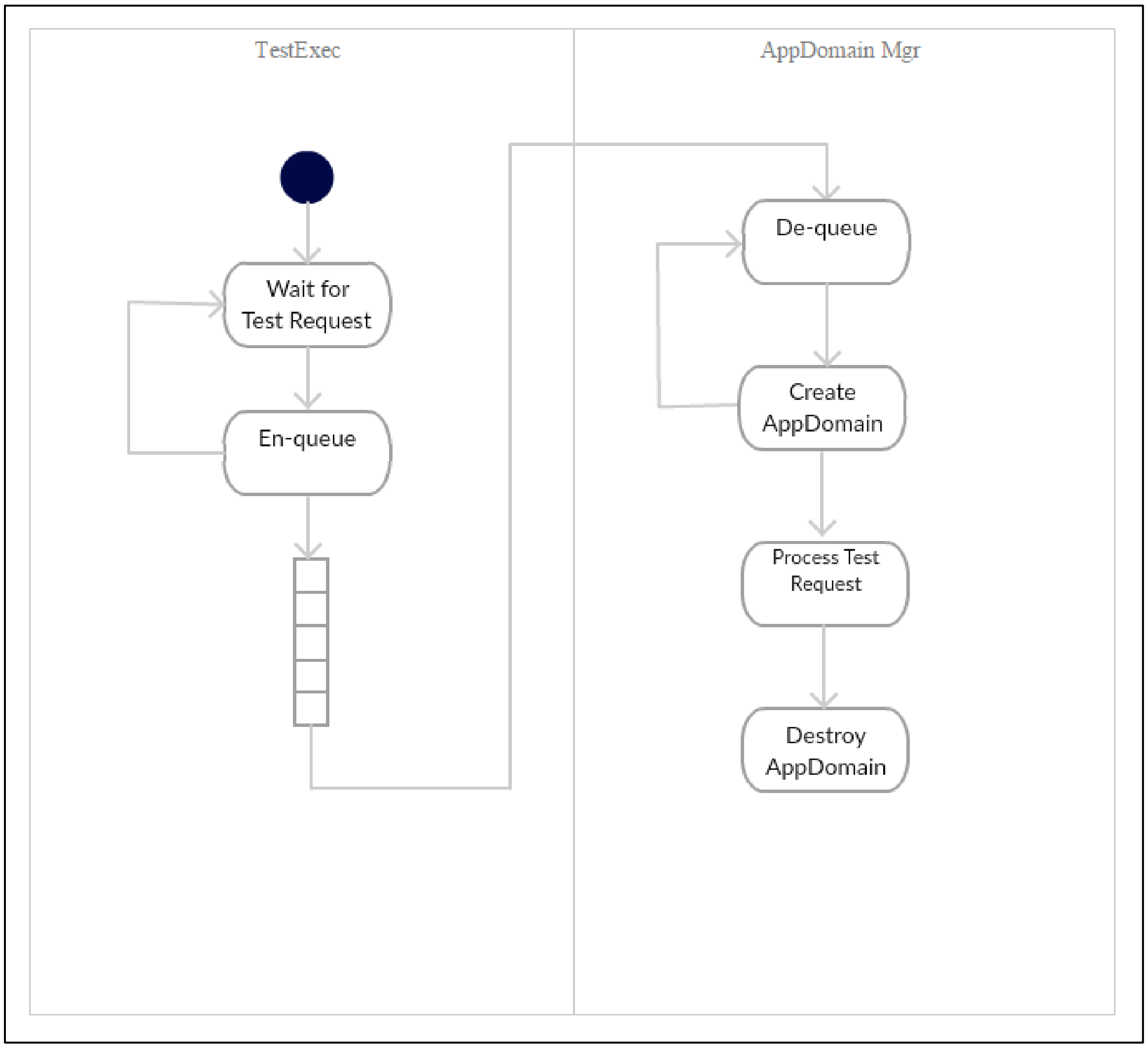


Figure 4 - Activity Diagram showing Lifecycle of Test Request

1. The TestExec waits for a test request in the form of an XML file from a user. The user may use command line interface to pass the test request or place the test request in a pre-determined directory from where the TestExec will read the test request.
2. The TestExec will en-queue the request and then continue listening for more requests to come.
3. The AppDomain Manager will wait on the queue to check if there are any requests to process. When it finds a test request, it de-queues it and creates an AppDomain to continue processing the request. It then goes back to waiting on queue for more requests.
4. The AppDomain then processes the test request by parsing it and then executing the test driver it contains.
5. After the test request is completed, the AppDomain is destroyed.

5.2. Processing Test Request

Below activity diagram shows how a test request will be processed.

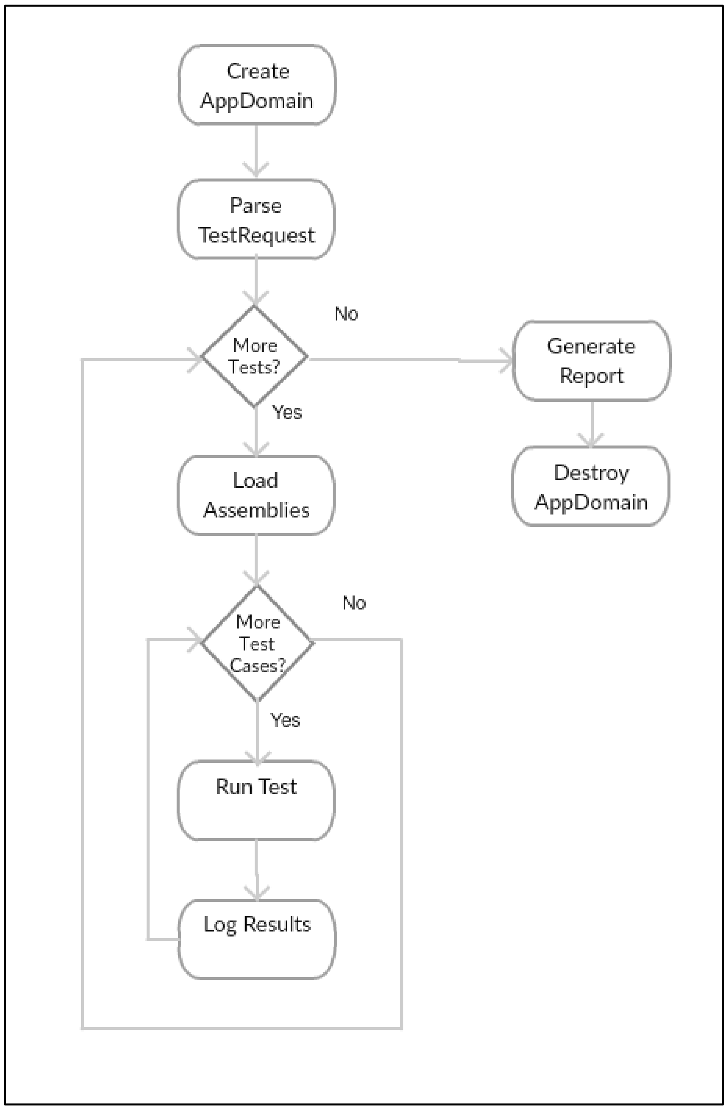


Figure 5 - Activity Diagram showing Execution of Test Request

1. The AppDomain Manager creates a new AppDomain to process each test request.
2. The AppDomain uses the XmlParser to parse the xml file and extract test information from it.
3. While there are tests to process, the AppDomain reads one test at a time and processes it serially.
4. The AppDomain uses the Loader to load assemblies, i.e. the TestDriver and Library.
5. It then runs the TestDriver and logs appropriate results into the database.
6. The AppDomain then Generates a Report based on all tests in the test request.

7. Finally, the AppDomain Manager destroys the AppDomain when the processing is completed.

6. Critical Issues

The test harness is a tool that will be in constant use throughout the lifetime of a project. As such, it is essential that foresee all critical issues that may arise during its execution. Some of the critical issues and their potential solutions are mentioned below:

6.1. Ease of Use

The test harness is a software that will be used by almost every member of a project. The same interface will be used by developers, QA's as well as managers. Developers and QA's will heavily use the system throughout the lifetime of the project. As such it is critical that the system be designed in such a way that it is easy to use for all. The main aim of creating a test harness is that developers and QA's should be able to automate testing and not spend much time on creating the setup for test cases. So it is essential that the user interface be smooth and present all the features to its users without having any complicated setup and operational requirements.

Solution:

We have a well-defined test request format in the form of an XML file that the developers and QA's will use to provide test information. While starting the program, we will accept an absolute directory path which will act as the working directory for test harness. Under this directory we will create directories "TestRequest", "ProcessedRequests", "TestReports" and "Logs". Users of the system can place their XML files under the "TestRequests" directory, from which the test harness will read and process the test request. Reports containing the results of the test will be generated under the "TestReports\Manager_Author" directory where a the "Manager_Author" directory will be generated using the details provided in the XML. This will allow the developers as well as managers to easily look for records of a particular test case. After processing the request, the XML file will be moved to the "ProcessedRequests" directory. Thus, interacting with the system is just a matter of placing test request in their defined location for the users.

On implementation of Project 4 we will have a client-server model, where users can use command line to pass test requests to the test harness and obtain results upon completion. We may also have a GUI which will allow users to send multiple test requests at once. The GUI can also provide capability to query past test results. For Managers, we can implement a GUI showing progress of his team members.

6.2. Performance

The performance of the test harness is the most critical issue to be considered while implementing the system. The following things are to be considered while implementing which define boundary conditions of the system.

1. What are the maximum number of test requests that the queue can hold?
2. If we chose to implement multi-threaded framework for processing TestRequests, what are the maximum test requests that the system can process in parallel?
3. Will there be a performance impact on the system during product release as the system will get continuous test requests from both developers and QA teams.
4. What will be the minimum system requirements for the test harness to support a desired number of users?

Solution:

We need to limit the number test requests the system can handle, after stress testing it, to a configurable value. The limit will depend on system configuration and so we may define it under requirements for using the system. We can publish guidelines for better use of system, which should include results of stress test on the system. This will help users set limits on using the test harness.

6.3. Security

Only authenticated users should be able to get access to the system. The test harness may allow querying logs of the system. We do not want unauthorized users accessing these logs or running their tests.

Solution:

We may choose to implement a login window where each user has to enter a unique ID and password. This way only registered users will get access to the test harness.

Alternatively, we have a user id field in the test request format. We may use this to validate the user against a stored database of registered users.

6.4. Reliability and Load Balancing

For an enterprise system, we might have several users sending test requests to the system. Having lots of users on one server may reduce the performance of the system. Additionally, the server may have downtime for maintenance. How do we ensure availability in that case?

Solution:

We may want to run multiple instances of the test harness on multiple servers. In this case a team of developers can use one test harness for development and a QA team may use another instance of test harness on another system.

Having multiple instances also helps in case we have a downtime on one of the servers running the test harness. The users can direct their test cases to alternate instances of test harness in case one server is down.

6.5. Language Interoperability

The test harness is designed in C# using .NET framework. How will it support execution of programs written in different languages such as java?

Solution:

We may choose to implement plugins which let use third party libraries like IKVM[1] or jni4net[2] to run java applications on C#. For python, we can use the IronPython[3] library.

We can find similar libraries for other programming languages and integrate them with the test harness.

6.6. Versioning and Backward Compatibility

We may have enhancements in the test harness and release future versions of test harness with additional functionality. In some cases, the test request structure defined by the XML or database format may change. We need to ensure that we do not break the functionality of existing system when we release future versions. This is particularly critical when we have scheduled test drivers to run at fixed time intervals.

Solution:

We have added a version field under the XML file format and the database key-value pairs which will tell us which the version of the test harness that particular test request belongs to. Each time we process the test request, we first check which version it belongs to and then parse the request accordingly.

6.7. Demonstration

We need to demonstrate that we need all the requirements stated by the professor. This needs to be done without any user interaction. The TA's should be able to run the system and validate if we meet all the requirements without interacting with the system.

Solution:

We will provide a batch file which will run the system, providing all necessary inputs. We will display relevant messages on the console after each requirement is met. This will help TA's pinpoint the location in code where we are implementing individual requirements, as well as look through output on console.

7. References

1. https://www.ikvm.net/
2. http://jni4net.com/
3. http://ironpython.net/
4. https://msdn.microsoft.com
5. https://www.wikipedia.org/

8. Appendix

8.1. Prototype - File Manager

The file manager prototype is used to parse files in a directory and display the names of the files in that directory on console.

8.1.1. Design

The prototype is implemented as an executable which accepts the absolute location of directory as command line argument. It then searches the directory and displays names of all library files (.dll) on the console, with each file name on a new line. If the user does not provide an input, the executable prints a help message on console describing the input syntax.

The File manager used a function '*GetFiles*' to search for files of a particular extension type in a directory and displays the file name under that directory. It accepts two arguments, the directory to search under and the file extension. Having the file extension argument allows us to use this function to list any type of file in the directory.

If we decide to read test requests from a fixed directory for the Test Harness, we can use the same prototype to search for XML files in the directory.

8.1.2. Conclusion

By building this prototype, we learnt how to programmatically enumerate files in a directory. Using this prototype, we can demonstrate how the test harness will enumerate files from a directory to search for a particular library or xml file.

8.2. Prototype - Child AppDomain Demo

The child AppDomain prototype is developed to demonstrate the use of AppDomains. For this demo, we will create an AppDoman and load and execute a test library in it. We have created a TestLibrary which contains two functions, "*display*" and "*add*". The display function prints a message on console and the add function returns sum of two integers.

8.2.1. Design

The prototype uses a relative path to locate the TestLibrary assembly and optionally accepts the TestLibrary path as command line input. It then creates an AppDomain and loads and executes the TestLibrary. The TestLibrary functions display and sum are invoked and appropriate logs are printed on console.

8.2.1.1. ExecuteLibrary

The AppDomainDemo prototype defines a function "*ExecuteLibrary*" which is used to test multiple libraries in one app domain. In this function we create a new appdomain and then create an instance of a "*LoadLibrary*" class. We then call the appropriate member function of LoadLibrary class to loads and execute the library. We pass the location of library as an argument to the member function.

In this prototype, we define a LoadTestLibrary function that loads and executes TestLibrary.dll assembly. For the test harness, the ExecuteLibrary acts as the AppDomin under which we process individual test request. The test request may have multiple TestDrivers and code assemblies that needs to be executed under one AppDomain.

8.2.1.2. LoadLibrary

The LoadLibrary class is a generic class that has methods to load and execute different libraries. Each member function first loads the assembly file of the library and then invokes member functions of the library.

The LoadLibrary inherits MarshalByRefObject which is required to pass parameters in the AppDomain. In this case, we are passing the path of the DLL file to the appDomain which then loads the file.

8.2.2. Conclusion

By building this prototype we learnt the use of application domains. We learnt how to load libraries in the appdomain and how to invoke functions from the library. By implementing this prototype, we demonstrated how the test harness will create AppDomain and execute a test request under the appdomain.