BUILD SERVER

Project #1 OCD

Instructor: Prof. Jim Fawcett

Course Name: CSE 681 Software Modeling & Analysis

Sonal Patil [SUID 997435672]

Email: spatil06@syr.edu

Date: 09-12-2017

TABLE OF CONTENTS

TABLE OF FIGURES

1. EXECUTIVE SUMMARY

This Operational Concept Document details the design concept of remote build server which is nothing but a centralized, stable and reliable environment for building distributed development projects. Few of the advantages of build server include:

- Acceleration of compile and link processing
- Elimination of redundant tasks
- Elimination of dependencies on key personnel
- Improvement of product quality
- Minimization of "bad builds"
- Having history of builds and releases in order to investigate issues
- Saving time and money - because of the reasons listed above

Following sections describe our simple build server which automates the tasks of building the code, creating test libraries for Test Harness and logging results of every such operation. Each time a new code is to be added or existing code is to be changed, build server is triggered by the user of the server. In response, the server will pull the new code from a code repository's respective branch (development, staging or production), build it from scratch and trigger the Test harness to run all the unit tests if the build is successful. The end result is the build of the most current code.

This increases predictability by enforcing source control and making it possible to flag issues and notify all the users (developers, managers, QA team etc.) of the server quickly if there are conflicts or missing dependencies.

The advantages of the build server extend beyond developers. With a build sever in place; QA teams always know which version they need to test. This allows them to streamline their own workflow by introducing automated testing tools. It helps ensure that the same dynamic link library is used for all builds and that out-of-sync check-ins don't lead to failure during quality assurance (QA) testing.

Managers can use such a server for their code sign off operations while release teams benefit from it while rolling in or out the production versions of the software.

Few critical issues can be identified for this build server:

1. Reduced efficiency and increased complexity of the system because of the complex long builds
2. Reduced accuracy of the system due to multiple false notifications to the client
3. Increased storage space to maintain long build logs and all the build versions
4. Very less support to handle reckless action from user side to send requests even after exceeding the input queue capacity and getting notifications from the system

Our build server is divided into following three separate modules which perform specific tasks:

- BQueuer : Queues the incoming build requests/jobs until BGenerator can accept a new one
- BGenerator: Builds the request & creates test libraries for Test Harness if the build is successful
- BReporter : Reports the results of each build operation attempt to the users

Section 2 of this document explains about the build server obligations and its architecture. Section 3 illustrates different users of the build server and their use cases while section 4 and 5 go into details of build server activities and design of the prototype. Section 6 discusses some of the critical issues related to my design and provides their potential solutions. Appendix section includes the sample prototype programs' outputs.

This server and its prototypes are both developed in C# using .Net framework class libraries and Visual Studio 2017.

2. INTRODUCTION

2.1. Background

Research spanning over the entire life of software industry has shown us harmful impacts of disconnects between important activities e.g., planning, development and implementation. The bigger the project more is the need to streamline and automate the important activities to mitigate risks in software development. Having highly efficient Build servers is one of the many such risk management strategies. Build Server is like heart beat monitor. It tells you how healthy your system is and it also notifies you in advance if your system is going to break down.

2.2. Application Obligations

The main responsibilities of this system are to build the change requests, create the build logs, trigger the Test Harness and notify users about the build results. Such a system needs to be scalable, efficient and secure with following obligations:

1. It should be machine independent
2. It should store the files sent from users (based on access level) to the repository
3. It should pull the required test files from the Repository on every build request
4. It should parse the test request (XML Files) into correct tool chain i.e. C#
5. It should attempt to build the test request, create the build logs, test libraries
6. It should save build logs to the repository and notify users of the build results
7. It should also trigger Test Harness on a successful build and send test libraries to it

2.3. Organizing Principles

The organizing principle of the system is to perform the functionality of the Build Server. It will rely on the mock repository and mock test harness to perform its functions.

2.4. Key Architectural idea

The key architectural idea is, building a GUI that'll interact with the user and will let them upload the files they want to build. These files will then get stored to the repository. When user gives the command to build, the request is stored in the BQueuer. When popped, the BGenerator will grab all the appropriate files from the repository and will try to build them and create build logs to indicate the results of build operation. Builds logs will be saved back to repository by BReporter. On a successful build, build server will trigger Test Harness to execute the tests using test libraries created by the server and produce the result. That result will then get notified to the users.

3. USERS & USES

Following types of users can interact with the Build Server:

3.1. Software Developers –

Developers' work is to develop a piece of software as per the customer requirement. When a developer makes changes in his or her code or introduces a new code, it shouldn't affect any other team member's work or shouldn't cause whole system to break down.

Uses – The build server can help developers to build & execute the test cases on their code and notify the result of build/test request to all team members and other teams working on same project. Using the build server thus helps maintain the source control and to pinpoint the cause of build or test.

3.2. Managers –

Manager is the person who works for both development and deploying (staging) stages. He has access to the development code which he'll then deploy for staging branches of the repositories for QAs.

Uses – Manager will get notified from the build server about the developer teams' builds and test results. This will help him keep track of the project's progress. Manager can run his own test requests on developers' code and get the build and test result from the build server. He can then decide whether to move the current clean code to the staging branch and do code sign-off for QAs.

3.3 Quality Assurance team –

Quality Assurance (QA) team works on project level testing. They need to check the developer team's code before sending it out for final sign off.

Uses – Build server can be useful for QA team to run their own high-level test cases on the staging branches' (signed off by Managers) code. If the bug is found, ticket filed against the developer will keep track the upcoming code fixes. QA can then take such fixed code and send requests to build server to see the validity of the fix. They will notify the developer team if any build fails or test fails. Build server also helps QAs keep track of all those builds and their results with the help of version ID provided in the log.

Once QA team gets the clean build and test results from build server and Test Harness, they can proceed with signing off the code for release team. Following activity diagram explains how the QA team is going to use the build server:

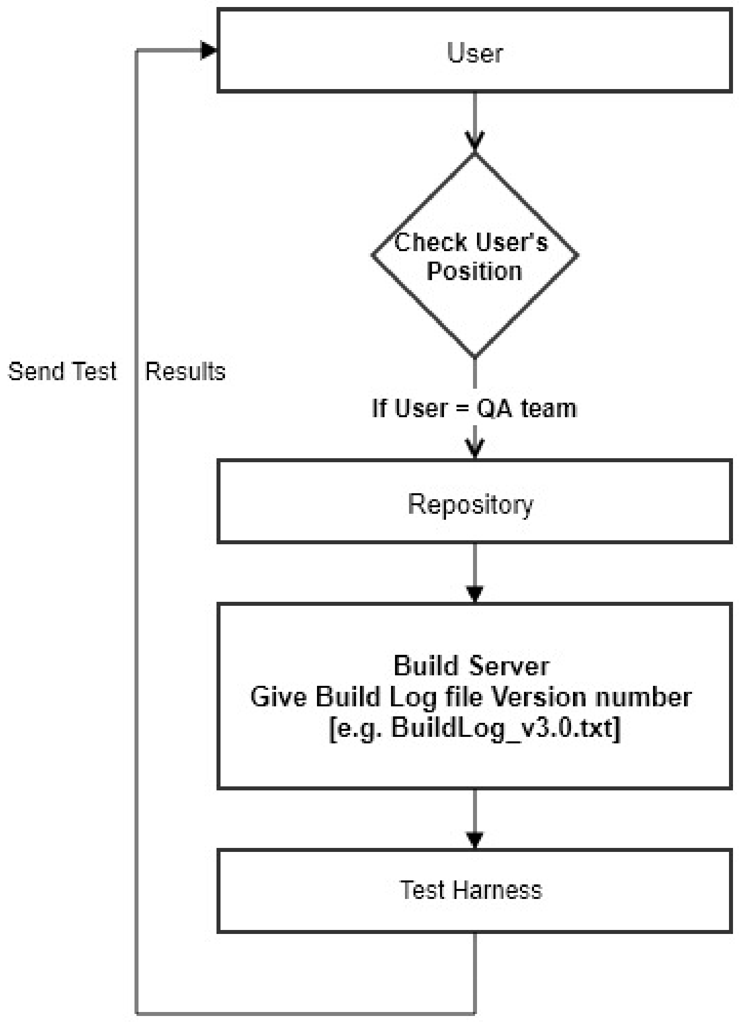


Figure 3.1 Activity Diagram for QA Team

3.4. Release Team –

Release teams decide to roll-out or roll-back the code form production branches after all the above users sign-off the software.

Uses – They need to be able to build and test key tests in case customers report some problem with the final software.

3.5. Customers –

Customers are the users who give the developers the job of software development. Their only Interaction with the build server is via cloud. Release teams push built software libraries to cloud for users to download. Build version info generated by build servers which is kept in the cloud to help users track the version of software they are using.

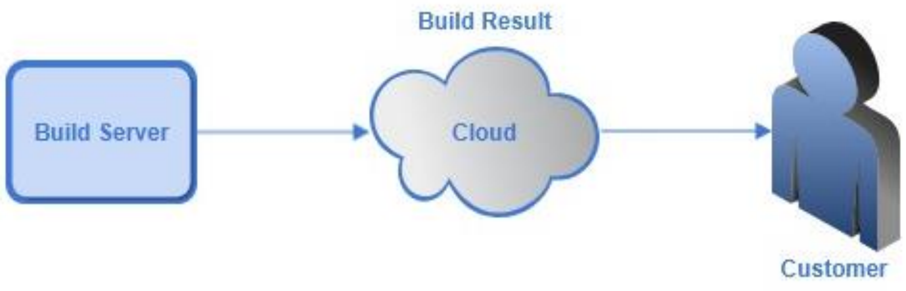


Figure 3.2 Customer-Build Server Connection Diagram

4. APPLICATION ACTIVITIES

1.1. High Level Application Activities

The purpose of presenting activity diagrams is to give clear idea about how activities are going to get performed in the software/application. If the system breaks down, activity diagram helps user to identify the root place of the break down.

The first activity diagram presented below is for the whole federation. Key activities of federation are important to understand in order to design & build the Build Server. Key activities of Federation are –

1. The user interface shown here is for clients. Clients will communicate to the 1st part of federation i.e. Repository using user interface. Clients will upload the test request and dependency files of that test request to the repository.
2. Repository stores the files sent by the clients into respective test request directories in the repository. Multiple clients can send multiple test requests & files.Therefore, main repository will have subrepositories for each client. E.g. if developer sends a test request and dependency code file, then those files will get stored into the developer subrepository.
3. After parsing the test request using XML parser, the first decision node will check the availablity of the test request and its dependency files. If the availablity result comes as true, it will wait for the clients' command to process the test request. If the availablity result fails, the repository will send the notification to client asking him/her to upload all dependent files for that test request and then it will go back to accept next test request.
4. The second decision node is used to take command from client. After sending proper files to repository, client will send a command to process the test request after which build server will pull the files from repository.
5. If client fails to send the command right after he/she uploads the files into repository, the decision node will take decision about the test request depending upon the command status from client and and then it will go back to accept next test request.
6. The build server will take the test request & dependency files from repository and will create the temporary directory for that particular test request. This directory will remain in use until the test request gets built.
7. After accepting files, build server will parse those XML files into the suitable and correct tool chain i.e. C# or C++ to identify the dependency, if any, and component parts.
8. If all dependecy files related to that test request are there, build server will start building the test request.

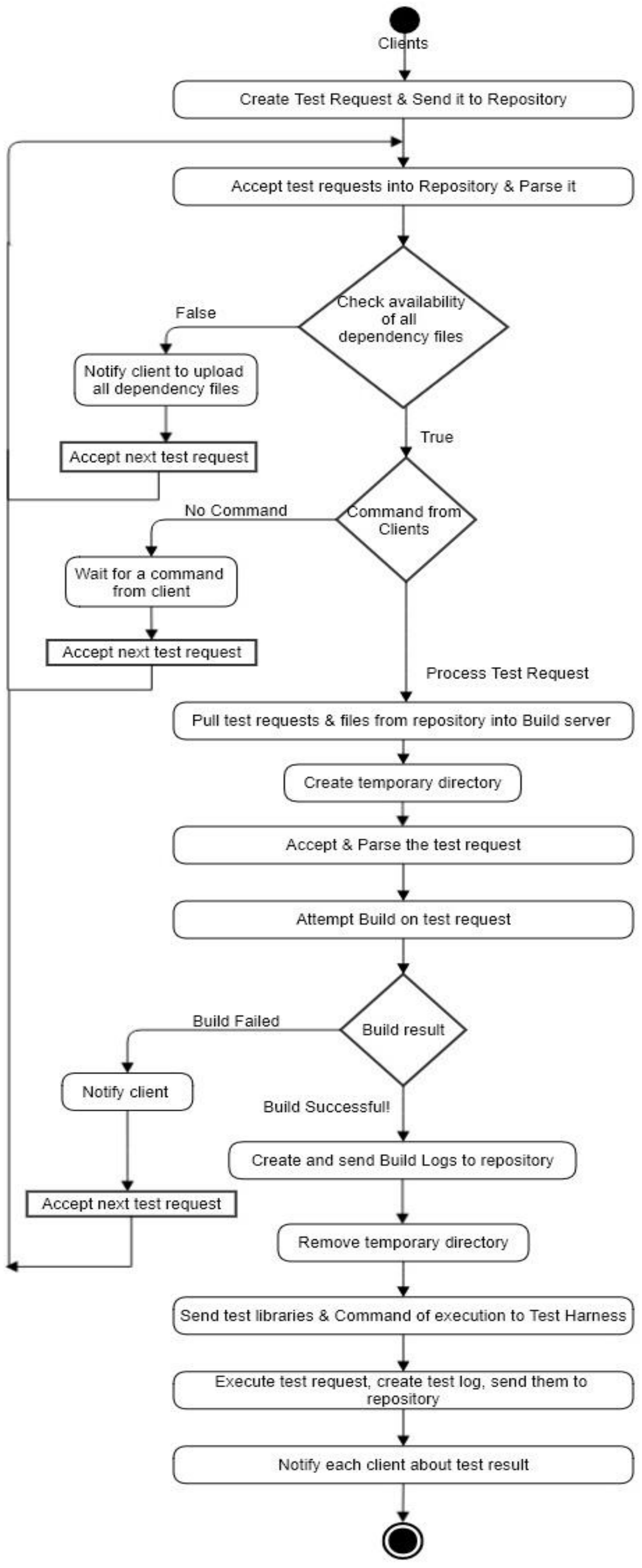

Clients
Create Test Request & Send it to Repository
Accept test requests into Repository & Parse it
Check availability of all dependency files
False
Notify client to upload all dependency files
Accept next test request
True
Command from Clients
No Command
Wait for a command from client
Accept next test request
Process Test Request
Pull test requests & files from repository into Build server
Create temporary directory
Accept & Parse the test request
Attempt Build on test request
Build result
Build Failed
Notify client
Accept next test request
Build Successfull!
Create and send Build Logs to repository
Remove temporary directory
Send test libraries & Command of execution to Test Harness
Execute test request, create test log, send them to repository
Notify each client about test result

Figure 4.1 Activity Diagram of Whole Federation

9. After building the test request, independent of weather the build was successful or not, build logs will get generated. These builds logs are then sent to the repository.
   Additional process – we can connect the build server to the cloud and send the build logs to that cloud, which will then get used by the customer.
10. The third decision node is used to check whether the build is successful or not. If the build is successful, build server will produce the '.dll' files i.e. test libraries and will send these test libraries to the test harness for testing.
11. If the build fails, the client will get a notification about the failure and he/she can access build logs to identify errors of build failure and and then it will go back to accept next test request.
12. After the building of the test request whether it's successful or not, build server will remove the temporary directory and will create the new temporary directory when a new build test request arrives.
13. The test harness will accept the .dll file from the build server and the command to execute the test request. It will load the test libraries and start the testing of the test request with available test cases. Once completed it will generate the test result logs for further analysis and notify clients of the results.

1.2. Build Server Activities

The diagram presented below is the activity diagram of different modules of the build server (Build server = BQueuer + BGenerator + BReporter).

- **BQueuer [Build Queuer]** –

1. There are multiple clients who use the build server and in turn the whole federation to build and run their test requests. When the build server is getting used by developers' team, at a time multiple developers can use the build server to build the test requests. This means multiple test requests from multiple clients need to be built.
2. On receiving the command from the clients to process the test requests, BQueuer pulls test requests and all available dependency files from repository and stores them in Blocking Queue one at a time.
3. As multiple clients are using the build server, sometimes queue might get full with no space left to take the new test request in. The first decision node decides the appropriate action for this situation.
4. If the blocking queue is fully occupied then the build server will notify the client & it will tell the client to wait till the queue frees up a space for the new test request. After this it will go back to accepting test request step and will accept the next test request.

5. If the queue has space to accept the test request, the procedure of building the test request continues.
6. BQueuer will dequeue the first test request from the queue and send it to the BGenerator.

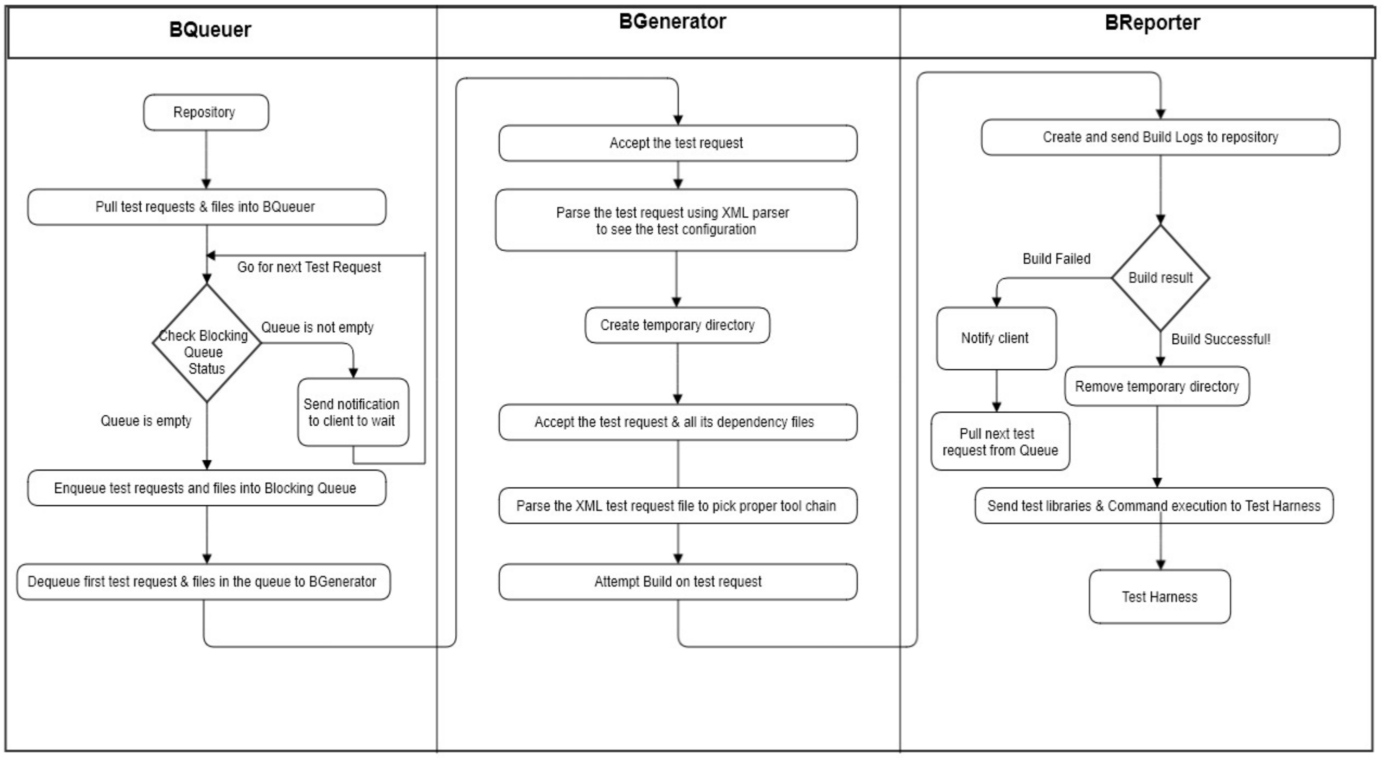

BQueuer
BGenerator
BReporter
Repository
Pull test requests & files into BQueuer
Go for next Test Request
Check Blocking Queue Status
Queue is not empty
Send notification to client to wait
Queue is empty
Enqueue test requests and files into Blocking Queue
Dequeue first test request & files in the queue to BGenerator
Accept the test request
Parse the test request using XML parser to see the test configuration
Create temporary directory
Accept the test request & all its dependency files
Parse the XML test request file to pick proper tool chain
Attempt Build on test request
Create and send Build Logs to repository
Build Failed
Build result
Notify client
Build Successful!
Remove temporary directory
Pull next test request from Queue
Send test libraries & Command execution to Test Harness
Test Harness

Figure 4.2 Activity Diagram for Build Server

The job of BQueuer for this particular test request is done but it will continue its job of accepting build test requests from the clients and arranging test request in a queue.

- **BGenerator [Build Generator] –**

1. BGenerator, if free, accepts new test request from BQueuer and attempts to build. It will parse the test request to check test configuration. XML parser will give details about test requests along with the components involved in that test request.
2. The BGenerator will now take the required dependency files from the BQueuer and will continue the process of building that test request.
3. It will then create the temporary directory for this particular test request to store the dependency files and will name each temporary directory using the client ID, time and date of the request which will make each directory very unique.

4. BGenerator will now parse the test request XML file into the proper tool chain i.e. C# using the XML parser and the XDocument class.
5. It will then send the test request to DLL to build the test libraries. BGenerator will use separate DLL for each test request.
6. In real world scenario, build server gets multiple test requests. To handle those multiple test requests, Process pool will do the work by accepting multiple test requests from the Blocking queue and start processing on them separately.

- **BReporter [Build Reporter] –**

1. BReporter will generate the build log files and send them to the repository for future use.
2. While creating the build log files, the BReporter will give the each log file a unique version name to make them useful to the QA team or code review team.
3. After attempting the build on test request, the second condition node will check whether the build was successful or not.
4. If the build was successful, the BReporter will send the test libraries to Test Harness for testing and it will remove the temporary directory for that particular test request.
5. If the build fails, BReporter will send notifications to the clients regarding build failure and will give them access to the build log for further analysis. It will also keep the failing build directory containing dependency files & other data and name it uniquely. After failure of this test request, it will proceed to next test request.

5. MODULAR STRUCTURE

5.1. Partitions

Partition processing of top level packages describes each package included in the system, how it works and what job is expected from that package along with the communication between all the packages.

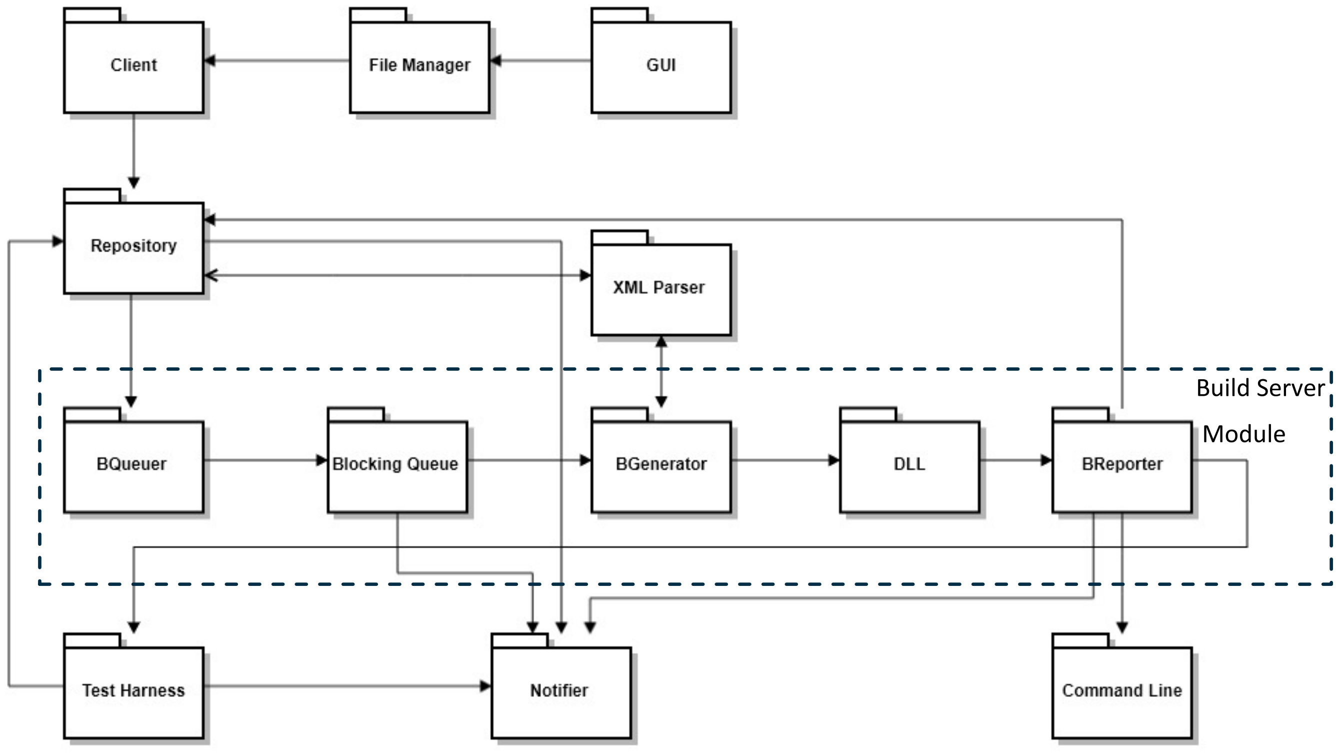
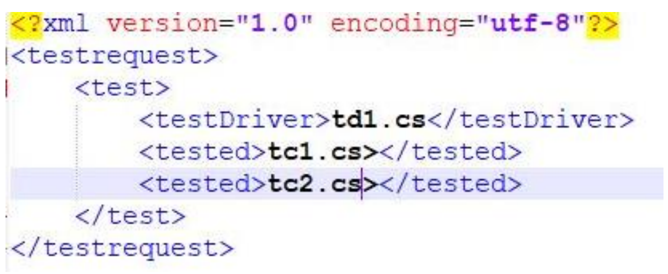


Figure 5.1 Top Level Package Diagram

1. GUI –
   It is user interface package, which will communicate with users. When users want to build a test request, the GUI will ask them to upload the appropriate files.
   Example: For following code, client need to upload the test request test.xml file along with dependency files mentioned in the test request, td1.cs, tc1.cs, tc2.cs.



```xml
<?xml version="1.0" encoding="utf-8"?>
<testrequest>
    <test>
        <testDriver>td1.cs</testDriver>
        <tested>tc1.cs></tested>
        <tested>tc2.cs></tested>
    </test>
</testrequest>
```

This user interface will allow user to browse the files from his/her local desktop. After uploading the files, when user is ready to build, he/she will send the command to build the test request using the provided 'Build' button.

2. File Explorer –
   This package is used when the user hits the 'Browse' button in GUI. This button forces this package to get activated and open the file manager of respective desktop and let the user search for the files he/she wants to upload for the test request.

3. Client –
   This package gets the file selected from the user using GUI and File Explorer and then send the files to the repository to store them on proper branch.

4. Repository –
   Repository package gets activated when user hits the upload button on the GUI. It accepts the files that user has asked GUI to upload and stores them to respective repository.
   As there are multiple users which can send multiple test requests of different levels, the repository will store the test requests and dependency files of the respective user into respective repository branches.

   The repository will then parse the test request by calling the XML parser, and it will check whether all dependency files are sent from user or not. If not, then repository package will call the notifier to send a notification to user.

   Packages for the build server are – Bqueuer, Blocking Queue, BGenerator, DLL, BReporter.

5. BQueuer –
   BQueuer package will pull the test requests and dependency files from the repository and arrange them in the blocking queue. As there are multiple users, chances of getting multiple test requests from multiple users/single user are more, hence the queue is needed as a buffer. BQueuer will accept the test requests only when there is a vacant space available in the queue.

Otherwise, the BQueuer will send the notification to the client regarding the status of the blocking queue.

6. Blocking Queue –

   Blocking queue package allows the build server to enqueue and dequeue one test request at a time. It also sends the current queue status as a notification to help user decide whether to send next test request or not. This package mainly includes 2 functions –

   Enqueue – It allows the BQueuer to add the test request files into the queue one at a time after checking the current status of Blocking Queue.

   Dequeue – This function will dequeue one test request at a time and send it to the BGenerator for further build process.

7. BGenerator –

   After getting a build request from user, this package will first parse the test request to get test configuration and then pull dependency files from the blocking queue. After that, BGenerator will call XML parser package to parse the test request and get the correct tool chain in C# and then will call the DLL package to build the test request.

8. XML Parser –

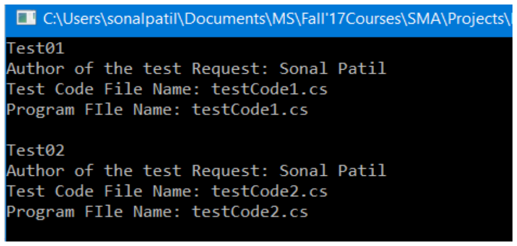
   This package communicates with BGenerator only. It will parse the XML test request file and will provide the proper output file to BGenerator.

   When XML parser package receives the XML document, it will call the XmlDocument class to parse the given files. It will then retrieve the test drivers and other dependency files.

   Prototype –

   I have designed a prototype for XML parser, where I have created a testRequest.xml file containing the test request and all its' dependency files. In the main program, xml file will get loaded from the specified path using XmlDocument class. After loading, I am parsing files name and storing it in List. This XML parser will be also useful when we want to pick correct tool chain in build server. As we are getting file names with their extension we can easily find out the coding language of source code. [Appendix B – Page Number 24]

   Prototype Output –



```
C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\
Test01
Author of the test Request: Sonal Patil
Test Code File Name: testCode1.cs
Program FIle Name: testCode1.cs

Test02
Author of the test Request: Sonal Patil
Test Code File Name: testCode2.cs
Program FIle Name: testCode2.cs
```

9. DLL –

   This package will receive the test request files which are parsed in the XML parser from BGenerator. It will then build the test libraries for that test request. The BGenerator will handle multiple DLLs for multiple test requests but at a time single test request should be sent to one DLL. The output from this package is the test library (.dll) files.

10. BReporter –

    This package receives the test library files from the DLL. It then creates the build logs and sends them to the repository. It will also call the command prompt to display the build logs to users.
    The BReporter checks the result of the build, whether it was successful or not.
    If the Build was successful, it will call the Test Harness package and will provide built test libraries and command to execute the test request.
    If build failed, it will call the notifier package and notify the user about the build failure.

11. Test Harness –

    Test harness package will accept the test library files (.dll) from the BReporter and it will execute the test request by running the test cases on given build library.
    It will produce the test log which will then be sent to the Repository package.
    It will also call the notifier package to notify the users of the test results.

12. Command line –

    This package gets called from the BReporter to show the build logs to the user after the building the test request. This is a command line interface for displaying the build log, to help user identify the build failure, if any.

13. Notifier –

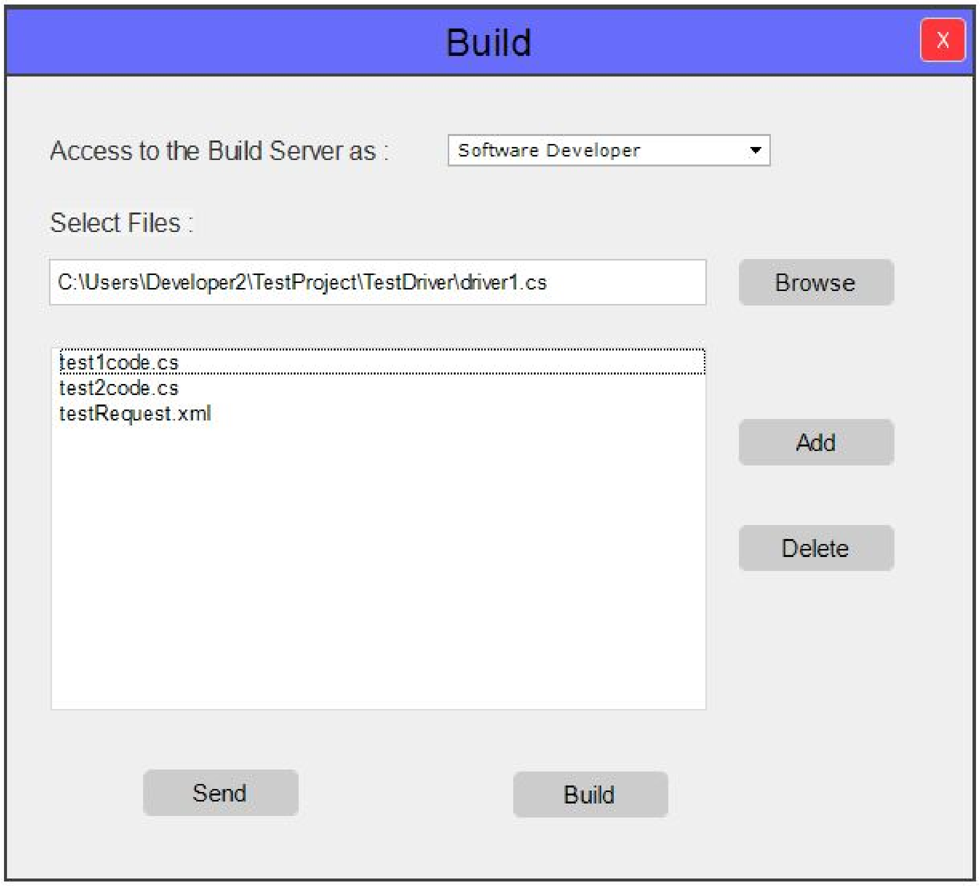
    Notifier is the package which gets called by many packages. It is basically used to give a notification to user regarding the particular change or build result or test result. This might be in the form of email, push message etc. This mainly helps when there are multiple users working on the same project but developing different parts of the system. If one of the team member's code changes and creates a problem, a notification will be sent to everyone informing the problem.

5.2. Proposed User Interface -

User interface helps clients to load files into the repository which they want to get built from build server and then tested by test harness. The proposed user interface is –

1. Having a GUI makes a user friendly interface.

2. As shown in the below diagram, when user want to build a test request, he/she'll open the 'Build' window where they first have to select their access level to the build server. We have 4 kinds of User who will have the access to the build server i.e Software Developers, Managers, Quality Assurance Team, Release Team.
3. Now, they'll browse the files they want to build by hitting 'Browse' button. Then they'll choose the appropriate file and hit the 'Add' button to add the file in the below list.
4. User can add multiple files into the list and can delete unnecessary files from the list by using 'Delete' button.



Access to the Build Server as : Software Developer

Select Files : C:\Users\Developer2\TestProject\TestDriver\driver1.cs

test1code.cs
test2code.cs
testRequest.xml

Figure 5.2 Preferred User Interface

5. After adding all required files, user will send those files to the repository to store them and depending upon the access level chosen by the user, repository will store their files in the respective section.
6. When user gets ready to build the test request, he/she will hit the button 'Build'. This will trigger the build request and will start the build process in build server.

Notifier –

There is one more user interface, Notifier and its job is to notify the users in between the build process. These notifications can be given to the user in the form of email or push notification. There are total 4 notifications will be sent from our system to the user.

1. When the blocking queue gets fully occupied and there are incoming test requests trying to enqueue into the queue, Notifier will send the notification to the user asking him to wait.

   Wait for some time. Queue is fully occupied.

2. When the Repository parses the test request and checks the dependency files with the parsed test request and the files that are missing, it will send the notification to the user through notifier.

   Dependency files for particular test request are missing

3. When BReporter checks whether the build is successful or not, it will send the notification of build result to the user through notifier.

   Build is Successful!

   Build Failed!

4. When Test Harness executes the test request, it will send the test result to the user using notifier.

   Test is Successful!

   Test is Failed!

6. CRITICAL ISSUES

6.1. Complex Builds –

Issue – Performance of the system will get reduced if the builds are more complex and taking too much time. Software systems usually have many developers working on million lines of code using various components like open source libraries, framework etc. In this situation building process becomes complex and sometimes cause the error.

Solution – To avoid the complexity because of long builds, we must make sure that the communication between all packages occurs in the efficient manner without any deadlock conditions. It can also get controlled by using minimum numbers of List, Arrays etc.

6.2. Accuracy –

Issue – When the build server notifies client with the build result and the build log file, the result must be accurate. If the notification is related to dependency files mission, then there must be some files missing in the repository which are not given by users only. If this doesn't happen, and the files are there and still build server is creating the false notification, then this is the major critical issue of the build server.

Solution – This accuracy problem can be solved by increasing the accuracy of each activity in the build server. The main activity to apply on is XML parser. The XML parser should correctly implement the parsing procedure using XDocument class represented in the System.Xml.Linq namespace.

6.3. Performance –

Issue – In real world scenario, chances of generating complex builds are more. They might take longer time than expected like hours to build, increasing the applications' deployment time.

Solution – One solution for this issue is to ask user to upload all the test requests and its' dependency files in the repository for his/her whole day work and giving the build command in the night while the chances of other users' working on the same code are less.

The other solution is, allocation of time for each type of user to send the build request. We can set particular timings for particular user, and when GUI asks the user about his/her access level to the build server, we can check the timing slot before allowing him/her to send process test request command to build server.

These solutions will increase the performance of the build server by saving the time during complex builds.

6.4. Versioning -

Issue – When we'll do the versioning for the build log files so that the user will get to know the latest version of successful/failed build log. In real world, while doing the builds on million lines of code, more than hundred build logs will get created. This might create a problem while storing all those build logs. It will take a lot of memory to hold those build logs.

Solution – The solution for this issue is, we can develop a package in our system which will keep the track of build test request, if the same test request comes to the build server due to previous failed build or due to some new changes, this new package will check whether the test request is same to the previous one, and if the result gets true, it'll delete all previous versions except for the current one and the last successful build log version.
The reason of storing the last successful version is, build logs gets created and stored in the repository without knowing the build result. So the current build might get failed and if we delete all the previous versions' of build logs then it will cause us to lose last successful build log version.

6.5. Test Request passing using Blocking Queue –

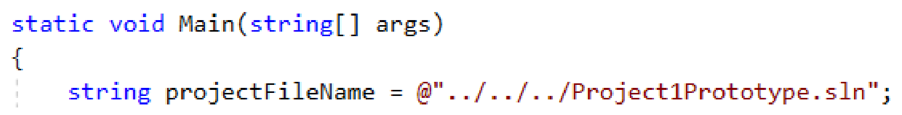
Issue – Blocking queue solves the problem of concurrent access. But sometimes build server might get a lot of build requests more than expected. Build server couldn't process those requests as queue is fully occupied. Unaware of the status of the queue, users keep sending the build requests. This might raise problem where repository is sending the files and build server is unable to catch it because Queue is full.

Solution - One solution to this issue is sending a notification to user to wait till the queue gets vacant space to accept the next requests. Another solution is to provide dynamic blocking queue for the system with the heavy build requests.

7. APPENDICES

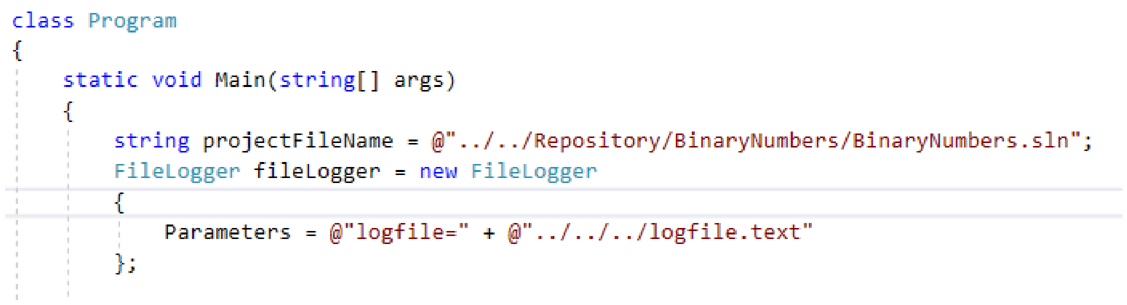
7.1. Appendix 1: Prototype for Visual Studio Code Builder

Input – Path to the solution file which is going to get build. For this prototype, I have specified the relative path of the solution file in the program only. We can also take this as a user input from the command line.



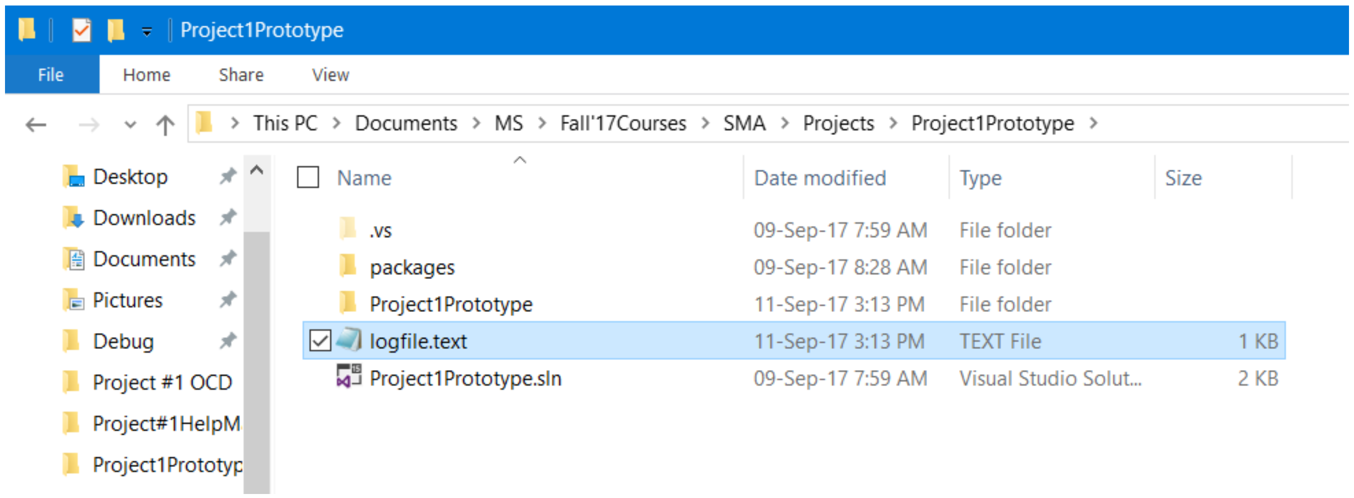
```
static void Main(string[] args)
{
    string projectFileName = @"../../../Project1Prototype.sln";
```

Output – After the build process gets completed, the log file gets generated & stored at the relative path specified in the program. This log file contains the report of success or failure or warning encountered during the attempt of the build.



```
class Program
{
    static void Main(string[] args)
    {
        string projectFileName = @"../../Repository/BinaryNumbers/BinaryNumbers.sln";
        FileLogger fileLogger = new FileLogger
        {
            Parameters = @"logfile=" + @"../../../logfile.text"
        };
```

Path for storing the logfile.txt



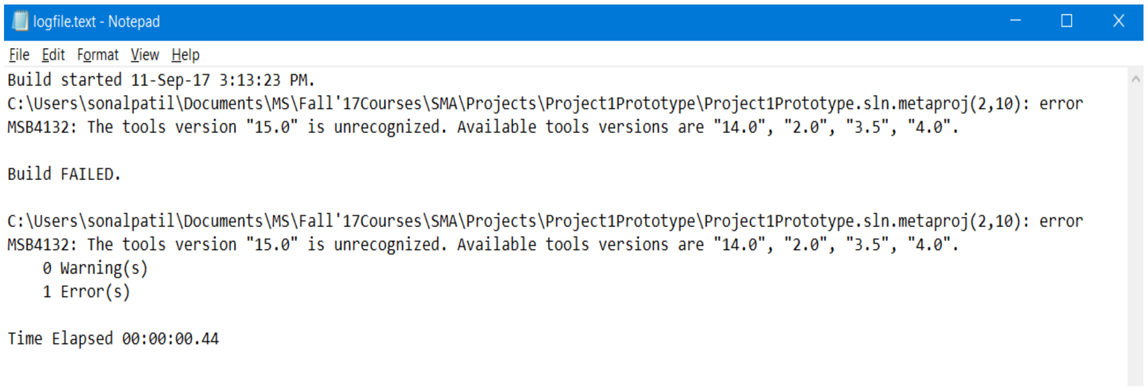
Name	Date modified	Type	Size
.vs	09-Sep-17 7:59 AM	File folder	
packages	09-Sep-17 8:28 AM	File folder	
Project1Prototype	11-Sep-17 3:13 PM	File folder	
logfile.text	11-Sep-17 3:13 PM	TEXT File	1 KB
Project1Prototype.sln	09-Sep-17 7:59 AM	Visual Studio Solut...	2 KB

Generated logfile.txt at the path specified

Description –

In this prototype, I am specifying the relative path of a solution (.sln) file of Visual Studio project and building this file using MSBuild to create the build logs. While implementing this prototype, I encountered with three different outputs.

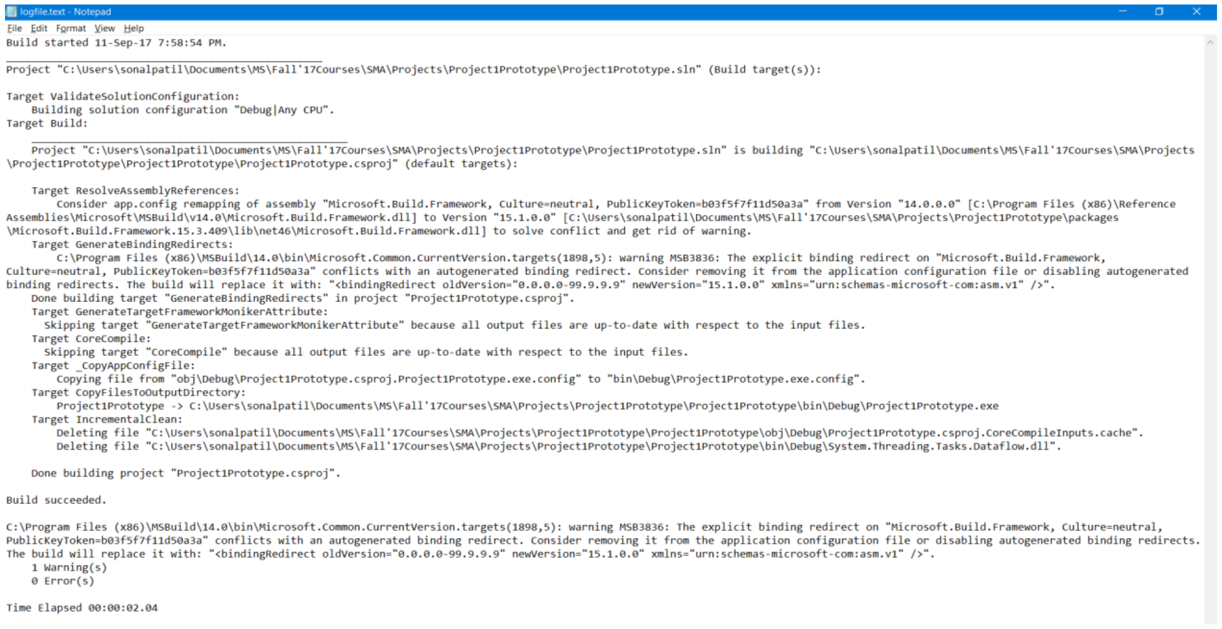
1. Output when Build is failed –
   For this output, the error occurred due to unrecognized MSBuild tool version. To solve this problem I then installed MSBuild v14 which is recognized by Visual Studio 2017.



```
Build started 11-Sep-17 3:13:23 PM.
C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype.sln.metaproj(2,10): error
MSB4132: The tools version "15.0" is unrecognized. Available tools versions are "14.0", "2.0", "3.5", "4.0".

Build FAILED.

C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype.sln.metaproj(2,10): error
MSB4132: The tools version "15.0" is unrecognized. Available tools versions are "14.0", "2.0", "3.5", "4.0".
    0 Warning(s)
    1 Error(s)

Time Elapsed 00:00:00.44
```

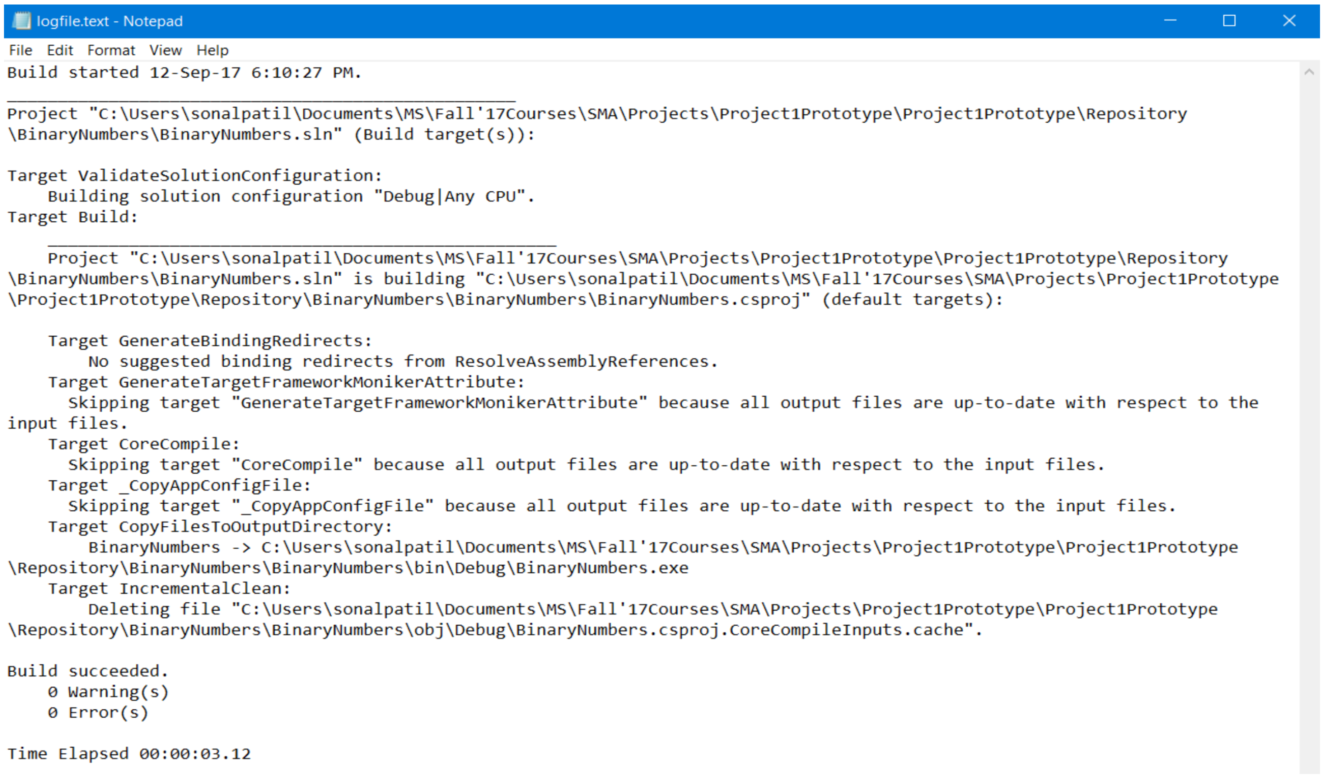
Build result in logfile.txt

2. Output when Build Succeeded with one warning –
   While changing the version to remove the error occurred above, I accedinetly kept the version of ‘Microsoft.Build.Framework ‘ NuGet package version 15.0 and that generated the warning as the version of this package was different than version of other MSBuild packages.



```
Build started 11-Sep-17 7:58:54 PM.
__________________________________________________
Project "C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype.sln" (Build target(s)):

Target ValidateSolutionConfiguration:
    Building solution configuration "Debug|Any CPU".
Target Build:
    __________________________________________________
    Project "C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype.sln" is building "C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects
\Project1Prototype\Project1Prototype\Project1Prototype.csproj" (default targets):

    Target ResolveAssemblyReferences:
        Consider app.config remapping of assembly "Microsoft.Build.Framework, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" from Version "14.0.0.0" [C:\Program Files (x86)\Reference
Assemblies\Microsoft\MSBuild\v14.0\Microsoft.Build.Framework.dll] to Version "15.1.0.0" [C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\packages
\Microsoft.Build.Framework.15.3.409\lib\net46\Microsoft.Build.Framework.dll] to solve conflict and get rid of warning.
    Target GenerateBindingRedirects:
        C:\Program Files (x86)\MSBuild\14.0\bin\Microsoft.Common.CurrentVersion.targets(1898,5): warning MSB3836: The explicit binding redirect on "Microsoft.Build.Framework,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" conflicts with an autogenerated binding redirect. Consider removing it from the application configuration file or disabling autogenerated
binding redirects. The build will replace it with: "<bindingRedirect oldVersion="0.0.0.0-99.9.9.9" newVersion="15.1.0.0" xmlns="urn:schemas-microsoft-com:asm.v1" />".
    Done building target "GenerateBindingRedirects" in project "Project1Prototype.csproj".
    Target GenerateTargetFrameworkMonikerAttribute:
      Skipping target "GenerateTargetFrameworkMonikerAttribute" because all output files are up-to-date with respect to the input files.
    Target CoreCompile:
      Skipping target "CoreCompile" because all output files are up-to-date with respect to the input files.
    Target _CopyAppConfigFile:
        Copying file from "obj\Debug\Project1Prototype.csproj.Project1Prototype.exe.config" to "bin\Debug\Project1Prototype.exe.config".
    Target CopyFilesToOutputDirectory:
        Project1Prototype -> C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype\bin\Debug\Project1Prototype.exe
    Target IncrementalClean:
        Deleting file "C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype\obj\Debug\Project1Prototype.csproj.CoreCompileInputs.cache".
        Deleting file "C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype\bin\Debug\System.Threading.Tasks.Dataflow.dll".

    Done building project "Project1Prototype.csproj".

Build succeeded.

C:\Program Files (x86)\MSBuild\14.0\bin\Microsoft.Common.CurrentVersion.targets(1898,5): warning MSB3836: The explicit binding redirect on "Microsoft.Build.Framework, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" conflicts with an autogenerated binding redirect. Consider removing it from the application configuration file or disabling autogenerated binding redirects.
The build will replace it with: "<bindingRedirect oldVersion="0.0.0.0-99.9.9.9" newVersion="15.1.0.0" xmlns="urn:schemas-microsoft-com:asm.v1" />".
    1 Warning(s)
    0 Error(s)

Time Elapsed 00:00:02.04
```

3. Output when Build Succeeded [No warning, No Error] –
   Finally, after working on the error and warning I got Build succeeded in my build log.
   To go further, I created a new project and put it in the repository folder of current project. What I did is, I put the 'BinaryNumber' project in the Repository of the current 'Project1Prototype' folder. Then I took the .sln from the Repository by giving relative path and build the 'BinaryNumber' project using MSBuild. Below is the snapshot of that successful build log file.



```
Build started 12-Sep-17 6:10:27 PM.
__________________________________________________
Project "C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype\Repository
\BinaryNumbers\BinaryNumbers.sln" (Build target(s)):

Target ValidateSolutionConfiguration:
    Building solution configuration "Debug|Any CPU".
Target Build:
    __________________________________________________
    Project "C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype\Repository
\BinaryNumbers\BinaryNumbers.sln" is building "C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype
\Project1Prototype\Repository\BinaryNumbers\BinaryNumbers\BinaryNumbers.csproj" (default targets):

    Target GenerateBindingRedirects:
        No suggested binding redirects from ResolveAssemblyReferences.
    Target GenerateTargetFrameworkMonikerAttribute:
      Skipping target "GenerateTargetFrameworkMonikerAttribute" because all output files are up-to-date with respect to the
input files.
    Target CoreCompile:
      Skipping target "CoreCompile" because all output files are up-to-date with respect to the input files.
    Target _CopyAppConfigFile:
      Skipping target "_CopyAppConfigFile" because all output files are up-to-date with respect to the input files.
    Target CopyFilesToOutputDirectory:
        BinaryNumbers -> C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype
\Repository\BinaryNumbers\BinaryNumbers\bin\Debug\BinaryNumbers.exe
    Target IncrementalClean:
        Deleting file "C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\Project1Prototype\Project1Prototype
\Repository\BinaryNumbers\BinaryNumbers\obj\Debug\BinaryNumbers.csproj.CoreCompileInputs.cache".

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

Time Elapsed 00:00:03.12
```

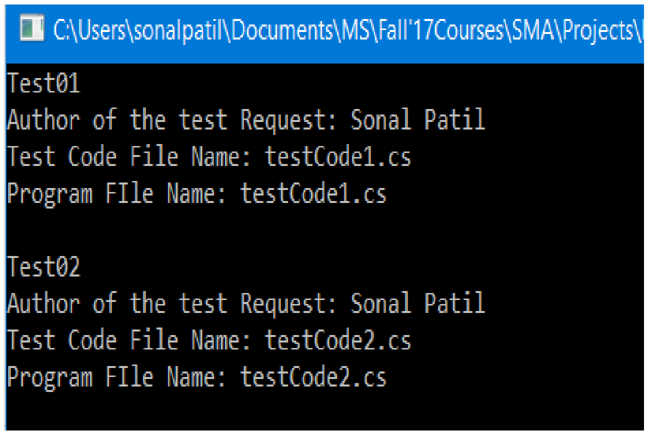
7.2. Appendix 2: Prototype for XML Parser

Input – XML file to parse. For this prototype, I have given the relative path of XML file in the program code instead of asking user to give the path of that file.

```
static void Main(string[] args)
{
    XmlDocument doc = new XmlDocument();
    doc.Load(@"../../testRequest.xml");
```

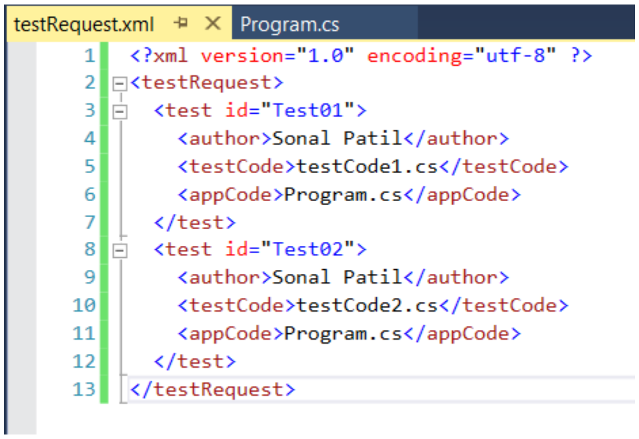
Here, I gave the path of the XML file to XmlDocument class to load that file.

Output – The output of this XML parsing prototype is parsed data of give test request in the user required format. This will get displayed on the command prompt.



```
C:\Users\sonalpatil\Documents\MS\Fall'17Courses\SMA\Projects\
Test01
Author of the test Request: Sonal Patil
Test Code File Name: testCode1.cs
Program FIle Name: testCode1.cs

Test02
Author of the test Request: Sonal Patil
Test Code File Name: testCode2.cs
Program FIle Name: testCode2.cs
```

Output of the Prototype



```xml
<?xml version="1.0" encoding="utf-8" ?>
<testRequest>
  <test id="Test01">
    <author>Sonal Patil</author>
    <testCode>testCode1.cs</testCode>
    <appCode>Program.cs</appCode>
  </test>
  <test id="Test02">
    <author>Sonal Patil</author>
    <testCode>testCode2.cs</testCode>
    <appCode>Program.cs</appCode>
  </test>
</testRequest>
```

testrequest.xml

Description – In this prototype,

1. XML file is the test request file that users will send to the repository for building it. Along with this test request file, users also need to send dependency files.
2. Repository will parse the test request in order to check the availability of the dependency files in order to get dependency file names mentioned in the test request.
3. That is exactly what this prototype does. It loads the test request xml file and by parsing, it gives the files names mentioned in the test request.
4. This might also be useful when you have to again parse the test request XML file to get the proper tool chain (i.e. C#, C++) in order to build the test libraries in the build server. And for deciding the tool chain, we can check the extension of program files uploaded by the user. For example, In this prototype, program file is in .cs format that means build server will pick up the C# tool chain for building that test request.

8. CONCLUSION

The Build Server finds its utility in users like Software Developers, Managers, and Quality Assurance Team etc. working on distributed development projects. It is a useful tool to automate, stabilize and version control the software development process. We have implemented the two prototypes, one for the understanding the build operation and log creation using MSBuild and the other to understand the actual implementation of XML parser package before writing this OCD. In this Operational Concept Document of our build server, we have described the key idea of build server using sample user interface, activity & package diagrams and a few critical issues that should be resolved in order to successfully implement the concept.

9. REFERENCES

1. http://ecs.syr.edu/faculty/fawcett/handouts/webpages/BlogOCD.htm
2. http://edn.embarcadero.com/article/31863
3. http://deviq.com/build-server/
4. https://martinfowler.com/articles/continuousIntegration.html#AutomateTheBuild
5. https://www.joelonsoftware.com/2001/01/27/daily-builds-are-your-friend/
6. http://www.doublecloud.org/2013/08/parsing-xml-in-c-a-quick-working-sample/