

# SMA-681 Software Modelling and Analysis

Instructor- Jim Fawcett

## Project# 1

Core Project Builder-OCD

Submitted by- Nishant Agrawal

SUID- 595031520

Submitted on- 17<sup>th</sup> Sep. 2017

# Index

S.no.	Page No.
1. Executive Summary	4
2. Introduction	5
2.1. Objective and Key idea	5
2.2. Obligations	5
2.3. Organizing principles	5
3. Use Cases	6
3.1. Developer	6
3.2. Project Manager	6
3.3. QA Team	6
3.4. Maintenance Team	7
3.5. Software Architect	7
3.6. Customers	7
3.7. Extending to Final Project	7
4. Application Activities	8
4.1. Mock client sends the files test requests	8
4.2. Store the data on the mock repository	9
4.3. Build server accepts and parses the requests	10
4.4. Build server builds the requests and sends logs, notifications	10
4.5. Test harness runs the test and sends test results, logs and notifications	12
5. Partitions	13
5.1. Executive	14
5.2. TestRequest	14
5.3. RepoMock	14
5.4. DllLoader	14
5.5. Logger	14
5.6. CoreProjectBuilder	15
5.7. HarnessMock	15

6. Critical issues	15
6.1. Test Request Direction	15
6.2. Load on test harness server	15
6.3. File modification risk	16
6.4. Keeping track of build events	16
6.5. References of other libraries in a test library	16
6.6. Redundancy while sending same test requests on same files	16
7. References	17
8. Appendix	17
8.1. Project Builder Prototype	17

## Figures

<b>Figure 1: Activity diagram for a mock client.....</b>	<b>8</b>
<b>Figure 2: Activity diagram for the repository.....</b>	<b>9</b>
<b>Figure 3: Activity diagram for the core build server.....</b>	<b>11</b>
<b>Figure 4: Activity diagram for the test harness.....</b>	<b>12</b>
<b>Figure 5: Package diagram for the core build server.....</b>	<b>13</b>

# 1. Executive Summary

The purpose of the project is to implement a core build server that is going to accept test request from a mock repository, parse to find which tool-chain to use to compile the tested files and finally build and send the test libraries(DLLs) to the mock test harness. The main objectives can be put in bullet points as under:

- Copy the source code files, test drivers and a test request with a test for each driver to a path known to the Core Project Builder.
- Attempt to build each Visual Studio project delivered by the mock Repository using the delivered code.
- Display success or failure of the build attempt and any warnings emitted to the console.
- On success, deliver the built library to a path known by the mock Test Harness.
- Attempt to load and execute each test library, catch the exceptions and display success or failure and the exception messages to the console.
- Build source code from several other widely-used programming languages like C#, C++, Java, etc.

In the future work on the project we are going to expand the functionalities of these mock servers by creating a proper communication channel between servers and also create a interactive user interface for the client.

The intended users of this project are Developer, Project Manager, Software architect, customers and finally the QA and the maintenance team. In the further sections, their uses have been elaborated.

Below are some of the critical issues associated with this project, their solutions have been expatiated ahead in this document.

- Sending test requests directly to the build server or through repository.
- Load on the test harness server when running a very high number of test requests simultaneously.
- Risk of modifying files while they are being used by multiple users.
- Keeping track of building events when there are a lot of test requests being sent from different users/actors.
- Handling the cases when test libraries have multiple references to other libraries.
- Redundancy caused by running the same test again and again.

## 2. Introduction

This document lays and analyzes the immutable concept for a Core Build Server. In today's tech-savvy world, with the advent and requirement of better and robust AI's and IOT's, which can easily be controlled with a very small device using a simple software/application, softwares have pretty much become part and parcel of everything. And in the future, looking at the ongoing efforts towards restoring consciousness after death and integrating bots with humans, that day is not far when softwares will be integrated with our body too.

Such softwares can be from a range of ten lines to thousands of pages of complex code. The smaller ones can be easily developed by a single developer with minimalistic user interface, documentation or architecture. On the other hand, in a large scale software system, there can be multiple simultaneous users(actors/softwares) that access the core functionality through some kind of network. In order to build such softwares, one of the the best practices is continuous integration, where newly generated code is thoroughly tested and once all the tests are cleared, that code becomes part of the software baseline. For this there are multiple services that need to be executed on a federation of servers. A build server is one such server where the building service is executed.

### 2.1. Objective and Key idea

The key idea of the project is to automate a repetitive and boring task of continuous integration where humans can end up overlooking things. Machines are more reliable and faster in this. In a large software system, this can be really beneficial by reducing the costs and resources required. Secondly, the tests can be run 24x7 and the already processed test requests can be reused in the form of their test results.

### 2.2. Obligations

The obligations include implementing the core builder part of a federation of servers where a user can create test requests, build and compile these tests into libraries and finally execute the libraries. The test results, build logs and test logs need to be maintained and the files need to be classified with proper versions.

### 2.3. Organizing Principles

- Use DLLs to execute the tests by compiling the test files into such libraries.
- Extend the functionality by using WCF so these services can run on remote servers.
- Integrate GUI using WPF to ease the process of building and processing test requests.
- Develop independent packages for specific functionality that can interact only through TestExecutive.

## 3. Use Cases

### 3.1. Developer

A Developer who has to write code and make builds numerous times in a day and run tests on a regular basis would have the most advantage using this tool and would be able to save a lot of time using this automated tool. A developer would want to:

- Update and modify the code files- An organized repository server would help maintain this fluidity.
- Access the most latest versions of files- Proper organization and versioning of files on all different servers would help ensure that older as well as the newer versions of files are maintained well, where multiple users are constantly accessing and modifying the files.
- Write new small code snippets, build and run quick tests in order to ensure software rigidity- Using the test harness, adding new code files and running tests on them is going to be quick and easy using this tool.
- Will get logs and reports in case of failures- This tool would ensure that building and testing of hundreds of build and test requests is not blocked by failures by processing the other requests and sending timely notices in case of failures.

### 3.2. Project Manager

The manager who has to lead the projects can use this tool and its parts in the following ways:

- Monitor and keep track of the progress of the project.- No. of build requests will be available for the manager and thereby giving an absolute idea of the project advancement. This will in turn help to draw better and realistic deadlines, also to know when to motivate and push the employees.
- Send timely notifications to superiors and customers about the schedules and estimated time.

### 3.3. QA

A QA can have maximum benefits by using this tool by getting timely test logs and thereby be able to put more time towards building better test-cases and better functional docs and BRDs. A QA can use this tool to:

- Build executable or library with multiple test drivers in a single test request.
- Access previously build test requests and libraries stored in the cache of the build server and test harness.

### 3.4. Maintenance Team

A Maintenance team can benefit by using this tool by:

- Accessing all the older versions of files- The maintenance team would need to ensure that everything is up and running. In case of failures or complaints from clients, they need to be able to run tests on all the versions of the software.
- Running tests in order to resolve issues real fast whether it may be on customers' or their end. They can also identify the problems way faster by checking the old log files.

### 3.5. Software Architect

A software architect would need to:

- Review the project progress in a fast and reliable way.
- Check all the implementations and whether those implementations go hand in hand with the project requirements.

### 3.6. Customers

If this tool is made open to the customers, their developers can resolve their issues faster without needing to wait for the product company to find the problem and respond back.

- For trivial issues, they would want to run tests on their end and save time on the communication back and forth.
- Check out what new implementations are being done and get those upgraded in their software version.

### 3.7. Extending to final project

In the final project, we are going to implement a remote build server with fully functional repository and test harness and not just mock versions. Here, proper communication channel would be ensured by using windows communication foundation(WCF) in order to send and receive files and request messages. Also a GUI would be integrated so as to make the execution of requests and commands can become seamless. This would be done using another .Net framework i.e windows presentation foundation(WPF).

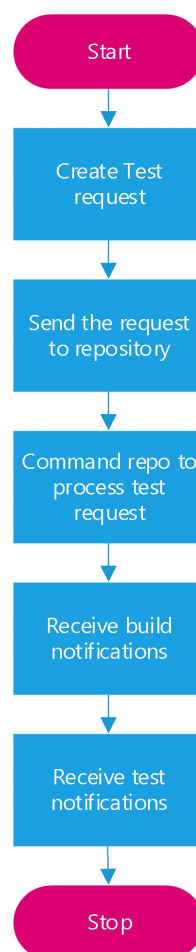
## 4. Application Activities

The key task of the application is to implement a core build server as part of a federation of servers. When projects are huge, relying on manual testing is not the solution. In such cases there is a need to automate this process. Whenever there is code modification, it is imperative to test it against the previous test standards to ensure that nothing is breaking. This build server allows a user to send test requests that are built and tested automatically on command. All the main activities are classified as under:

### 4.1. Mock client sends the files test requests

The activities of the mock client encompasses checking files in and out and sending test requests to the repository. Each test request is an XML file with multiple test drivers i.e tests to run, basically a piece of code. In this project we are going to use pre-built test requests, but in future work we are going to allow the user to create these test requests using a GUI.

All the activities of the mock client are displayed below in the activity diagram.

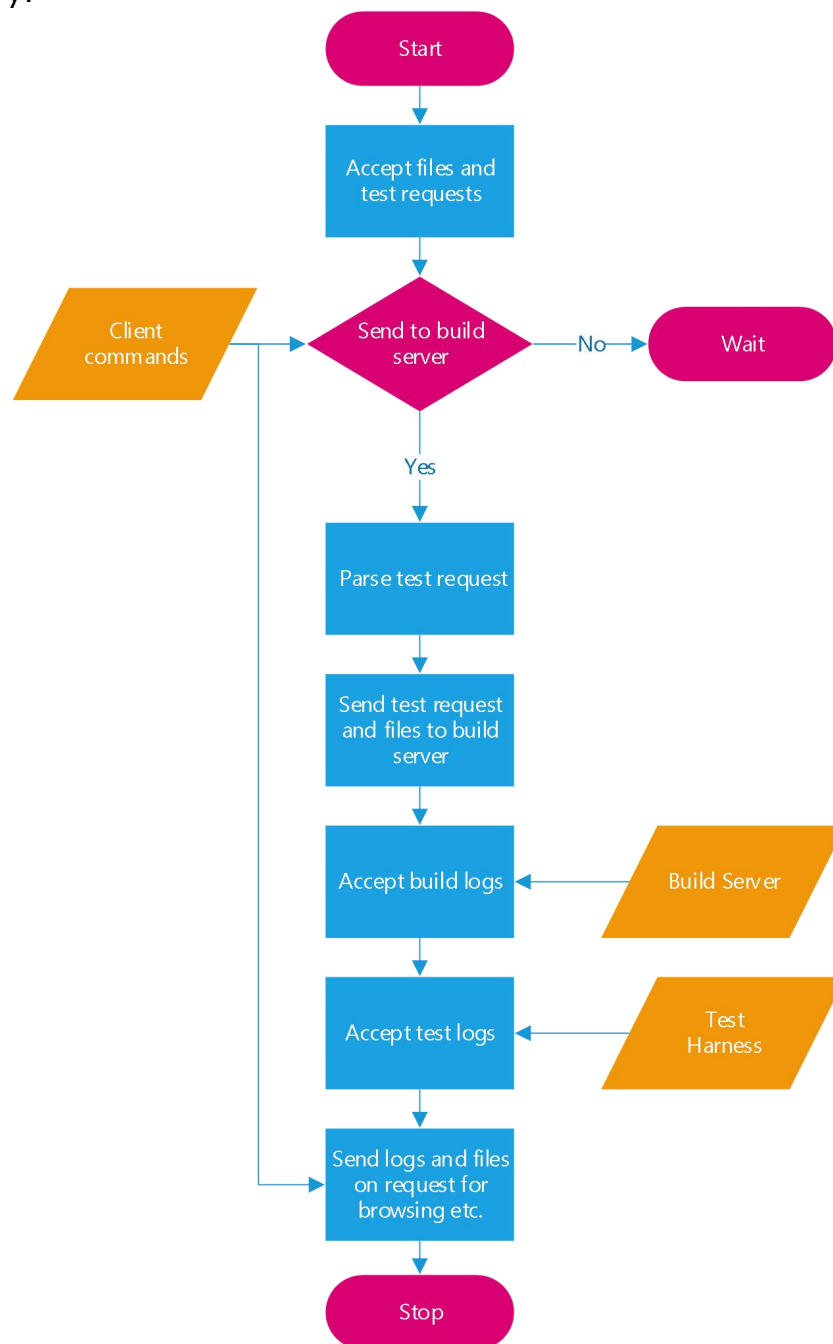




## 4.2. Store the data on the mock repository

A repository is a storage base which sits on a family of directories. For this project we are using only a single directory in accordance with the small scale. It classifies different types of storage for different files. It accepts the test requests from the client and stores them according to the file type. Once it gets the command from the client, it parses the test requests and then sends the requests. The associated files to the build server are sent when it requests for it.

The activity diagram below mentions step by step activities of the mock repository.



**Figure# 1**

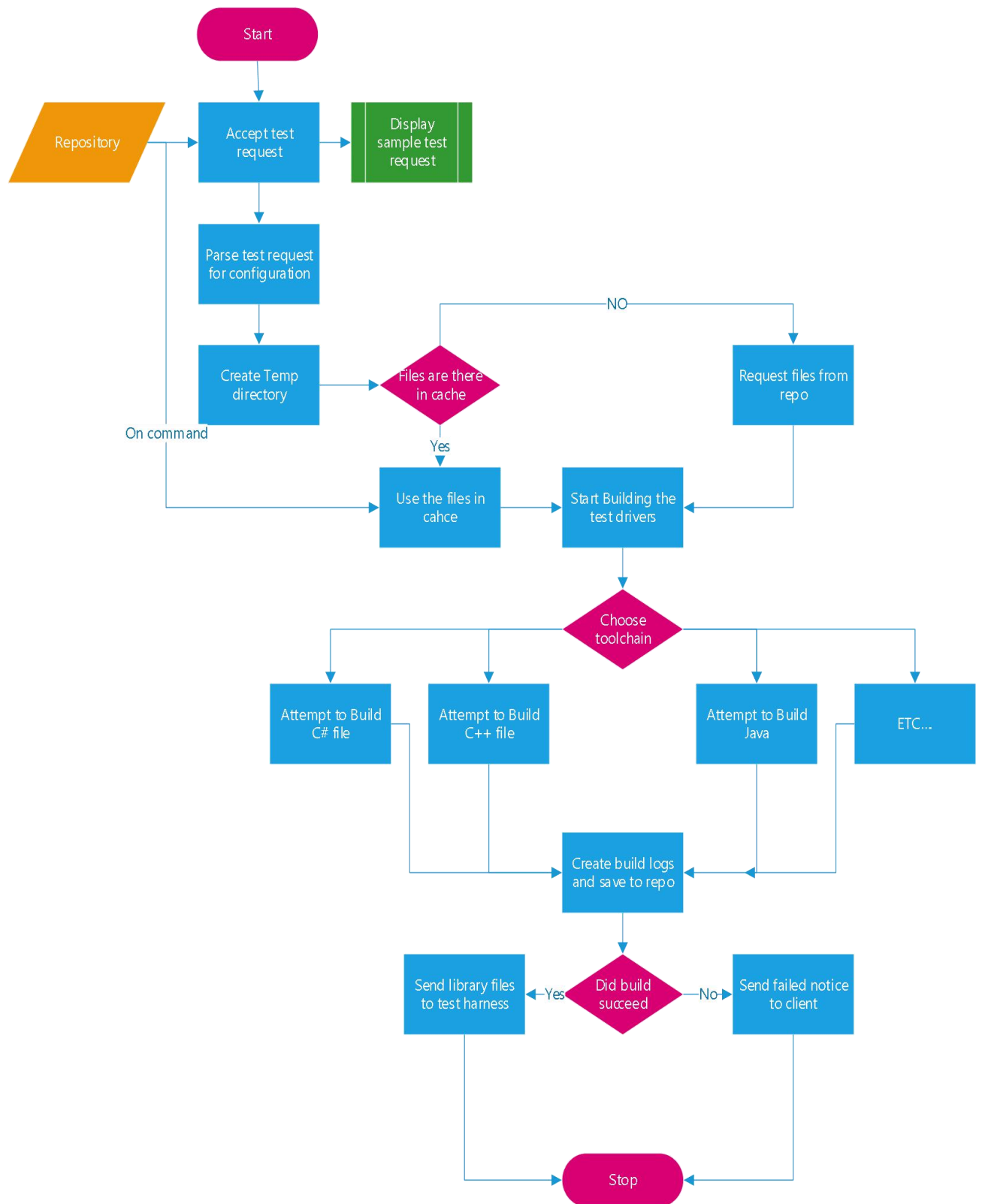
### 4.3. Build server accepts and parses the requests

The build server accepts the test requests and then parses them to find out what kind of code needs to be tested so as to figure out which tool chain to use for compiling the files. This build server also creates a temporary directory to store the test files. Soon after parsing the request, the build server sends the request to the repository for the code files on which the tests need to be run and stores these files in this directory. Once these files are built and sent to the test harness, this temporary directory is deleted.

### 4.4. Build server builds the requests and sends logs, notifications, libraries

As soon as the build server gets the command to build the libraries, build server attempts to build the files into DLLs. If the files are successfully built, the libraries are sent to the test harness. In case of a failure, a notification is sent to the client. Logs are also sent to be stored in the repository whether there is a failure or success while building. After this process is complete, it sends the libraries to the test harness for testing. Now is the time when the temporary directory is deleted.

Activities of the build server are ordered in the correct sequence below on the next page.

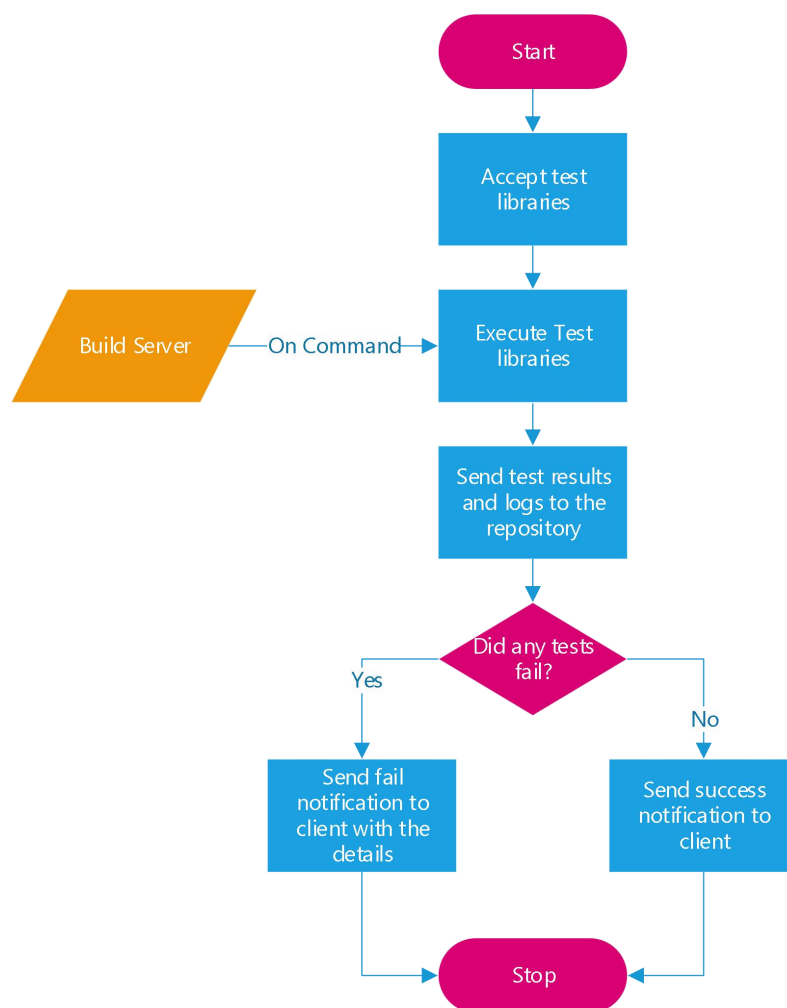


**Figure# 2**

## 4.5. Test harness runs the test and sends test results, logs and notifications

The test harness accepts the libraries from the build server and waits for the command to execute the libraries. Once the harness gets it, the tests start running. As the tests are finished, it sends the test results and logs to the repository to be stored where a user can browse and examine the files and logs. It also sends the notifications about the failed tests to the client.

Activities of the build server are displayed in the correct flow in the activity diagram below.

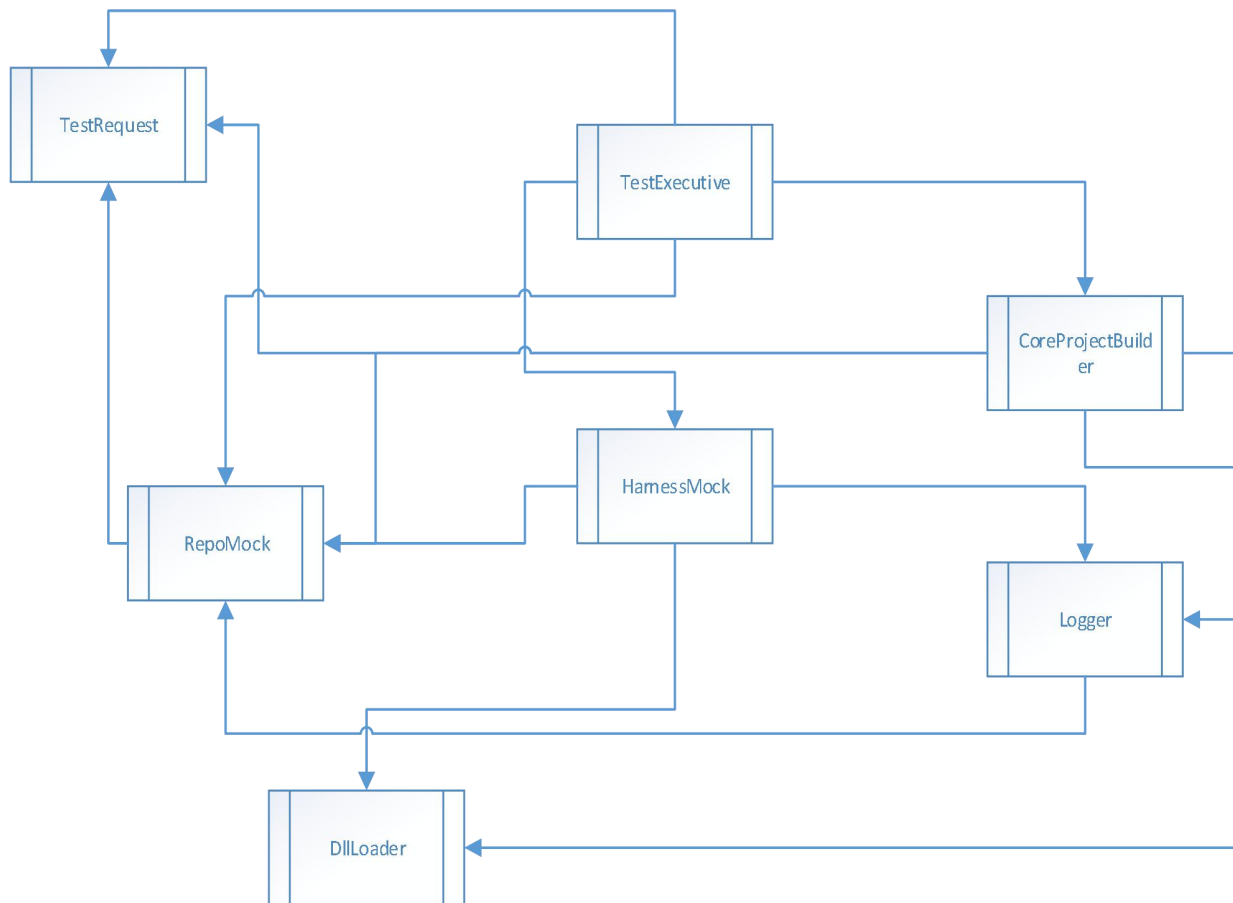


*Figure# 3*

## 5. Partitions

The package partition based on the tasks have been described in this section. A package diagram below displays the interactions of these packages in a clear and fluid manner.

The partition details are explained after this figure.



*Figure# 4*

## 5.1. TestExecutive

The executive is the package where the main control flow of the application enters through. Since this is a console application, we have a main function and it is in the TestExecutive package. So, the execution of the program starts here. Here, we will create test request using TestRequest package. It will also take input from command line to initiate commands to perform particular tasks. Since, we do not have a GUI, the executive acts as the mock client. The executive initiates below sequence of actions

- Build test requests.
- Send the requests to the mock repository.
- Read input and ask for command to parse and send the test requests to the build server.
- Once the notification from the build server is received, asks for permission to start building.
- As the build process is completed, sends the command to execute the test libraries.

## 5.2. TestRequest

This package is responsible for creating a test request by loading and saving it to XML file. After the test request is created, the TestExecutive will send the file to the repository. In this project, we need to parse the test requests twice, once at the repository and a second time at the builder. We can use this package for parsing the XML test request and then load the test drivers in the builder for further processing.

## 5.3. RepoMock

This package simulates the basic functionalities of a repository. It allows accessing and saving files to repository and builder storage. This package uses TestRequest package to parse through a test request and then itself used by CoreProjectBuilder and HarnessMock to access the repository.

## 5.4. DllLoader

This Package helps deal with the DLL files that are used as executables in this project. It interacts with CoreProjectBuilder to help build test files to DLL files and later with test harness to execute these libraries. It can load the assemblies and then run simulated tests present in those assemblies.

## 5.5. Logger

This helper module is going to help create log files and then save them to the repository. We need to create two types of logs: first, the build logs which will be created after the test requests are built into libraries and second, the test logs which will be created after the libraries are executed. Therefore this package is going to interact with repository, builder as well as the test harness.

## 5.6. CoreProjectBuilder

In this package, we will implement the main functionalities of the build server. It will interact with TestRequest to parse the test files and with DllLoader to build the test request and create the assemblies. Finally it will use the Logger package to create log files and send them to the repository.

## 5.7. HarnessMock

This package is where we will execute the test libraries using the DllLoader package and then use the Logger to create and send the logs to the repository. It will also display the failed tests notification at the command prompt and send the test results to be saved in the repository.

# 6. Critical Issues

## 6.1. Test Request Direction

One of the ways to design this tool can be to send the test request directly to the build server which has the primary job of building the test request instead of the repository. But, since the end goal is to have a dedicated server for each type of task, which includes storing files, request building and running the test libraries by remote access, it seems smarter to save everything to the repository. Also it is safer to do all the storage including files and requests on a single server. This not only provides better security and consistency, but also helps in maintaining test request history. Therefore, here the program is designed in a way that the request will be sent and stored in the repository first and then only will these requests be sent to the build server by the command from the client.

## 6.2. Load on test harness server

Server-load is one of the biggest issues that may need to be handled in case too many tests are running on the test harness server at the same time. The chances of getting too many requests to run tests is quite viable as this tool has an array of uses for different users. The best way to resolve this issue is to scale out the task on multiple servers/machines. Maximizing the hardware ensures operation no matter the volume of the upgrade demand.

### 6.3. File modification risk

The biggest risk while using this tool is the chance of file modification which can happen in a number of ways.

If the same file is being updated by multiple users, there can be huge inconsistency and therefore, it can result in a catastrophic failure. This can be resolved by storing the files with proper meta-data, for example author's name and date.

Another issue can be created if the older files get deleted as the newer versions arrive. This can be a cause of concern when a customer is still using the older version of the software. Because of the unavailability of the older version, it might not be possible to run tests to find issues in case of a software failure. Proper versioning of the files not only resolves this issue but also the former one to some extent. It means that once a file is versioned, it becomes immutable and cannot be modified any longer by any user.

### 6.4. Keeping track of build events

In large software systems, there are huge teams dedicated to work on the project which means that on a regular basis there may be hundreds of build requests in a single day. This can result to inconsistencies if there are multiple builds with same name. This can also be resolved by storing the build requests in a proper way by using more intuitive meta-data like user's name, date and time grouped together.

### 6.5. References of other libraries in a test library

In a large project there are bound to be multiple references to other libraries in a test file. In such a scenario, while running test, the test harness is not going to know the names of those libraries. Therefore, it can result in many inconsistencies. Such issues can create binding errors that can be resolved by using delegate, which gets assigned a target method at run-time and acts as a delegate for the caller.

### 6.6. Redundancy while sending same test requests on same files

Another issue can be raised when the same test requests, which have been run before, need to be run again. This can result in serious redundancy and create unnecessary load on servers and thereby increasing the costs. While working on large projects, with many users, this is definitely going to happen. This can be prevented by building cache storage in both build and test harness servers, where recently run test requests and results can be stored. Since cache storage is faster than sending remote requests, this will reduce the effort and also save a lot of time.



## 6. References

- <http://te.ieeeusa.org/2010/Jun/backscatter.asp>
- <https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/presentations/ProjectCenterUseCases.pdf>
- Project helper code by Prof. Jim Fawcett

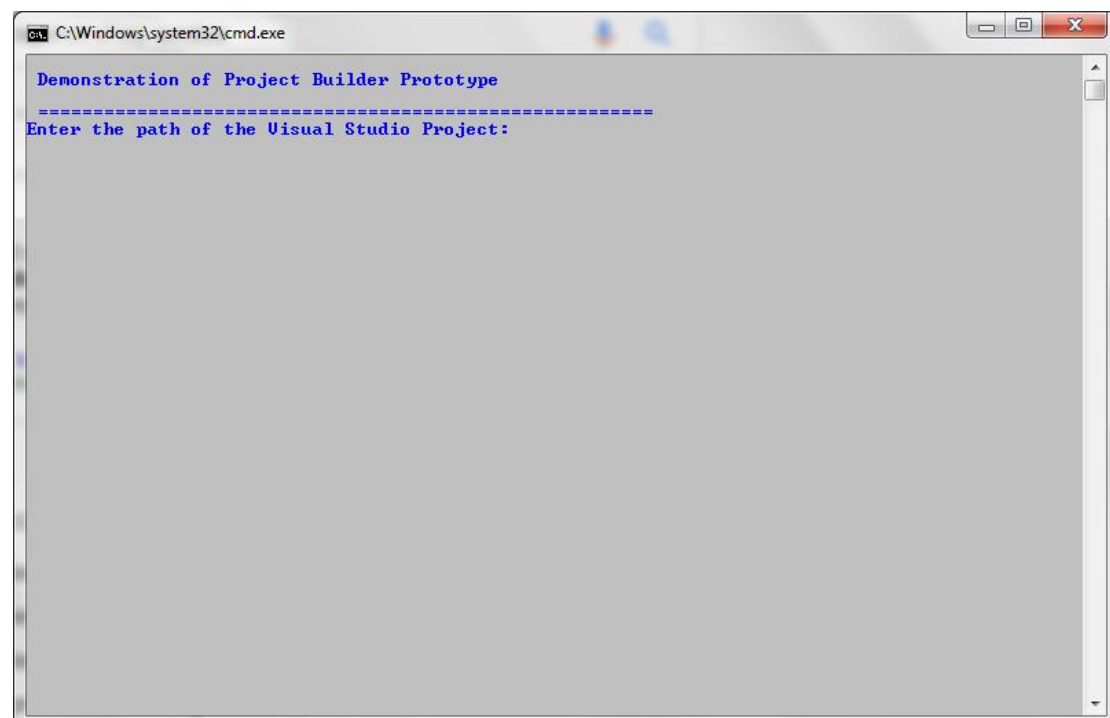
## 7. Appendix

### 7.1. Prototype Project Builder

This section discusses the project builder prototype, implemented to get some understanding of the build process, to be used in the project work. The goal is to load a visual studio project from a given path, access the code and build it.

The general idea used here is to make 2 batch files: one to run the C# code and the other C++ code. Although in the future work, the goal is to try and build several other programming language code, this is a simple console application written in C# just to get things started.

At the start, this application asks the user to provide the project path, following which the program finds the source file with the main method and then attempts to compile that file. Here is an example run for compiling and running a C++ visual studio project with its explanation and output.



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

Demonstration of Project Builder Prototype
=====
Enter the path of the Visual Studio Project:
```

Firstly, the program explores the sub-directories in the user provided path using the `getFiles()` method which in-turn calls `getFilesHelper()` method recursively by using the `System.IO.Directory` class and saves all the files to a List variable.

After that it goes through each file to find the source file with the main method and also verifies its extension. If the extension is `.cpp`, it calls the `CompileCpp()` method where the corresponding batch file `test_cpp.bat`'s path and the file path are sent as arguments to the `runProcess()` method. This method runs the batch file using the `System.Diagnostics.Process` class and sends the file path and name as arguments. Below is the output when this is run on a project with source file `Source.cpp` which does 3 types of sort on given random numbers.



```

C:\Windows\system32\cmd.exe

Demonstration of Project Builder Prototype
=====
Enter the path of a C++ or a C# Visual Studio Project:
C:\Users\nash\Desktop\Csharp ex prac\Testproj
****Attempting to compile the source file: C:\Users\nash\Desktop\Csharp ex prac\Testproj\Testproj\Source.cpp
█

C:\Windows\system32\cmd.exe
Microsoft (R) Incremental Linker Version 14.00.24215.1
Copyright (C) Microsoft Corporation. All rights reserved.

/out:Source.exe
Source.obj

68 53 11 41 64 73 33 22 11 3 94 35 99 67 12 69 38 71 26 47 95 18 16 21 82 92 53 2 4 91 36 27 42
95 91 61 27 81 45 5 64 62 58 78 24 69 0 34 67 41

Sorted using bubble sort...

0 2 3 4 5 11 11 12 16 18 21 22 24 26 27 27 33 34 35 36 38 41 41 42 45 47 53 53 58 61 62 64 64 67
67 68 69 69 71 73 78 81 82 91 91 92 94 95 95 99

=====

41 29 82 18 38 37 23 26 39 44 8 31 76 66 40 56 54 84 23 29 48 93 1 6 50 70 29 90 5 46 48 64 42 4
0 6 88 42 90 35 16 78 29 41 23 59 37 57 62 44 47

Sorted using selection sort...

1 5 6 6 8 16 18 23 23 23 26 29 29 29 29 31 35 37 37 38 39 40 40 41 41 42 42 44 44 46 47 48 48 50
54 56 57 59 62 64 66 70 76 78 82 84 88 90 90 93

=====

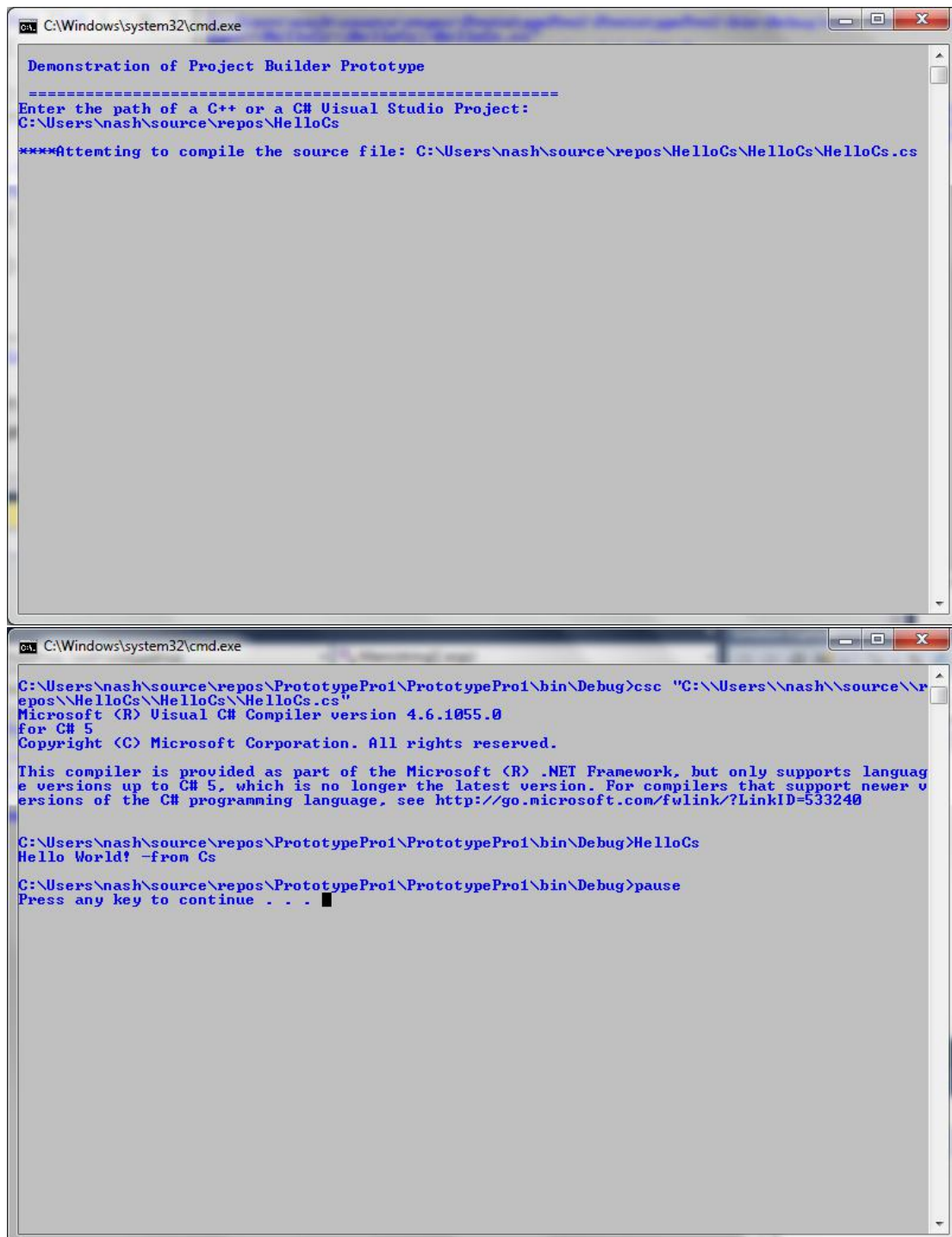
88 21 58 9 9 45 83 53 87 57 37 7 91 7 30 66 24 41 50 50 52 31 74 55 67 55 36 61 90 86 12 97 73 7
7 29 70 72 24 45 21 86 73 6 77 30 4 58 39 15 33

Sorted using insertion sort...

4 6 7 7 9 9 12 15 21 21 24 24 29 30 30 31 33 36 37 39 41 45 45 50 50 52 53 55 55 57 58 58 61 66
67 70 72 73 73 74 77 77 83 86 86 87 88 90 91 97

```

Since this is very simple prototype, so there is no real need for multiple packages. So, I have used only one package 'PrototypePro1'. Here is another example run of the program on a C# project that prints "Hello World! -from cs" to the console and below are the output screen-shots.



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

Demonstration of Project Builder Prototype
=====
Enter the path of a C++ or a C# Visual Studio Project:
C:\Users\nash\source\repos\HelloCs
****Attempting to compile the source file: C:\Users\nash\source\repos\HelloCs\HelloCs\HelloCs.cs

C:\Users\nash\source\repos\PrototypePro1\PrototypePro1\bin\Debug>csc "C:\Users\nash\source\repos\HelloCs\HelloCs\HelloCs.cs"
Microsoft (R) Visual C# Compiler version 4.6.1055.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, which is no longer the latest version. For compilers that support newer versions of the C# programming language, see http://go.microsoft.com/fwlink/?LinkID=533240

C:\Users\nash\source\repos\PrototypePro1\PrototypePro1\bin\Debug>HelloCs
Hello World! -from Cs

C:\Users\nash\source\repos\PrototypePro1\PrototypePro1\bin\Debug>pause
Press any key to continue . . .
```