BUILD SERVER

CSE-681, Project 1

Rahul Kadam

9/13/2017

Table of Contents

List of Figures

1. Executive Summary

For big software systems, software developed consists of hundreds or even thousands of packages and perhaps several million of lines of code which has been developed by hundreds of developers over the years. Every developer is responsible for a set of packages which are developed, build, tested and then integrated in current software baseline. As new parts are added to the baseline and as developer make changes to fix latent errors or performance problems developer will have to re-run test sequences for those parts and, perhaps, for the entire baseline. Because there are so many packages the only way to make this intensive testing practical is to automate the process.

The process, described above, supports continuous integration. That is, when new code is created for a system, we build and test it in the context of other code which it calls, and which call it. As soon as all the tests pass, we check in the code and it becomes part of the current baseline. Continuous integration works on the principle of **"Keep it working Principle" (KIPW)**.

There are several services which are necessary to efficiently support continuous integration, collectively called, a **"Federation of Servers"**, each providing a dedicated service for continuous integration. This Federation mainly consists of: **Repository Server, Build Server, Test Harness Server, Client and Project Server**, as given in Fig. 1 below.

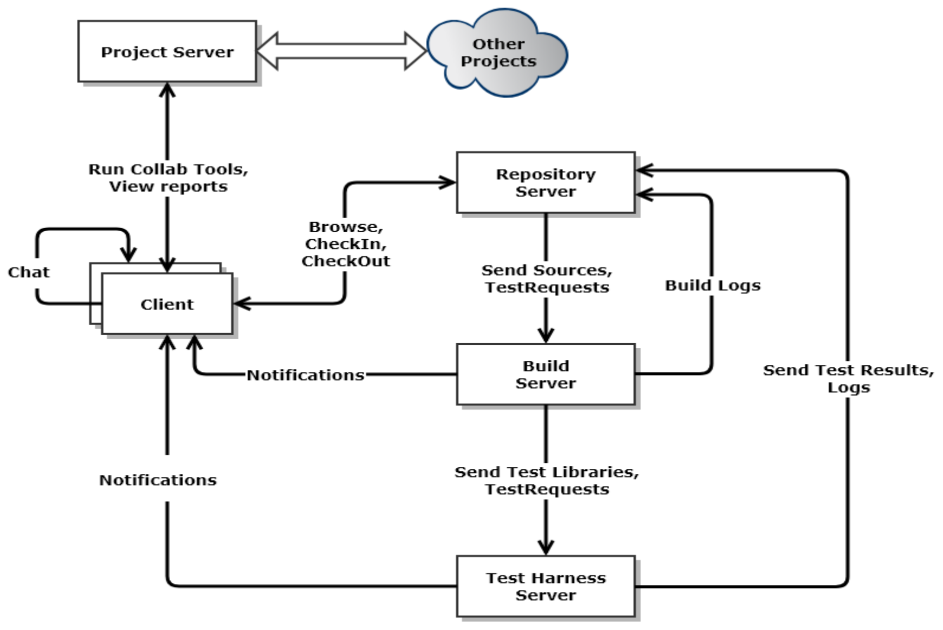


Fig 1. Federation of Servers.

The focus of this Operational Concept Document is on “**Build Server”**. Build Server is an automated tool that builds test libraries based on Test Requests and code sent from the Repository, for submission to the Test Harness. On completion, if successful, the build server submits test libraries and test requests to the Test Harness, and sends build logs to the Repository.

The users of Build Server can be Developers, Instructors, Teaching Assistants, Graders, QAs, Managers. Developers can use this system to continually test their new as well existing code with the baseline through an automatic process. Instructors or Teaching Assistants or Graders can test the coding assignments by creating Test Requests and writing Test drivers(cases) to thoroughly test them. QAs and Managers too can test the current software baseline by a simple automated process. Metaphorically it can be thought of a user simply “pushing a button” and then receiving Build logs and Test Results automatically.

A few critical issues shall be considered while developing the system and their solutions would be provided.
Some of these are:

- Ease of Use - The application is easy to use for any user.
- Performance and Productivity - The application has robust performance and high productivity.
- Logging - The application’s logging facility should be descriptive and easy to understand.
- Accuracy - The applications should provide accurate test results
- Security - The application should be secure from any malicious activities.
- Exception Handling - The application should handle exceptions that could occur and have complete system breakdown.
- Incorrect Test Request Format - The application should provide a Test Request template to user and prompt user when such template is not followed.

2. Introduction

As discussed in previous section about “Federation of Servers” concept, we will be developing one of these federated servers, the Build Server - an automated tool that builds test libraries. Below is a High-Level Architecture of what is to be implemented.

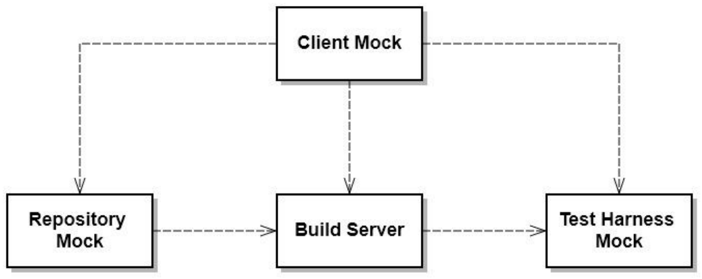


Fig 2. High Level Architecture of Entire System

The Focus of our system being on Build Server, we will be developing Mock Client, Mock Repository, Mock Test harness to support the Build Server. Mock here is used to indicate that these systems are not fully functioning-Federation of servers but are just enough for a working model of Build Server which can be developed in given timeframe.

The Overall structure of this system consists of:

- Repository Mock:
  Contains all code for current baseline along with their dependency relationships. It also holds Test Results for given Test Requests.
- Build Server:
  Based on build requests and code sent from the Repository, the Build Server builds test libraries for submission to the Test Harness. On completion, if successful, the build server submits test libraries and test requests to the Test Harness, and sends build logs to the Repository
- Test Harness Mock:
  Runs tests, concurrently for multiple users, based on test requests and libraries sent from the Build Server. The Test Harness executes tests, logs results, and submits results to the Repository. It also notifies the author of the tests of the results.
- Client Mock:
  The user's primary interface into the Federation, serves to submit code and test requests to the Repository. Later, it will be used view test results, stored in the repository.

Entire Process will be as follows:

Clients will check-in, to the Repository, code for testing, along with one or more test requests. The repository sends code and requests to the Build Server, where the code is built into libraries and the test requests and libraries are then sent to the Test Harness. Test Harness then loads the libraries in a separate container and execute tests. It then provides these Test Results to the Repository, which can be accessed by Client.

2.1 Application Obligations

Following are the functionalities that Build Server should perform:

- It should accept Test Request from the Repository Mock.
- Then, parse and process the Test Request to receive relevant information.
- Select a Toolchain to build based on information acquired earlier.
- Attempt to build the libraries based on selected Toolchain.
- If Build is Successful, it should provide these libraries to Test Harness Mock for further process.
- If Build is Failed, it should notify the Client Mock about the same.

2.2 Organizing 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 dlls and log the results and execution.

2.3 Key Architectural Ideas

The Client Mock is main actor that makes requests and accesses required information form the Repository. While Repository Mock, Build Server and Test Harness Mock are reactors that react on the request received by them. So, the Repository Mock reacts on requests from Client Mock and issues requests to Build Server, the Build Server then reacts to requests from Repository Mock and issues requests to Test Harness Mock, finally Test Harness Mock reacts to requests from Build Server performs its assigned activities and notifies Client Mock and Repository Mock after its completion.

3. Uses

3.1 Developers

Developers are the primary users of test harness. Robust 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.

Design Impact

Ease of use is the major design impact for this application because if the User Interface is bad the developer may want to avoid using the application and it would remain as an unused resource. We therefore will provide a GUI (Graphical User Interface) in the end. The client will interact with the build server through the console in first version and then in the final version client will be provided with a GUI and the remote access facilities which would be very useful to the developer.

3.2 Instructors/Teaching Assistants/Graders

The Instructors, Teaching Assistants and Graders can use this tool to examine the system and check if all the requirements are met or not. They will be provided with a test executive by the developers which will demonstrate all the requirements of the test harness.

Design Impact

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 Executive 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.

3.3 Quality Assurance (QA)

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 this application. The QA's may need to run tests on certain part of baseline code or maybe the entire baseline. In a huge code base with complex dependencies such a application is a must for the successful working of the project. The QA's may start with 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.

Design Impact

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.

3.4 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.

Design Impact

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

3.5 Extending Application

We will extend the system to handle multiple clients in coming versions. It would also support remote accessing from different clients. One or more client(s) will concurrently supply the application with Text Requests. One or more client(s) will concurrently extract Test Results and logs by enqueuing requests and waiting for applications replies.

A full-fledged test harness can be built from the prototypes designed in previous versions as we add GUI as well. It would work with the repository to provide a professional testing environment to facilitate a small working Federation of Servers.

4. Application Activities

4.1 High Level Activity Diagram

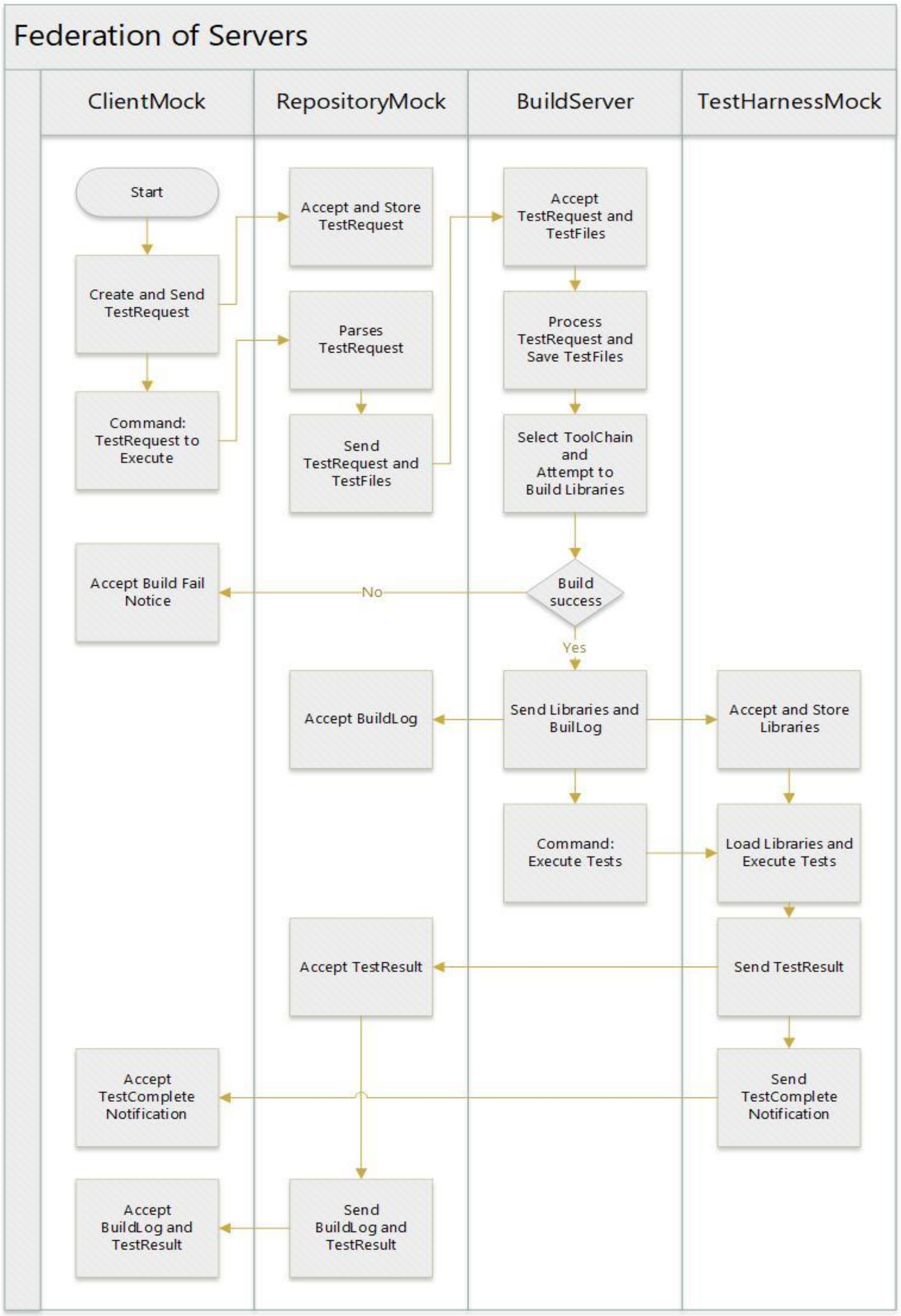

Federation of Servers
ClientMock
RepositoryMock
BuildServer
TestHarnessMock
Start
Create and Send TestRequest
Command: TestRequest to Execute
Accept and Store TestRequest
Parses TestRequest
Send TestRequest and TestFiles
Accept TestRequest and TestFiles
Process TestRequest and Save TestFiles
Select ToolChain and Attempt to Build Libraries
Build success
No
Yes
Accept Build Fail Notice
Accept BuildLog
Send Libraries and BuilLog
Accept and Store Libraries
Command: Execute Tests
Load Libraries and Execute Tests
Accept TestResult
Send TestResult
Accept TestComplete Notification
Send TestComplete Notification
Accept BuildLog and TestResult
Send BuildLog and TestResult

Fig 3. High Level Activity Diagram of Entire System

The above activity diagram describes the flow of activities which will occur in the application that we will build.

Following are the steps of action for the entire system:

- Client Mock creates a Test Request which is an xml file which consists of several Test drivers which consists of several Test files.
- Client then sends this Test Request to Repository Mock. Repository Mock accepts and stores this Test Request.
- Client can on command ask the Repository Mock to execute one or several Test Requests.
- After Receiving the command to Process Test Request, Repository Mock parses through the xml file gathers appropriate information needed and passes the Test Requests and Test Files to Build Server.
- After accepting the Test Request and Test Files, Build Server Processes the Test Request and saves Test Files for build process.
- After processing the Test Requests, the Build Server selects the appropriate toolchain required for Building the libraries. (e.g.: c++ toolchain, c# toolchain or java toolchain etc.)
- After selecting the appropriate toolchain, the build server attempts to build libraries.
- If Build Fails, a Build fail notification is send to Client Mock.
- If Build succeeds, Build Server Sends Build libraries to Test Harness Mock and Build Logs to Repository Mock.
- Test Harness Mock accepts and stores the libraries.
- On receiving command to execute tests, Test Harness will load the libraries and execute tests.
- After completion of tests, Test Harness Mock will send the Test Results to Repository Mock. It also sends Tests Complete Notification to Client Mock.
- After receiving Tests complete notification, Client Mock can command the Repository Mock top provide the Build Log and Test Results.

4.2 Build Server Activity Diagram

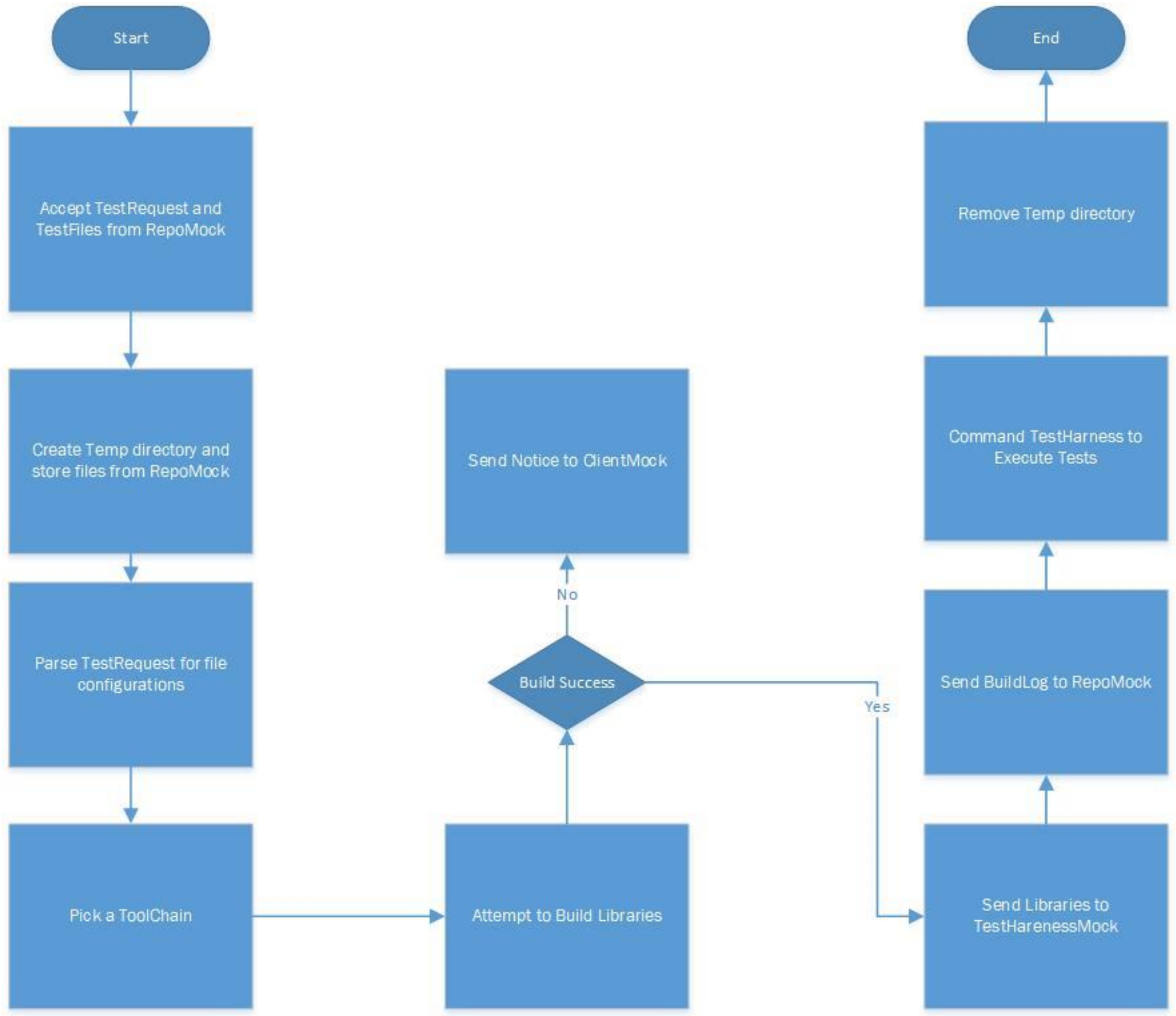

Start
Accept TestRequest and TestFiles from RepoMock
Create Temp directory and store files from RepoMock
Parse TestRequest for file configurations
Pick a ToolChain
Attempt to Build Libraries
Build Success
No
Send Notice to ClientMock
Yes
Send Libraries to TestHarenessMock
Send BuildLog to RepoMock
Command TestHarness to Execute Tests
Remove Temp directory
End

Fig 4. Build Server Activity Diagram

The above activity diagram describes the flow of activities which will occur in the Build Server that we will build.

Following are the steps of action for this system:

- The Build Server accepts the Test Request and corresponding Test Files from Repository Mock.
- It parses the Test Request to get certain required configurations such as author name, test files names, test driver name, date and time stamp of Test Request etc.
- Based on above collected information it creates a unique Directory for this Test Request and stores the Test Files in this Directory for further process.
- During the parsing process, it also picks up information which helps in selecting the appropriate toolchain to be used. (e.g.: c++ toolchain, c# toolchain or java toolchain etc.)
- After picking a toolchain it attempts to build libraries, libraries here are dlls (dynamic link libraries), each Test Request has a separate dll, it can also cache this dll or share dlls among multiple Test Requests.
- If Build process fails, a Build Fail notification is send to Client Mock.
- If Build process succeeds, it sends these dlls to Test Harness Mock to execute tests.
- It then removes the directory.

5. Partitions

5.1 High Level Package Diagram

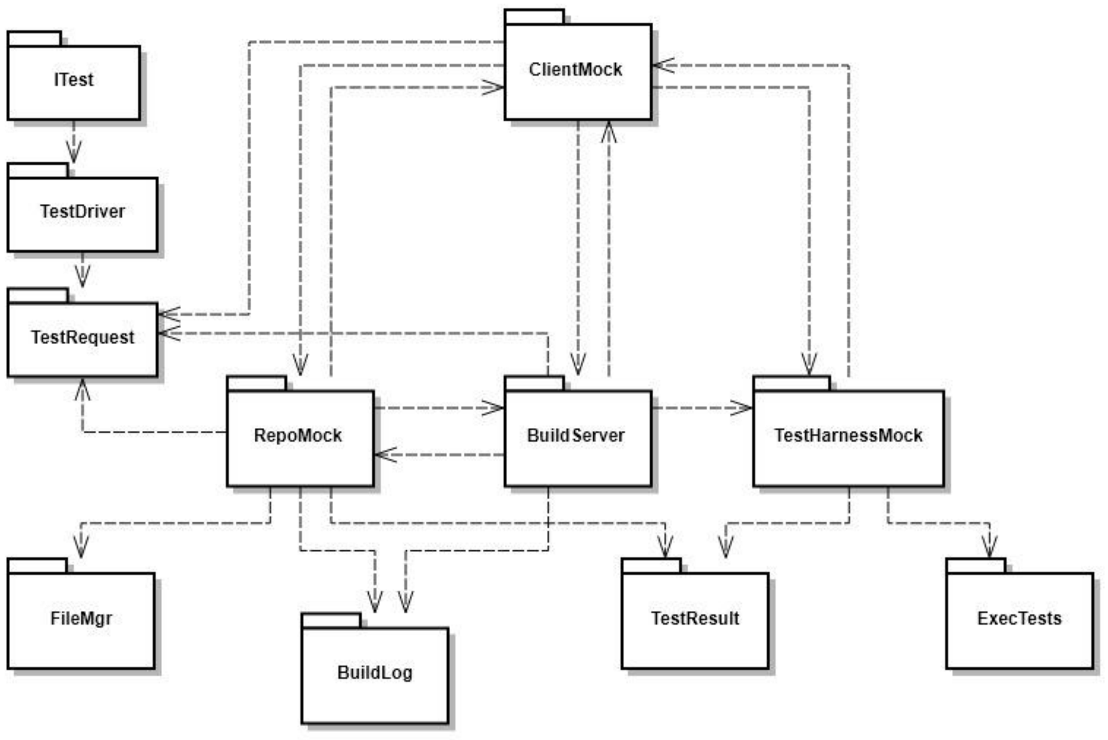


Fig 5. High Level Package Diagram

The Packages involved are as follows:

Client Mock:

- Client Mock as the Test Executive, is the main entry point of the application. It will make use of Test Request package to create Test Request (xml file) which may contain user related information such as author name, designation, timestamp and files to be tested.
- After creating the Test Request (xml file), Client Mock will send this file to Repo Mock.
- It will also accept Build Logs from Build Server and Test Results from Test Harness Mock.

ITest:

- The ITest package consists of ITest Interface which is implemented by any Test Driver Class.

- To create a Test Driver, we need to implement this ITest Interface.

TestDriver:

- The Test Driver package will be used to create Test Drivers by Client Mock which may call various other packages to be tested.

TestRequest:

- Test Request package is used to create xml files which configure the Test Driver and other Test files contained in it.

RepoMock:

- Repo Mock receives the Test Request and parses through it to get relevant necessary information.
- It also uses FileMgr package to manage Test Files and send them across Build Server.

FileMgr:

- FileMgr package consists of File/Directory operations which may be needed by Repo Mock to manage and transfer files to Build Server.

BuildServer:

- Build Server will be implemented with its own list of packages discussed in next section.
- But overall, it will Receive Test Request from Repo Mock.
- It will use BuildLog package to send build logs to Repo Mock.
- It will build test libraries and send these libraries to Test Harness Mock.
- It will also notify Client Mock if build fails.

BuildLog:

- Build Server will use BuildLog package to create build logs to be sent to RepoMock.

TestHarnessMock:

- TestHarnessMock receives the dlls from Build Server and on command loads and executes dlls using ExecTests package.
- It uses TestResult package to create tests results and send it to Repository Mock.

- It will also notify the Client Mock about Test Completion status.

TestResult:

- TestHarnessMock will use TestResult package to create Test Results which will be shared with Repo Mock.

ExecTests:

- Exec Tests will be the main package to load and exec the dlls by the TestHarnessMock.

5.2 Build Server Package Diagram

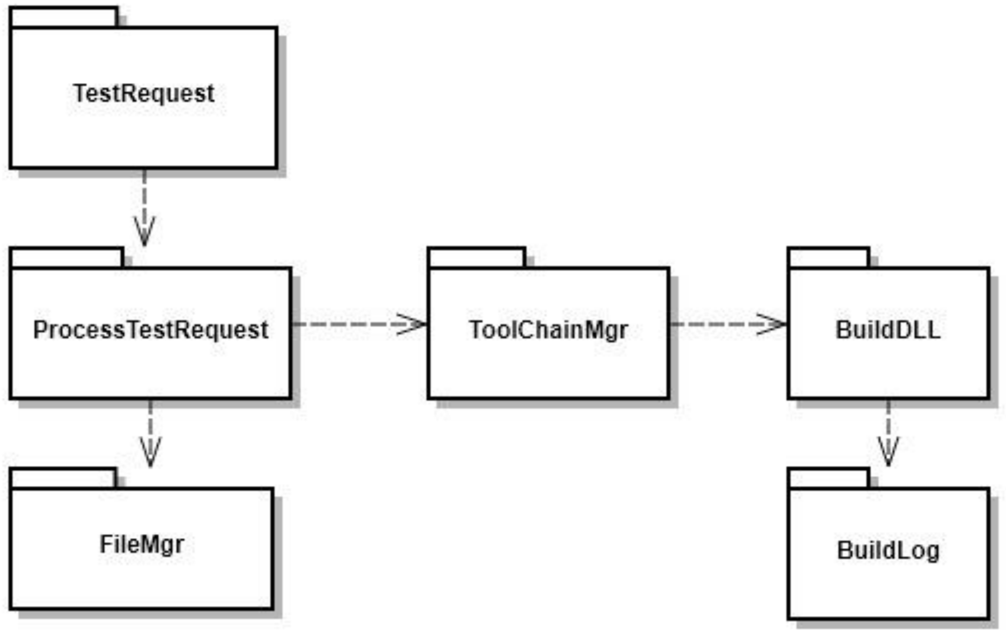


Fig 6. Build Server Package Diagram

TestRequest:

- Test Request package is used to create xml files which configure the Test Driver and other Test files contained in it. Repo Mock will provide Build Server with this Test Request.

ProcessTestRequest:

- ProcessTestRequest package will receive the Test Request file from Build Server.

- It parses/processes the Test Result xml file to attain relevant configurations.
- It also uses FileMgr package to manage Test Files received from Repo Mock and any other File/Directory related operations.
- Based on information from parsing the xml file it selects and appropriate Toolchain to build using ToolChainMgr package.

ToolChainMgr:

- ToolChainMgr package is used to manage and select submitted Test files to be build using that specific Toolchain.
- As an example, a C++ file will be build using g++, a C# file will be build using csc and Java file using javac.

BuildDll:

- BuildDll package is used to generate dlls of the given test files from test requests.

All the package defined above are following the “Single Responsibility rule” (SRP). They are focused on a single activity.

6. Critical Issues

6.1 Ease of Use

Ease of use is one of the major concerns for any developer. We should take care that we do not build a very rigid application, which only works for certain cases.

An overly complicated user interfaces discourages the largescale use of the systems. Hence providing 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 next version 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.

6.2 Performance and Productivity

The application 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 is therefore a critical issue. Apart from this the application should also support multiple clients.

The performance of the application can also refer to the complexity of the algorithms used to design the system. We can measure this by the time complexity and the space complexity of the system. The productivity of the system depends on the usage and the comparison of the time taken to perform the test requests through the test harness and manually.

Solution

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

To achieve low time complexity, we must make sure that the interaction between various packages occurs in an efficient manner without any deadlock conditions. Also, we can monitor the number of nested loops. The space complexity can also be reduced by reducing the number of data structures such as arrays, lists and generics such as dictionaries, hashtables etc. used for the development of the system.

6.3 Logging

Logging is a critical issue for the application since we need a efficient mechanism to send logs to the Repository and back to the client Mock to display logs to the user.

Solution

An interesting way of logging messages that will be adopted in this project is by recording the results of the test cases in a dictionary and print the values to the user after every test request is executed After all the test requests are executed we can retrieve them by simply calling a method and create a log file by adding the details. We can then insert the log file in a database.

6.4 Accuracy

The accuracy of the system is a significant issue since the test harness should provide the correct results of the code that is being tested. If the test harness gives the result as “Pass” then the code should be correct and function as per the requirement while if the test harness claims failure for a certain test case, then there should actually be some defect or bug in the code.

Solution

Accuracy can be ensured by adopting the following measures:

1. Correctly parsing the XML test request document and decoding all the library names accurately.
2. Proper maintenance of lists of libraries. Clearing the list every time the test runs.
3. Retrieving the correct paths for the dynamic link libraries.
4. Loading the test request libraries properly and recording the test results correctly for each test case.

6.5 Demonstration

Demonstration of the application is an important factor when we consider the flexibility of the system. The complete set of requirements need to be demonstrated so that the user understands the working of the system in an adequate manner.

Solution

The solution to demonstrating requirements can be fulfilled through the test executive package which demonstrates the working of the application by testing its requirements through the following test requests as a developer type of user:

1. Test Requests that generates test result “Pass”
2. Test code that generates test result “Fail”

6.6 Security

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.

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.

6.7 Exception Handling

There might be some unhandled exceptions in the test code, which would be encountered while running the test driver or during build prrocess. This might take down the entire application.

Solution

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

6.8 Incorrect Test Request Format

If the Test Request File generated is incorrect format, 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 template which is specific to Test Request format. If template is not followed we should inform the problem to user and ask them to correct it. In Appendix, we provide a Sample template for Test Request File.

7. Conclusion

The Federation of Servers is very important tool to have in a software development environment. Build Server as a part of such Federation of servers has been explained in this document both High Architecture wise and Low-level details about Build Server implementation and its activities.

To conclude, we can see that a Build Server is an extremely useful and important tool in the development of large software. But while designing it we should make sure that we address all the critical issues and make the application robust and secured.

8. References

1. http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project1-F2017.htm
2. http://deviq.com/build-server/
3. https://stackoverflow.com/questions/15637092/project-collection-build-error-logger

9. Appendix

9.1 Sample Test Request

Below is a sample Test Request in the form of xml file which will be created by Client Mock and send to Repository Mock for processing.

```xml
<?xml version="1.0" encoding="utf-8"?>
<testRequest>
  <author>Rahul Kadam</author>
  <dateTime>9/9/2017 8:13:34 PM</dateTime>
  <test>
    <testDriver>td1.cs</testDriver>
    <tested>tf1.cs</tested>
    <tested>tf2.cs</tested>
    <tested>tf3.cs</tested>
  </test>
</testRequest>
```

Fig 7. Sample Test Request File

9.2 Prototype for Build Server

The Build Server Prototype takes as input:

1. Path to a ".csproj" file for a Visual Studio Project to be build
2. Path to a text file to log build results.

The ".csproj" is itself an xml file which is parsed to get relevant C# package information and build configurations.

Below is the "Build Successful" output of the Prototype:

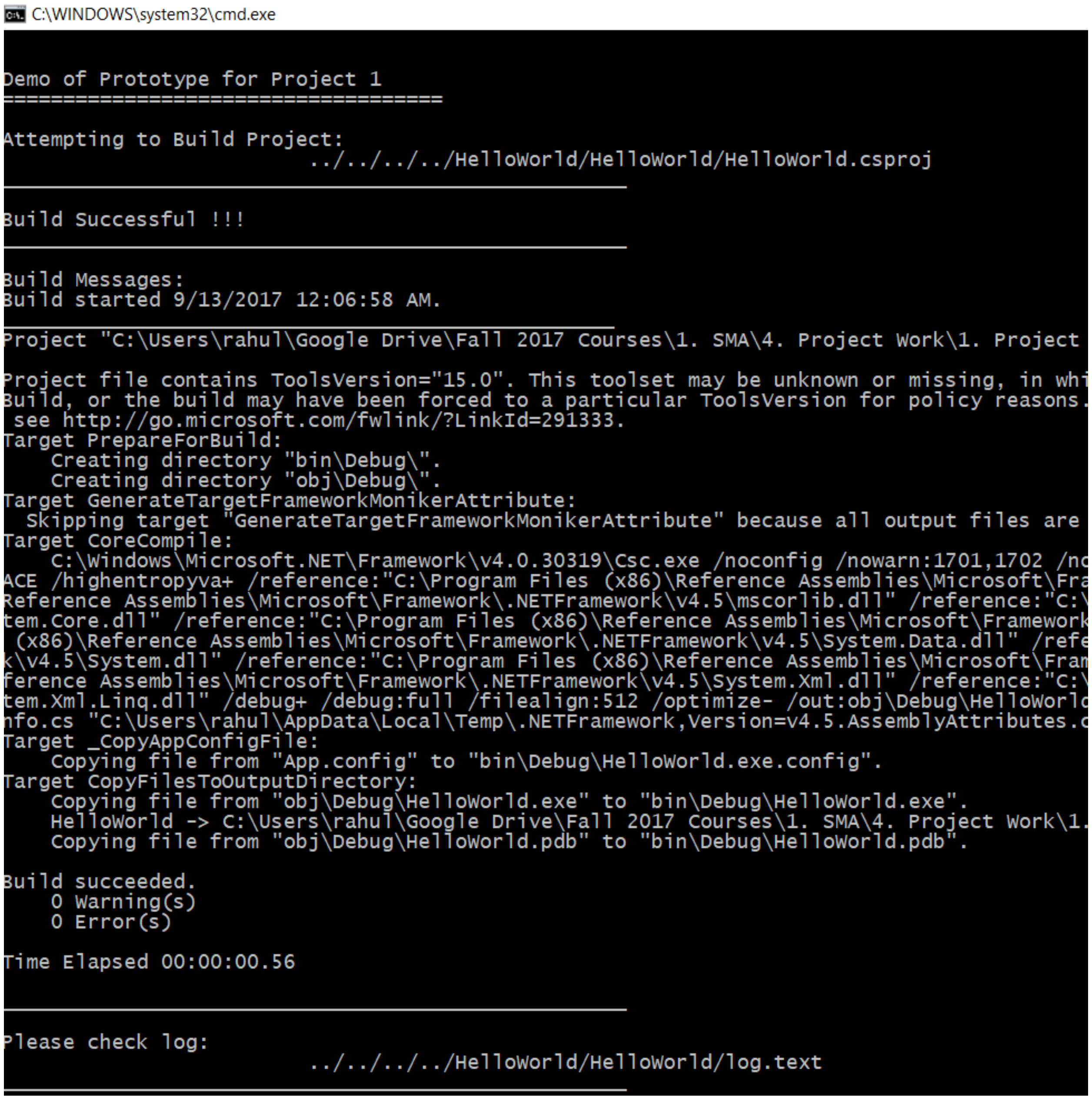
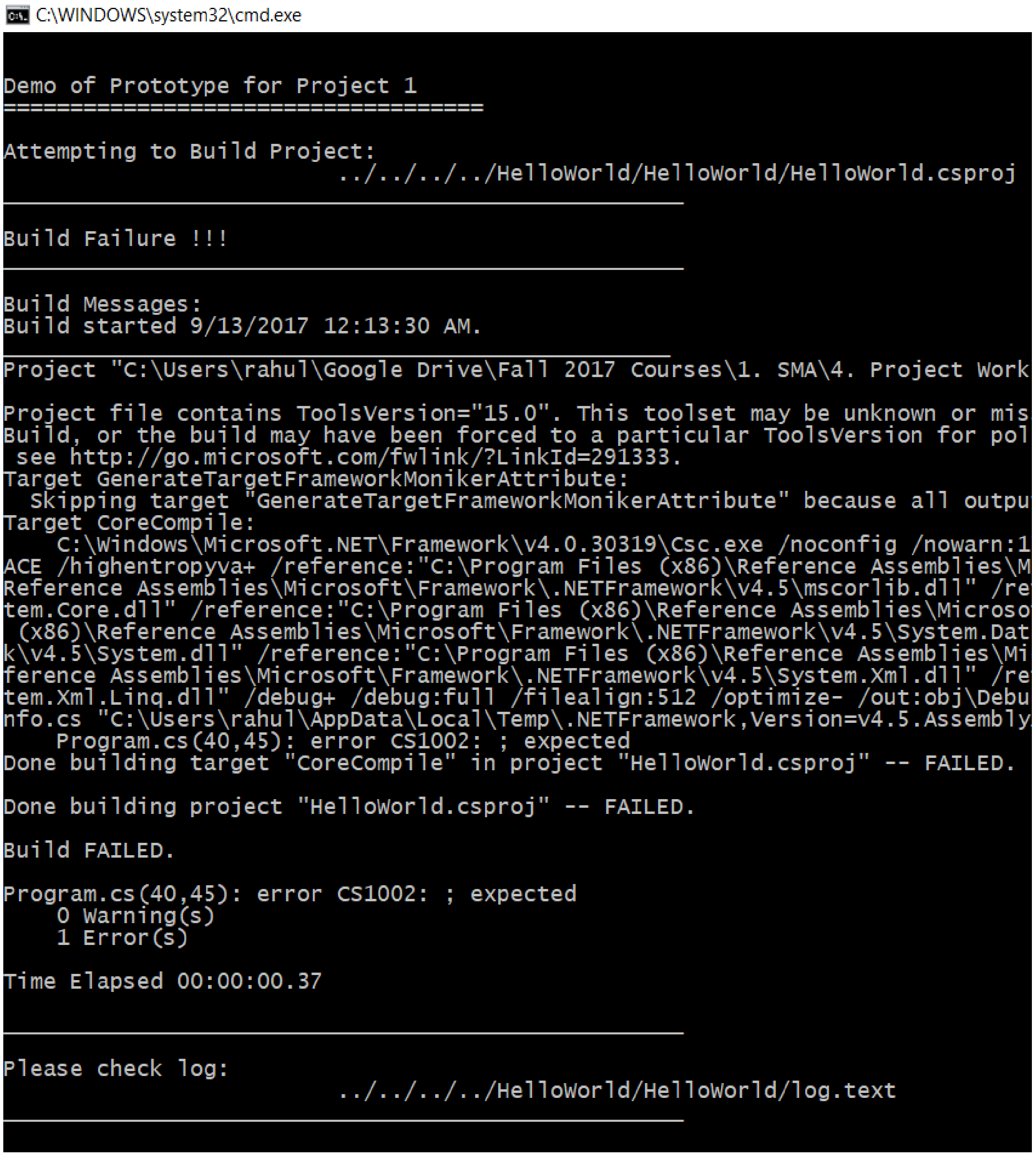

```
C:\WINDOWS\system32\cmd.exe

Demo of Prototype for Project 1
====================================

Attempting to Build Project:
                         ../../../../HelloWorld/HelloWorld/HelloWorld.csproj
____________________________________________________

Build Successful !!!
____________________________________________________

Build Messages:
Build started 9/13/2017 12:06:58 AM.
____________________________________________________
Project "C:\Users\rahul\Google Drive\Fall 2017 Courses\1. SMA\4. Project Work\1. Project

Project file contains ToolsVersion="15.0". This toolset may be unknown or missing, in whi
Build, or the build may have been forced to a particular ToolsVersion for policy reasons.
 see http://go.microsoft.com/fwlink/?LinkId=291333.
Target PrepareForBuild:
    Creating directory "bin\Debug\".
    Creating directory "obj\Debug\".
Target GenerateTargetFrameworkMonikerAttribute:
  Skipping target "GenerateTargetFrameworkMonikerAttribute" because all output files are
Target CoreCompile:
    C:\Windows\Microsoft.NET\Framework\v4.0.30319\Csc.exe /noconfig /nowarn:1701,1702 /nc
ACE /highentropyva+ /reference:"C:\Program Files (x86)\Reference Assemblies\Microsoft\Fra
Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\mscorlib.dll" /reference:"C:\
tem.Core.dll" /reference:"C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework
 (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\System.Data.dll" /refe
k\v4.5\System.dll" /reference:"C:\Program Files (x86)\Reference Assemblies\Microsoft\Fram
ference Assemblies\Microsoft\Framework\.NETFramework\v4.5\System.Xml.dll" /reference:"C:\
tem.Xml.Linq.dll" /debug+ /debug:full /filealign:512 /optimize- /out:obj\Debug\HelloWorlc
nfo.cs "C:\Users\rahul\AppData\Local\Temp\.NETFramework,Version=v4.5.AssemblyAttributes.c
Target _CopyAppConfigFile:
    Copying file from "App.config" to "bin\Debug\HelloWorld.exe.config".
Target CopyFilesToOutputDirectory:
    Copying file from "obj\Debug\HelloWorld.exe" to "bin\Debug\HelloWorld.exe".
    HelloWorld -> C:\Users\rahul\Google Drive\Fall 2017 Courses\1. SMA\4. Project Work\1.
    Copying file from "obj\Debug\HelloWorld.pdb" to "bin\Debug\HelloWorld.pdb".

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.56

____________________________________________________

Please check log:
                         ../../../../HelloWorld/HelloWorld/log.text
____________________________________________________
```

Fig 8. Output of Prototype of Build Server (Build Successful)

As shown in above Snapshot, about build successful and 0 Warnings and 0 Errors is reported.

Below is the “Build Fail” output of the Prototype:



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

Demo of Prototype for Project 1
====================================

Attempting to Build Project:
                         ../../../../HelloWorld/HelloWorld/HelloWorld.csproj

___________________________________________________

Build Failure !!!
___________________________________________________

Build Messages:
Build started 9/13/2017 12:13:30 AM.
__________________________________________________
Project "C:\Users\rahul\Google Drive\Fall 2017 Courses\1. SMA\4. Project Work

Project file contains ToolsVersion="15.0". This toolset may be unknown or mis
Build, or the build may have been forced to a particular ToolsVersion for pol
 see http://go.microsoft.com/fwlink/?LinkId=291333.
Target GenerateTargetFrameworkMonikerAttribute:
  Skipping target "GenerateTargetFrameworkMonikerAttribute" because all outpu
Target CoreCompile:
    C:\Windows\Microsoft.NET\Framework\v4.0.30319\Csc.exe /noconfig /nowarn:1
ACE /highentropyva+ /reference:"C:\Program Files (x86)\Reference Assemblies\M
Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\mscorlib.dll" /re
tem.Core.dll" /reference:"C:\Program Files (x86)\Reference Assemblies\Microso
 (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\System.Dat
k\v4.5\System.dll" /reference:"C:\Program Files (x86)\Reference Assemblies\Mi
ference Assemblies\Microsoft\Framework\.NETFramework\v4.5\System.Xml.dll" /re
tem.Xml.Linq.dll" /debug+ /debug:full /filealign:512 /optimize- /out:obj\Debu
nfo.cs "C:\Users\rahul\AppData\Local\Temp\.NETFramework,Version=v4.5.Assembly
    Program.cs(40,45): error CS1002: ; expected
Done building target "CoreCompile" in project "HelloWorld.csproj" -- FAILED.

Done building project "HelloWorld.csproj" -- FAILED.

Build FAILED.

Program.cs(40,45): error CS1002: ; expected
    0 Warning(s)
    1 Error(s)

Time Elapsed 00:00:00.37

___________________________________________________

Please check log:
                         ../../../../HelloWorld/HelloWorld/log.text
___________________________________________________
```

Fig 9. Output of Prototype of Build Server (Build Fail)

As shown in above Snapshot, about build Fail and 0 Warnings and 1 Errors is reported, with 1 Error of Missing Semicolon “;”.

Lessons Learnt:

1. To Handle Exceptions:
   An Exception would occur if user provides incorrect path of .csproj as input.
   Hence, the Application should be able to handle Exceptions.

2. Relative Paths as Input:
   For application to work on different systems, Relative path must be used as input.
   Hence, the Application should be able to handle Relative Paths given as input.