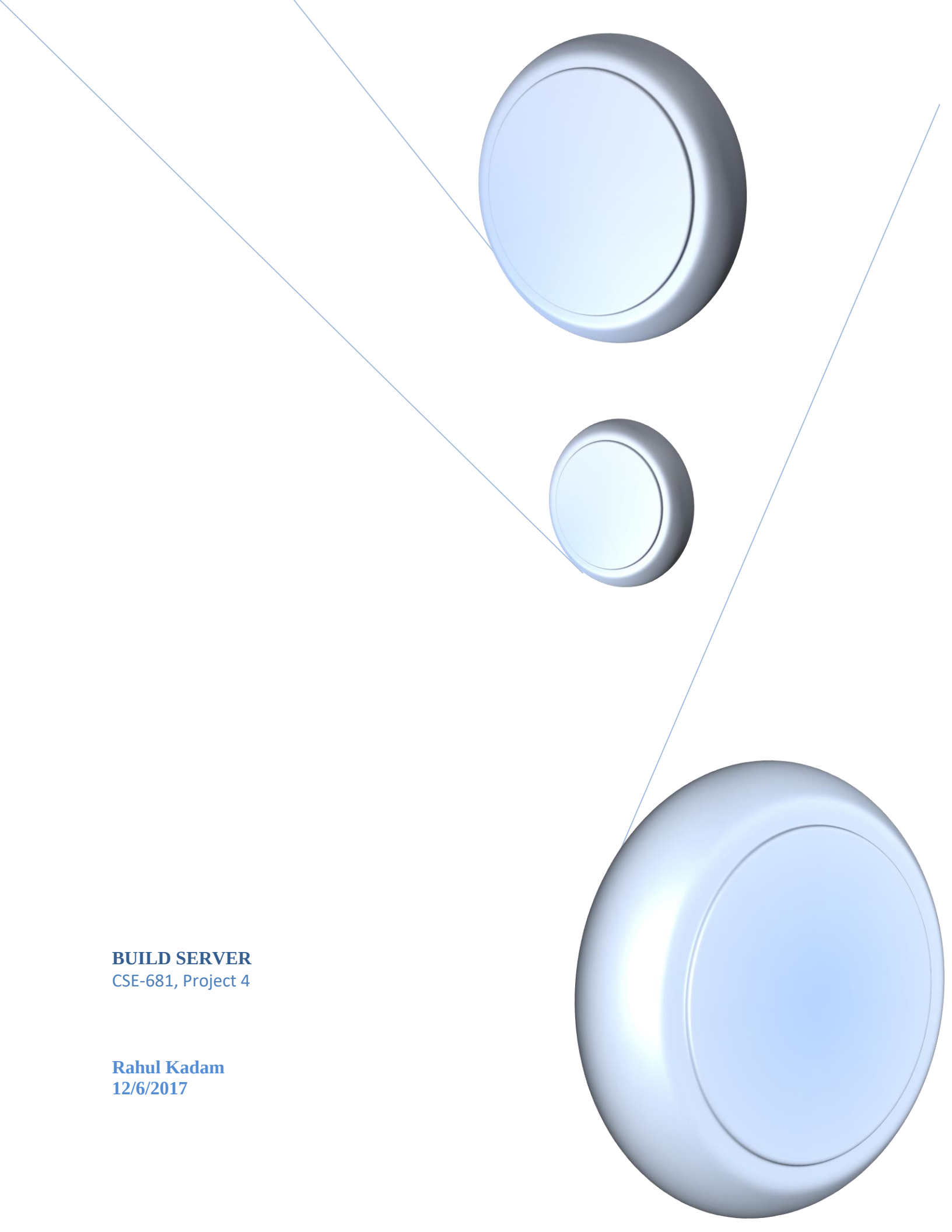

BUILD SERVER

CSE-681, Project 4

Rahul Kadam

12/6/2017

Table of Contents

List of Figures

1. Executive Summary

In this development we created a **Build Server**, capable of building C# libraries, using a process pool to conduct multiple builds in parallel. The implementation was accomplished in three stages.

The first, Project #2 was implemented as a local Build Server that communicates with a mock Repository, mock Client, and mock Test Harness, all residing in the same process. Its purpose was to allow the developer to decide how to implement the core Builder functionality, without the distractions of a communication channel and process pool.

The second, Project #3 developed prototypes for a message-passing communication channel, a process pool, that uses the channel to communicate between child and parent Builders, and a WPF client that supports creation of build request messages.

Finally, the third stage, Project #4, completed the build server, which communicates with mock Repository, mock Client, and mock Test Harness, to thoroughly demonstrate Build Server Operation.

The final product consists of a relatively small number of packages. For most packages there already exists prototype code that show how the parts can be built. For this reason, there is very little risk associated with the Build Server development.

The users of Build Server can be Developers, Instructors, Teaching Assistants, Graders, QAs, Managers. Metaphorically it can be thought of as, a user simply “pushing a (Test) button” and then receiving Build logs and Test Logs automatically.

Few important critical issues include: building source code using more than one language, scaling the build process for high volume of build requests, managing End-Point information for Repository, Build Server, and Test Harness, and using a single message structure for all message conversations between clients and servers. All of these issues have viable solutions which are described in later section.

2. Introduction

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.

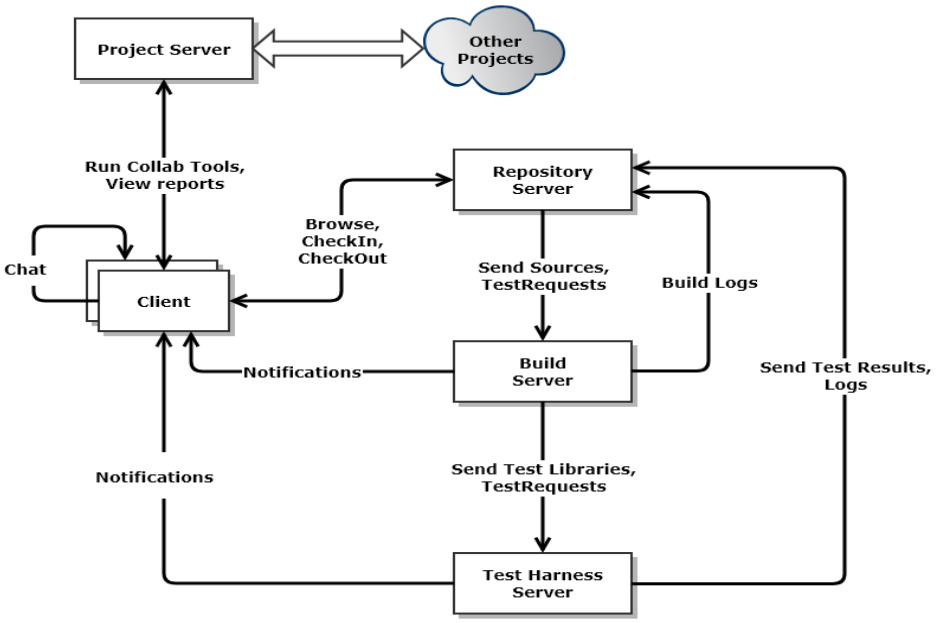

Fig 1. Federation of Servers.

The Federation of Servers components are as follows:

- Repository Server:
  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 libraries for submission to the Test Harness. On completion, if successful, the build server submits these libraries and test requests to the Test Harness, and sends build logs to the Repository. It also notifies the client about the build results.
- 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 build requests to the Repository. It is used to view build logs and test results stored in the repository.

The focus of this development was mainly on “Build Server”. Hence in these projects, we did not integrate our Build Server with a Federation's Repository and Test Harness Servers. Instead, we built mock Repository and Test Harness servers that supply just enough functionality to demonstrate operations of the Remote Build Server. The Build Server uses a "Federation ready" communication channel to communicate with the mock servers, and we have built a mock client that has just enough functionality to demonstrate Build Server working in this environment.

2.1 Application Obligations

Following are the main functionalities that Build Server performs:

- It accepts Build Request from the Repository Mock selected by user using GUI.
- Then, parses and processes the Build Request to receive relevant information.
- It requests for Test files present in Build Request from Repository.
- On Receiving all the Test files, attempts to build the libraries.
- If Build is Successful, it provides these libraries to Test Harness Mock for further process.
- It provides the Build Logs to Repository and notifies Client on Build status.

2.2 Key Architectural Ideas

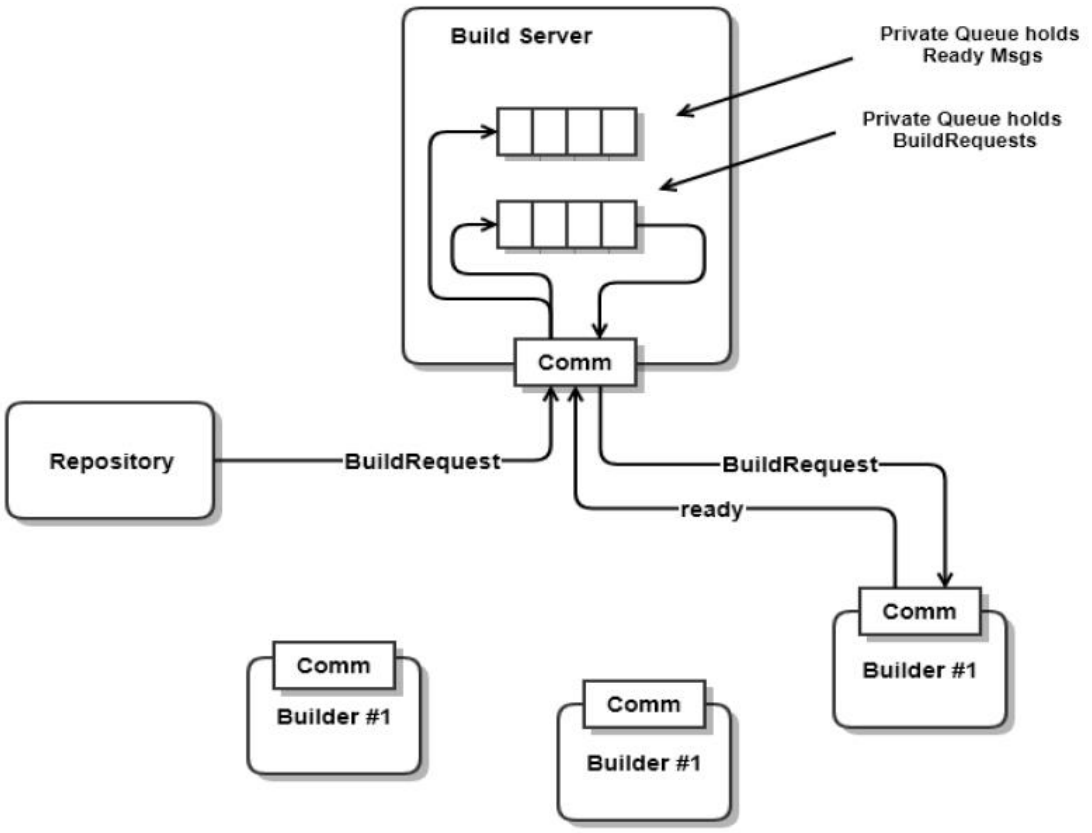

Fig 2. Build Server with Process Pool.

The main idea of Build Server with Process Pool is as follows:

- The Build server may have very heavy workload just before customer demos and releases. We want to make the throughput for building code as high as is reasonably possible. To do that the build server uses a "Process Pool". That is, a limited set of processes spawned at startup.
- The Builder Server Processing is divided in two parts having One Mother Builder and Multiple Child Builders spawned at startup.
- The Mother Builder provides a queue of build requests, and each Child Builder retrieves a request, processes it, sends the build log and, if successful, libraries to the test harness, then retrieves another request.
- The Mother Builder additionally has a queue that holds ready messages sent by the Child Builders after completing building process. This enables the Mother Builder to know which Child Builder has finished processing and is available for processing the next Build Request.

3. Uses

3.1 Developers

Developers are the primary users of Build Servers. 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 provide a GUI (Graphical User Interface). The client interacted with the build server through the console in first version and then in the final version client we have provided a GUI and 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 Build Server.

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 includes 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 an 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 provides good logging facilities with author name, proper time and date stamp. These logs identify the test developer, the code tested including version, and are time date stamped for each test execution.

4. Application Activities

4.1 High Level Activity Diagram

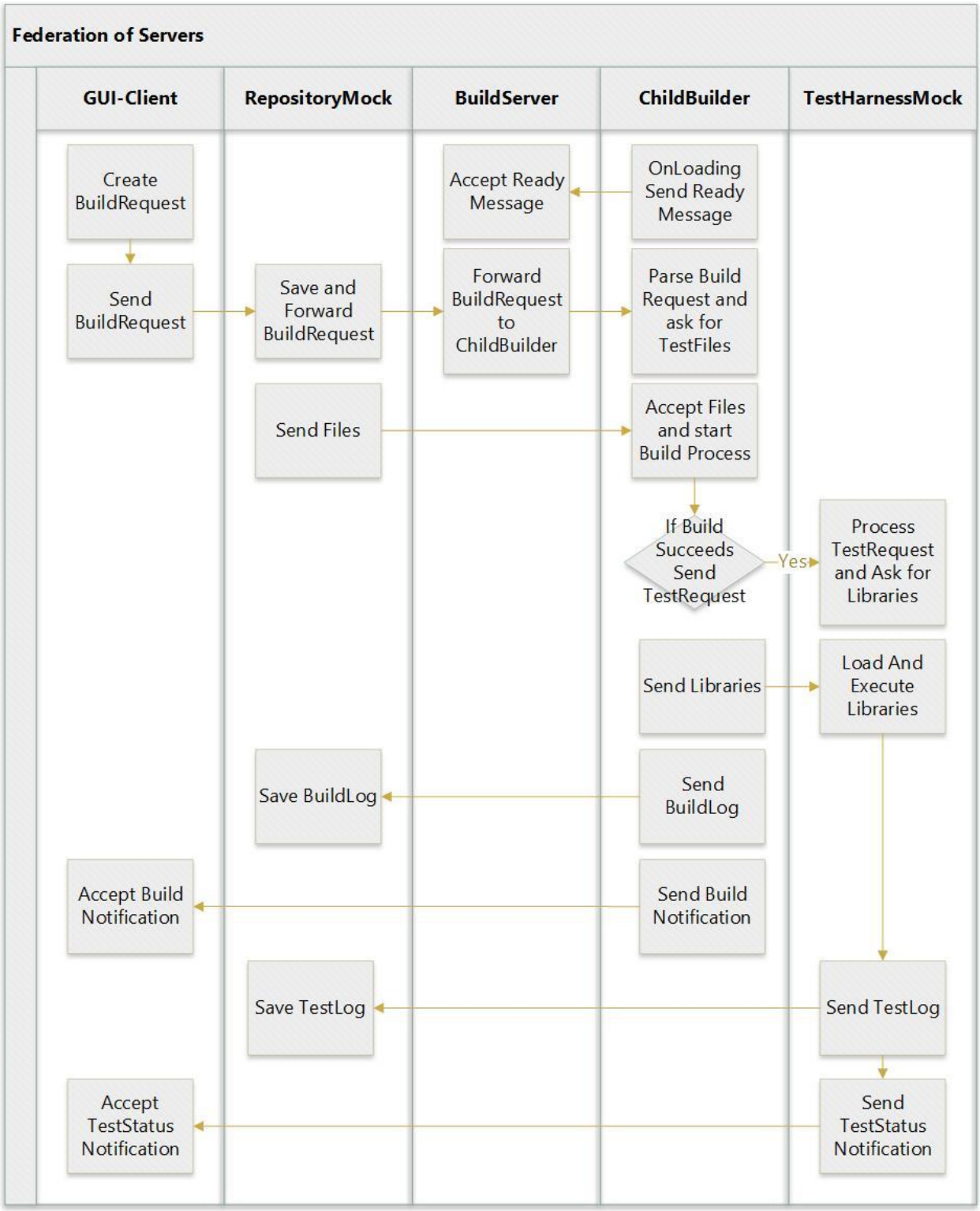

Fig 3. High Level Activity Diagram of Entire System

The above activity diagram describes the flow of activities which occur in the entire system.

Following are the steps:

- **GUI-Client** can be used by user to create a Build Request, which is an xml string consisting of multiple tests where each test has one Test driver and several Test files.
- User can use **GUI-Client** to send this Build Request to **Repository Mock**, which then accepts and stores this Build Request for future use.
- After Receiving the command to execute the Build Request, **Repository Mock** forwards the Build Request to Mother Build Server.
- **Build Server** on startup spawns a finite number of **Child Builders** and these Child Builders after initialized send ready message to Mother Builder indicating that they (Child Builders) are available to process a Build Request.
- Based on these ready messages received from **Child Builders**, **Mother Builder** forwards the Build Request to appropriate Child Builder for processing.
- On Receiving the Build Request, **Child Builder** parses the build request to get the relevant test files (C# source files). Child Builder then asks the **Repository** to send for these test files.
- On Receiving these test files from **Repository**, **Child Builder** start building process.
- If Build Succeeds, then it creates a separate Test Request and send this test Request to **Test Harness Mock** for further processing.
- **Child Builder** sends the build logs to **Repository** Mock for storage and sends the Build Notification to GUI-Client.
- On Receiving the Test Request, the **Test Harness Mock** parsed the Test Request and asks the Child Builder to send the libraries for testing.
- **Child Builder** sends the libraries to **Test Harness Mock** and then Test Harness mock loads and Executes these libraries.
- The Test log of testing these libraries is send to **Repository Mock** for storage and Test Status notification is also sent to **GUI-Client** by **Test Harness Mock**.

4.2 Build Server Activity Diagram

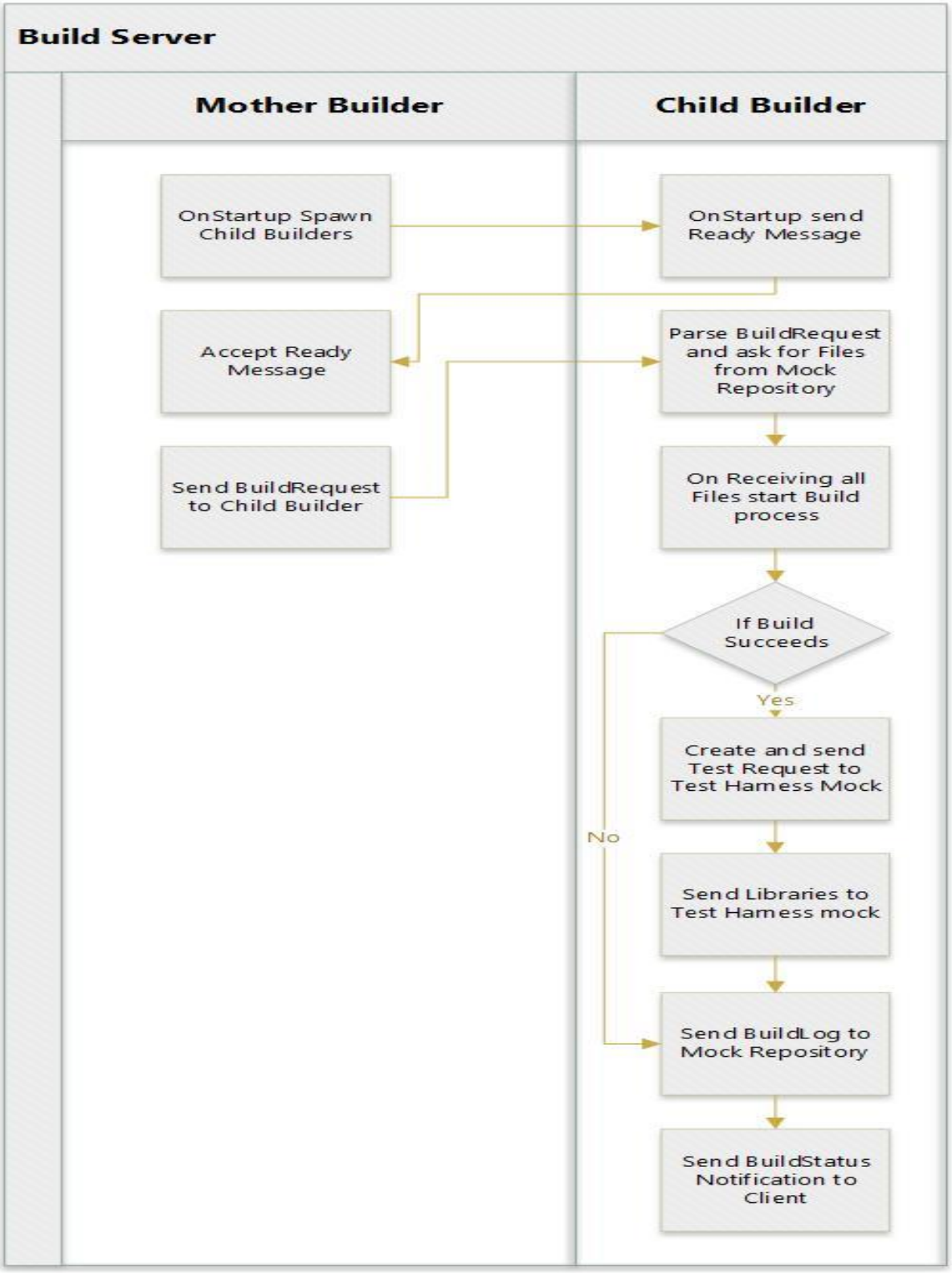

Fig 4. Build Server Activity Diagram

The above activity diagram describes the flow of activities which occur in the Build Server.

Following are the steps:

- The **Build Server** processing can be divided in two parts: One Mother builder and multiple Child Builders.
- The **Mother Builder** on startup spawns a finite number of Child Builders.
- The **Child Builders** on startup send ready message back to Mother Builder indicating that they are available for Build request processing.
- **Mother Builder** accepts ready messages and based on these ready messages received decides the available Child Builder to forward the Build Request for processing.
- **Child Builder** on receiving the Build Request parses the build request – xml string and asks the Repository Mock to provide the required test files.
- On Receiving all the test files, the **Child Builder** starts the building process.
- If Build succeeds, then **Child Builder** create a Test Request and send the Test Request to Test Harness Mock for further processing.
- **Child Builder** also sends the built libraries to Test Harness Mock for execution when asked for it.
- Then, Build Log is send to **Repository Mock** for storage and Build Status notification is send to **GUI-Client**.
- Note: If Build fails, Test Request is not created for Test Harness Mock.

4.3 Message Flow Diagram

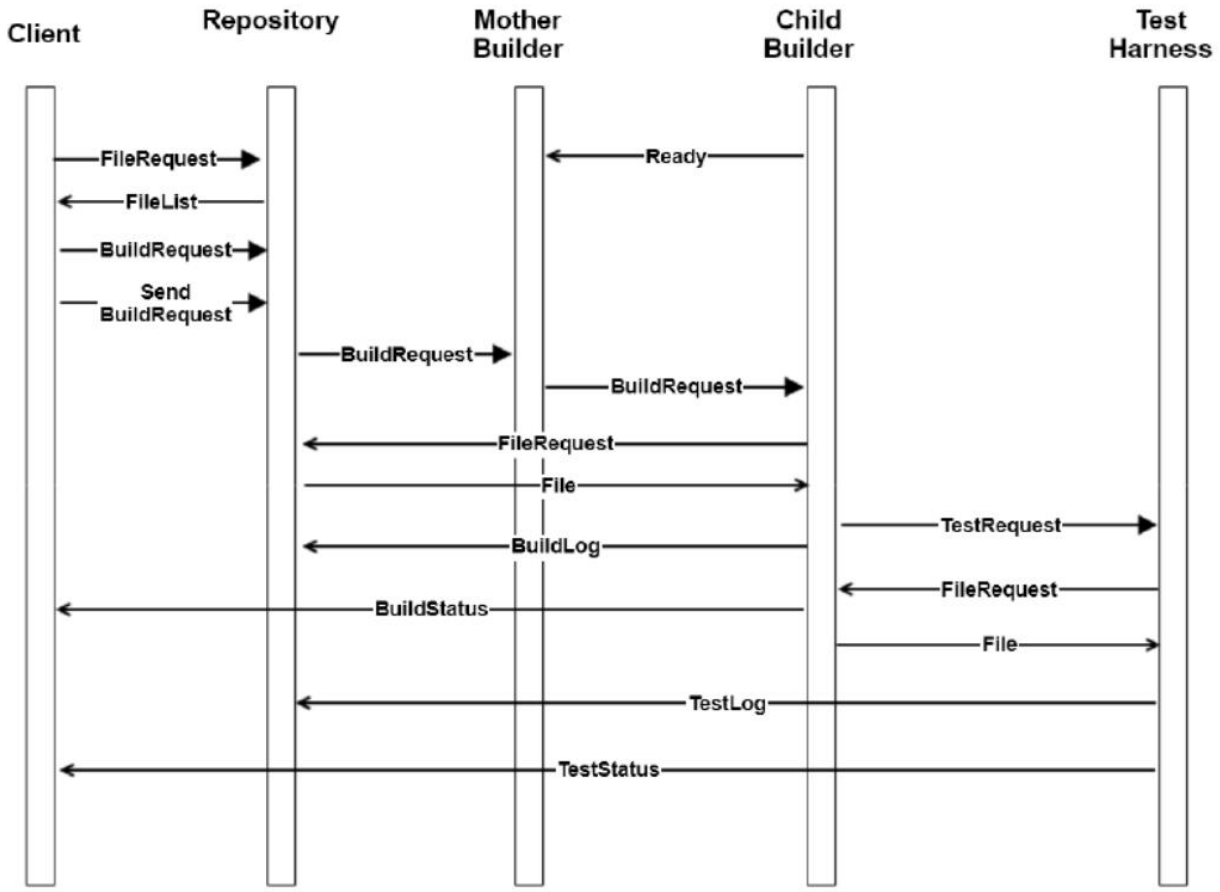

Fig 5. Message Flow Diagram of Entire Application

The above message flow diagram describes the flow of messages which occur in the System.

Following are the steps:

- GUI-Client sends **FileRequest** message to request for list of C# source code files and Build Request xml files present in Repository Mock.
- Repository compiles the list of C# source code files and Build Request xml files present in its storage and sends this list of filenames with **FileList** message.
- GUI-Client on receiving this list of files from Repository Mock, shows them on Listbox for user to select.
- User can select the filenames in Listbox and send **SendBuildRequest** message to Repository, on receiving this message the Repository will save the Build Request into a xml file for future use.

- User can also select the filenames in Listbox and send **BuildRequest** message to directly start building and testing process directly.
- On receiving this **BuildRequest** message, Repository forwards the message directly to Mother Builder for processing.
- Mother Builder on receiving **BuildRequest** message forwards the request to first available Child Builder based on **Ready** message received.
- Child Builder on startup or after completion of building process sends the **Ready** message to Mother Builder indicating that they are available to process the next request.
- Child Builder on receiving **BuildRequest** message, parses the build request for test files information.
- Child Builder then sends **FileRequest** message, asking for test files to be built to Repository Mock.
- Repository Mock sends **Files** to appropriate Child Builder.
- After Completion of Building process, Child Builder sends **BuildLog** message which consists of xml log string to Repository for storage.
- Child Builder also sends **BuildStatus** notification to GUI-Client.
- If build succeeds, Child Builder creates a **TestRequest** message consisting of xml string to Test Harness Mock for processing.
- Test Harness Mock parses the **TestRequest** and asks for libraries to be tested in **FileRequest** message.
- Child Builder sends the libraries as **Files** to Test Harness Mock.
- Test Harness Mock loads and executes these libraries and sends **TestLog** message which xml string log to Repository Mock for storage and **TestStatus** to GUI-Client notifying about completion of Testing process.

5. Partitions

5.1 High Level Package Diagram

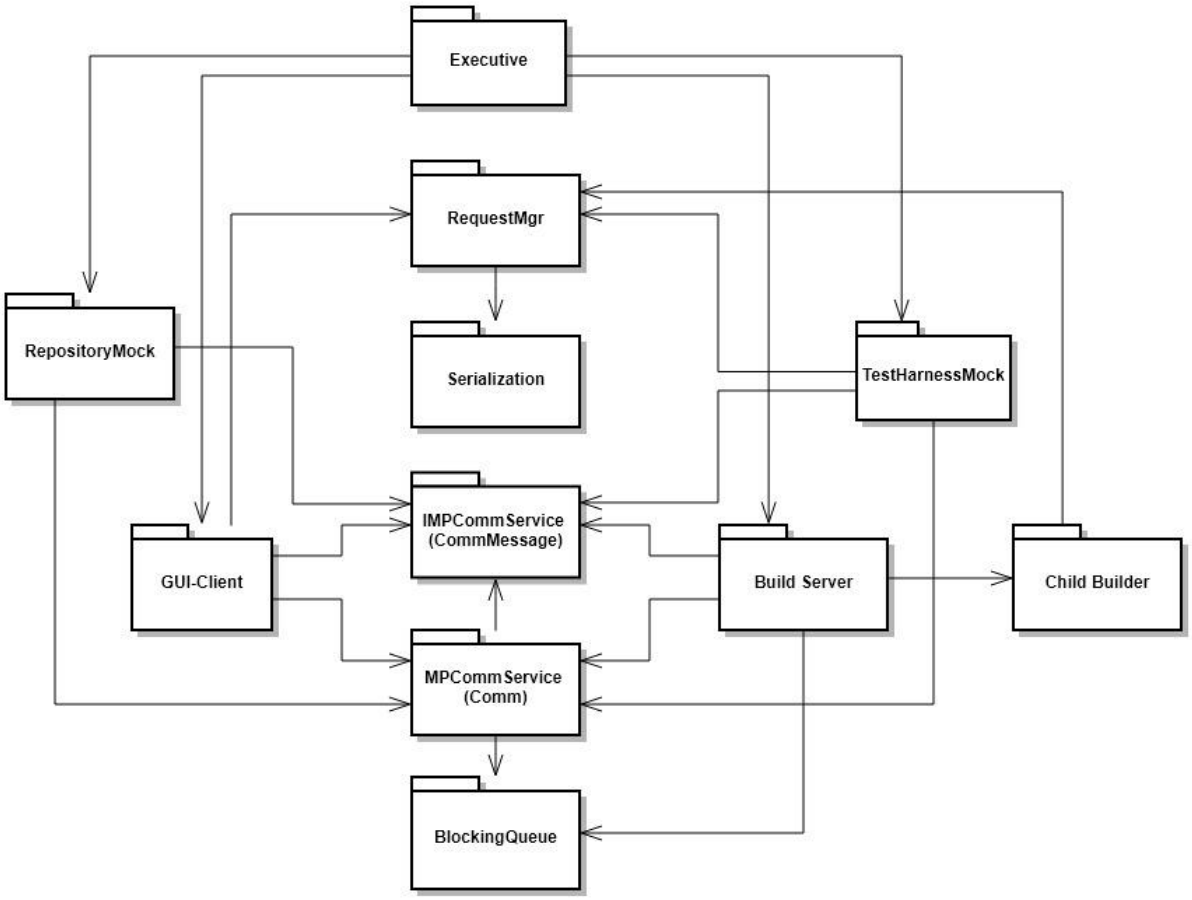

Fig 6. High Level Package Diagram

The Packages involved are as follows:

Executive:

- Executive package is used to demonstrate and explain how the entire system meets the given requirements.
- It depends on GUI-Client, Repository Mock, Build Server and Test Harness Mock packages for demonstration.

IMPCommService:

- IMPCommService package include the WCF data and message contracts used for message passing communication.

- CommMessage class in this package is used by all 5 main packages GUI-Client, Repository Mock, Build Server, Child Builder and Test Harness Mock.

MPCommService:

- MPCommService package defines the Sender, Receiver and Comm classes for message passing communication.
- The Comm class is used by all 5 main packages GUI-Client, Repository Mock, Build Server, Child Builder and Test Harness Mock.

BlockingQueue:

- BlockingQueue package consists of a thread safe queue used to send messages between main components of the system.
- The Comm class in MPCommService package and Builder Server package use BlockingQueue.

GUI-Client:

- GUI-Client is the main interface for users to operate upon.
- It can be used to create Build Request, view stored Build Requests in Repository, command to Build Request and view Build Logs and final Test Results.
- It uses Comm and CommMessage for message passing communication.

Repository Mock:

- Repository Mock stores all the C# source code files, Build Request, Build Logs and Test Results.
- It also enables GUI-Client to view above files effectively.
- It uses Comm and CommMessage for message passing communication.

BuildServer:

- BuildServer package is the Mother Builder which decides to which Child Builder to forward the Build Request.
- It uses two Blocking queues for its processing, one Blocking queue to hold the Build Requests and another Blocking queue to hold ready messages from Child builders.
- It uses Comm and CommMessage for message passing communication.

ChildBuilder:

- ChildBuilder package is the actual Builder which builds the files mentioned in the Build Request into libraries.
- If building libraries succeeds, then it creates Test Request and sends it with its corresponding libraries to Test Harness Mock for execution.
- It also sends Build Log to Repository Mock for storage and Build Notification to GUI-Client.
- It uses Comm and CommMessage for message passing communication.

TestHarnessMock:

- TestHarnessMock uses current AppDomain to load and execute the libraries send by Child builder.
- Initially TestHarnessMock parses through the Test Request and asks to Child Builder for libraries.
- It also sends Test Results to Repository Mock for storage and Test Notification to GUI-Client.
- It uses Comm and CommMessage for message passing communication.

RequestMgr:

- RequestMgr packages is used for managing the Build Request, Test Request, Build Logs and Test Results.
- It is used by GUI-Client to create Build Request, show Build Logs and Test Results.
- It is used by Child Builder to create Test Request and create Build Logs.
- It is used by Test Harness Mock to parse Test Request and create Test Results.

Serialization:

- RequestMgr package which uses classes for Build Request, Test Request, Build Logs and Test Results are all serialized to and from xml string using Serialization package.
- This xml serialized information is sent in the body of CommMesssage class during message passing communication.

5.2 Build Server Package Diagram

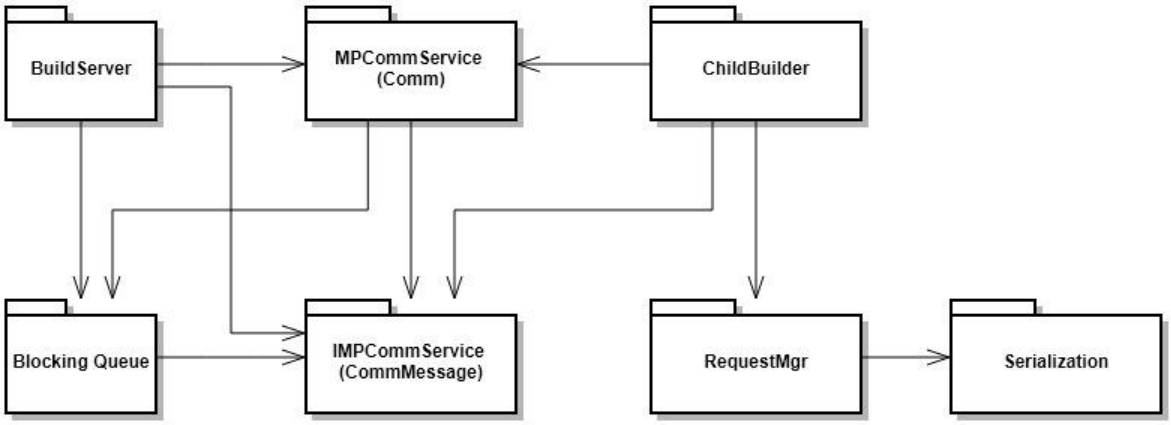

Fig 7. Build Server Package Diagram

BuildServer:

- BuildServer package is the Mother Builder which forwards Build Request received from Repository Mock to available Child Builder.
- It uses two Blocking queues for its processing, one Blocking queue to hold the Build Requests and another Blocking queue to hold ready messages from Child builders.
- It uses Comm and CommMessage for message passing communication.

ChildBuilder:

- ChildBuilder package builds the files mentioned in the Build Request into libraries.
- If building libraries succeeds, then it creates Test Request and sends it with its corresponding libraries to Test Harness Mock for execution.
- It also sends Build Log to Repository Mock for storage and Build Notification to GUI-Client.
- It uses Comm and CommMessage for message passing communication.

RequestMgr:

- RequestMgr packages is used for managing the Build Request, Test Request, Build Logs and Test Results.
- It is used by GUI-Client to create Build Request, show Build Logs and Test Results.
- It is used by Child Builder to create Test Request and create Build Logs.
- It is used by Test Harness Mock to parse Test Request and create Test Results.

Serialization:

- RequestMgr package which uses classes for Build Request, Test Request, Build Logs and Test Results are all serialized to and from xml string using Serialization package.
- This xml serialized information is sent in the body of CommMesssage class during message passing communication.

IMPCommService:

- IMPCommService package include the WCF data and message contracts used for message passing communication.
- CommMessage class in this package is used by all 5 main packages GUI-Client, Repository Mock, Build Server, Child Builder and Test Harness Mock.

MPCommService:

- MPCommService package defines the Sender, Receiver and Comm classes for message passing communication.
- The Comm class is used by all 5 main packages GUI-Client, Repository Mock, Build Server, Child Builder and Test Harness Mock.

BlockingQueue:

- BlockingQueue package consists of a thread safe queue used to send messages between main components of the system.
- The Comm class in MPCommService package and Builder Server package use BlockingQueue.

6. Class Design

6.1 GUI-Client Class Diagram

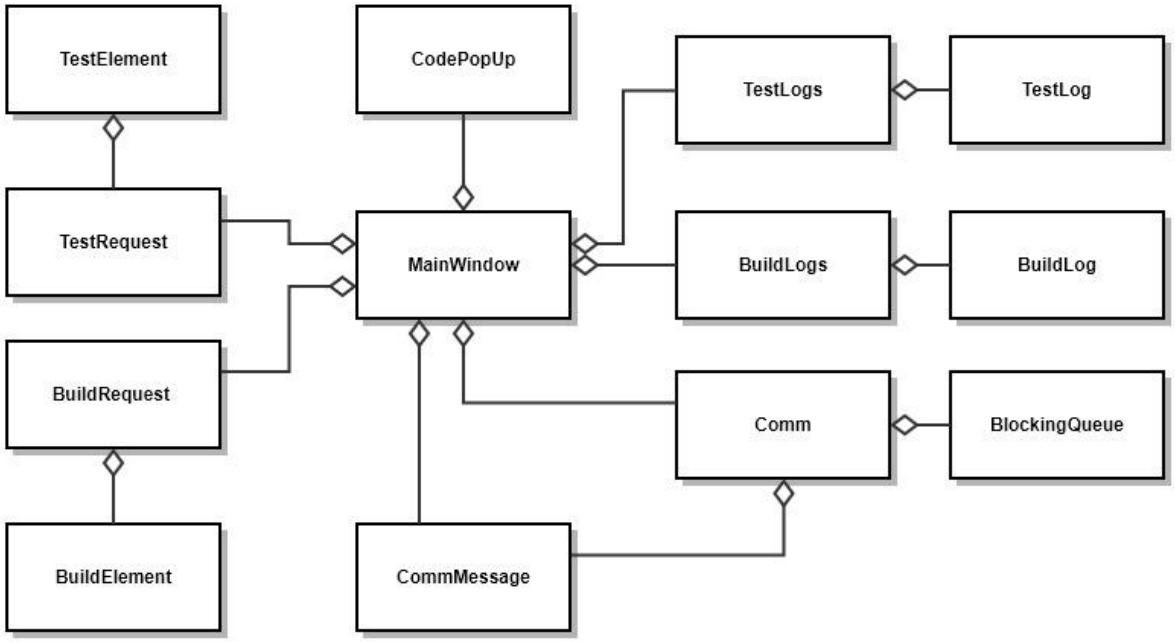

Fig 8. GUI-Client Class Diagram

The above class diagram consists of following main components:

- MainWindow is the main GUI user interface class which aggregates all the other objects.
- CodePopUp class is used in GUI to display file contents on double clicking the filenames on GUI.
- CommMessage class consists of Data Contracts for WCF.
- Comm class consists of Sender, Receiver and both communicate using a thread safe Blocking queue for message passing communication.
- Blocking queue is a thread safe queue.
- BuildRequest and BuildElement classes are used for build request generation.
- TestRequest and TestElement classes here are used to display test request on GUI.
- BuildLogs and BuildLog classes are used to display build logs.
- TestLogs and TestLog classes are used to display test logs.

6.2 Repository Mock Class Diagram

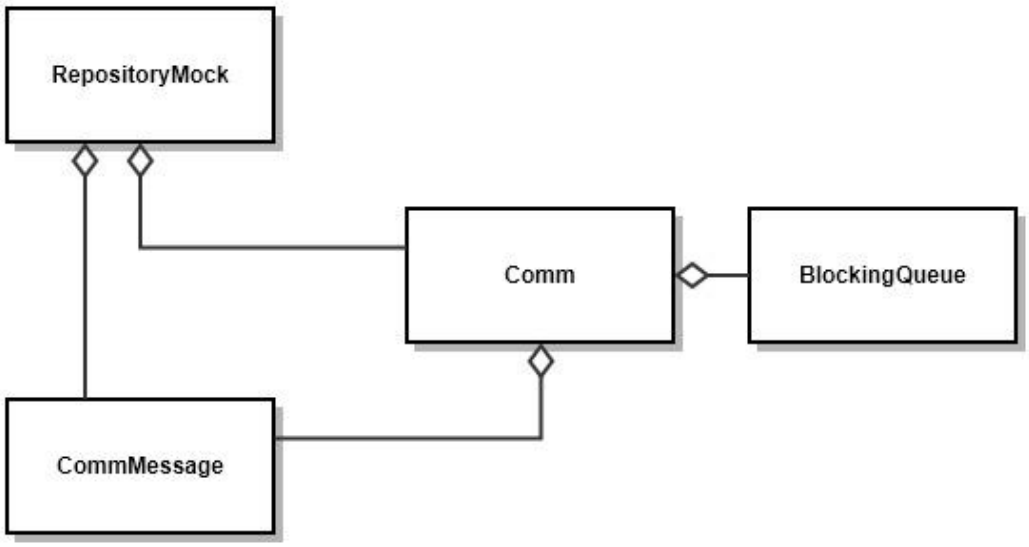

Fig 9. Repository Mock Class Diagram

The above class diagram consists of following main components:

- RepositoryMock is the main class which aggregates all the other objects.
- CommMessage class consists of Data Contracts for WCF.
- Comm class consists of Sender, Receiver and both communicate using a thread safe Blocking queue for message passing communication.
- Blocking queue is a thread safe queue.

6.3 Build Server Class Diagram

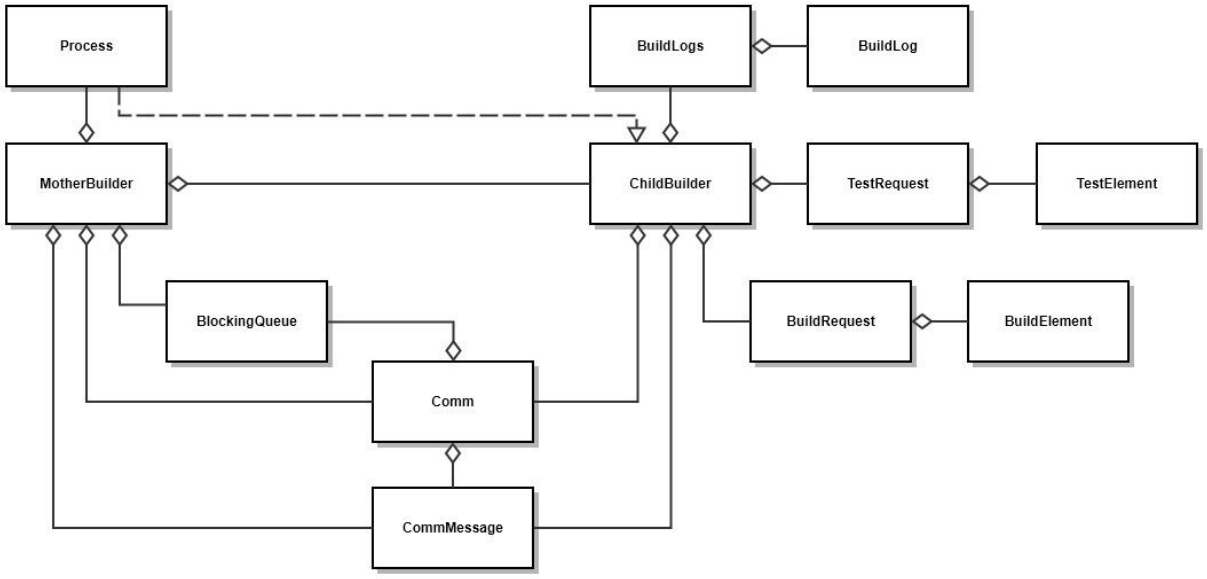

Fig 10. Build Server Class Diagram

The above class diagram consists of following main components:

- MotherBuilder is the main class which aggregates objects such as Process, Blocking queue, Comm and CommMessage.
- ChildBuilder is another important class which aggregates objects such as BuildLogs, Test Request, Build Request.
- MotherBuilder and ChildBuilder user .NET Process class for spawning a process for building C# files.
- BuildRequest and BuildElement classes are used for build request generation.
- BuildLogs and BuildLog classes are used for generating logs for build process.
- TestRequest and TestElement classes here are used generate test request for Test Harness Mock.
- CommMessage class consists of Data Contracts for WCF.
- Comm class consists of Sender, Receiver and both communicate using a thread safe Blocking queue for message passing communication.
- Blocking queue is a thread safe queue.

6.4 TestHarnessMock Class Diagram

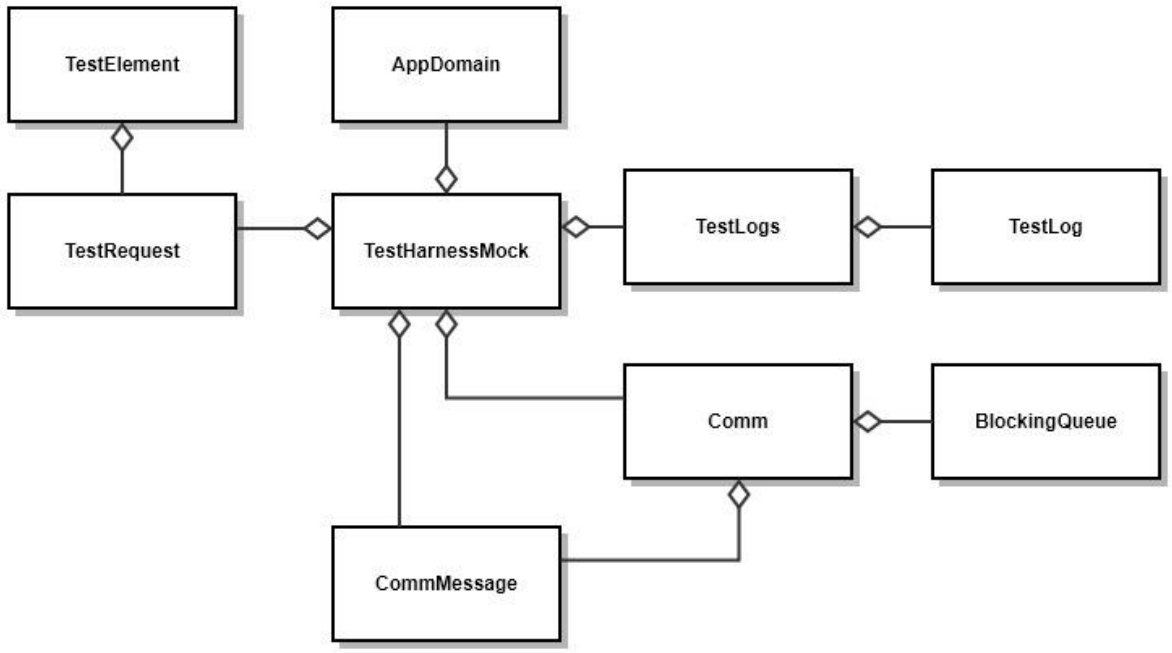

Fig 11. Test Harness Mock Class Diagram

The above class diagram consists of following main components:

- TestHarnessMock is the main class which aggregates all other objects.
- AppDomain uses current AppDomain to load and execute libraries.
- TestLogs and TestLog classes are used for logs for test process.
- TestRequest and TestElement classes are used for test request generation.
- CommMessage class consists of Data Contracts for WCF.
- Comm class consists of Sender, Receiver and both communicate using a thread safe Blocking queue for message passing communication.
- Blocking queue is a thread safe queue.

7. Critical Issues

7.1 Define a single message structure

How to define a single message structure that works for all messages used in the Federation consisting of GUI-Client, Mock Repository, Mother Builder, Child Builder and Mock Test Harness.

Solution

A message that contains To and From addresses, Command string and List of strings to hold file names, and a string body to hold build logs, test logs, build request and test request is used for all needed operations.

7.2 Building and testing both C# and C++ code

A Build server that builds multiple languages like C# and C++ code.

Solution

For building, set environment variables and use tool chain commands for each type of source. For testing, trap exceptions on loading native code libraries in the C# TestHarness and direct to C++ TestHarness. The Current application only builds C# code, building C++ code is one of the deficiencies discussed in later section.

7.3 Managing EndPoint information

Managing EndPoint information for Repository, BuildServer, and TestHarness and GUI-Client.

Solution

Store Endpoint information in XML file resident with all clients and servers and load at startup. The Current application doesn't use such a feature.

7.4 Scaling the build process for high volume of build requests

Build Server could have a high volume of build request from users if application is used in a larger development team.

Solution

Build Server uses a Process Pool where the number of Child processes spawned can be managed by the development team, hence build requests would be processed parallelly reducing the load on Build Server for build requests.

7.5 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 is well designed, various use cases and their impact on design. A well designed Graphical user interface is provided to select the test drivers and the test code as desirable and view build logs and test results effectively.

7.6 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 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 make sure that the interaction between various packages occurs in an efficient manner without any deadlock conditions. The space complexity is also 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.

7.7 Accuracy

The accuracy of the system is a significant issue since the build server 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 be some defect or bug in the code.

Solution

Accuracy is 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.

7.8 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. Current application doesn't support this feature.

7.9 Exception Handling

There might be some unhandled exceptions in the test code, which would be encountered while running the test driver or during build process. 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.

8. Deficiencies

8.1 Building source code in more than one language

Current Application only works with C# source code files. We can extend the application for C++ and Java source files.

8.2 Managing End Point information

For managing EndPoint information for Repository, BuildServer, and TestHarness and GUI-Client we can store Endpoint information in XML file resident with all clients and servers and load at startup. Current Application uses hard-coded EndPoint information for client and servers.

8.3 Store Historical Build Logs and Test Logs

For each Build Request there will be a corresponding Build Log and Test Log in Repository accessible from GUI-Client with similar name. In current application, the build logs and test logs get replaced on testing the same build request multiple times. Instead of replacing the build and test logs we can append new logs to historical build and test logs. Such advance logs can be separated based on timestamp of build request submitted.

8.4 Modifying existing Build Request

Current application GUI-Client can view historical build requests stores in Repository by double clicking the filename in GUI listbox. But modifying or adding new tests to this build requests in not possible. This feature could help when a developer wants to use old similar build request and add new tests to the build requests.

8.5 Download files from GUI

As said previously, user can double click the filename in GUI listbox to view file contents. We can add additional download feature where user can download these files in Client Storage. Build Request, C# source code files, Build Logs and Test Logs which are not just viewable in WPF window but also downloadable to a certain folder would be helpful.

8.6 Check-in and Check-out source code in Repository

Client in current application cannot check-in and check-out code from Client storage to Repository storage. Current application assumes that the C# source code files are already present in Repository storage for build and test.

8.7 TestHarnessMock Child App Domain

In Current Application for loading and executing libraries in Test Harness Mock, no Child App Domain is created, while for each Test Request a new library(dll) is created and loaded in current AppDomain. Additionally, there is no unloading of libraries from current AppDomain. Such issue can be resolved by having an advance Test Harness with a Thread Pool of child app domains with ability for loading and unloading the libraries. Current application deletes the historical libraries(dlls) on startup of an application, hence resolving any storage issues.

8.8 Update Status in bottom tab of GUI.

Even though the current Application does have a status bar at bottom of GUI, the status is not updated while using GUI features.

9. User Manual

There are two tabs which divide main functionalities as:

1. Creating Build Request and
2. Building/Testing Build Request, Viewing Build Logs and Test Result

Tab 1 – Creating Build Request

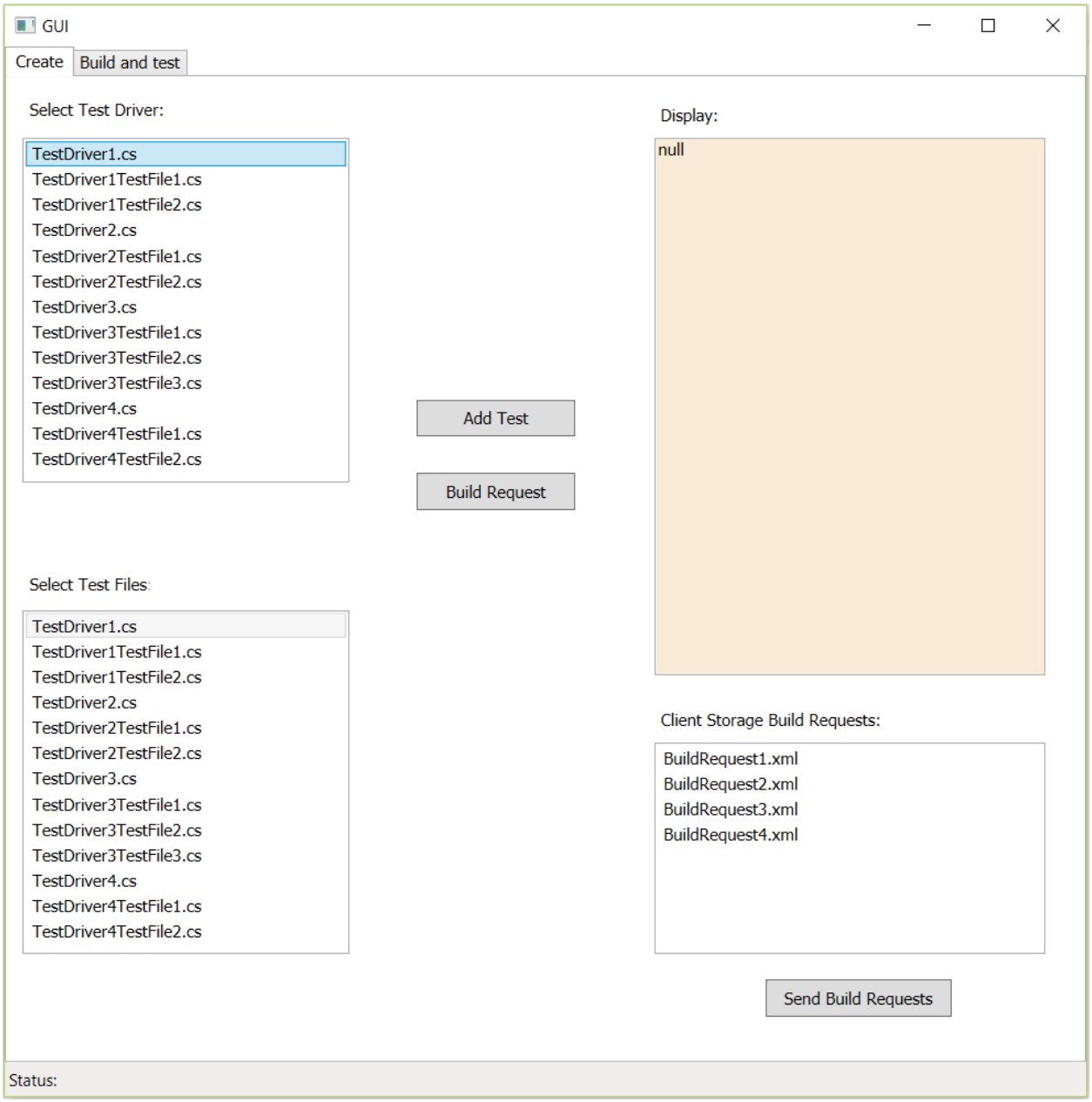

Fig 12. User Manual Diagram 1

User can select only 1 Test Driver from top left listbox and multiple Test Files from bottom left listbox. Then user can click “Add Test” button which will create a single test case as follows:

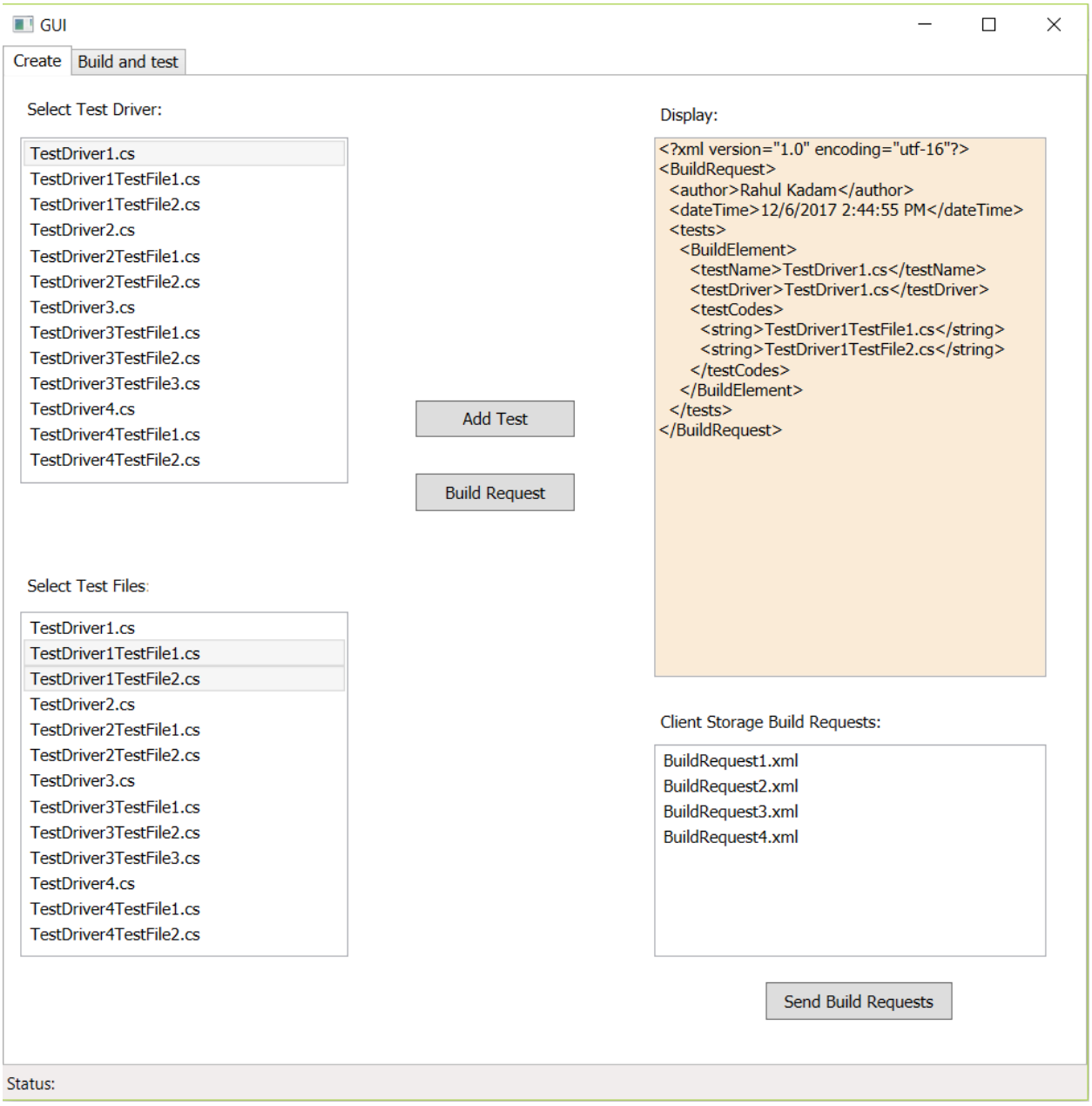

Fig 13. User Manual Diagram 2

User can also select another set of 1 test driver and multiple test files for add to previous test shown as follows:

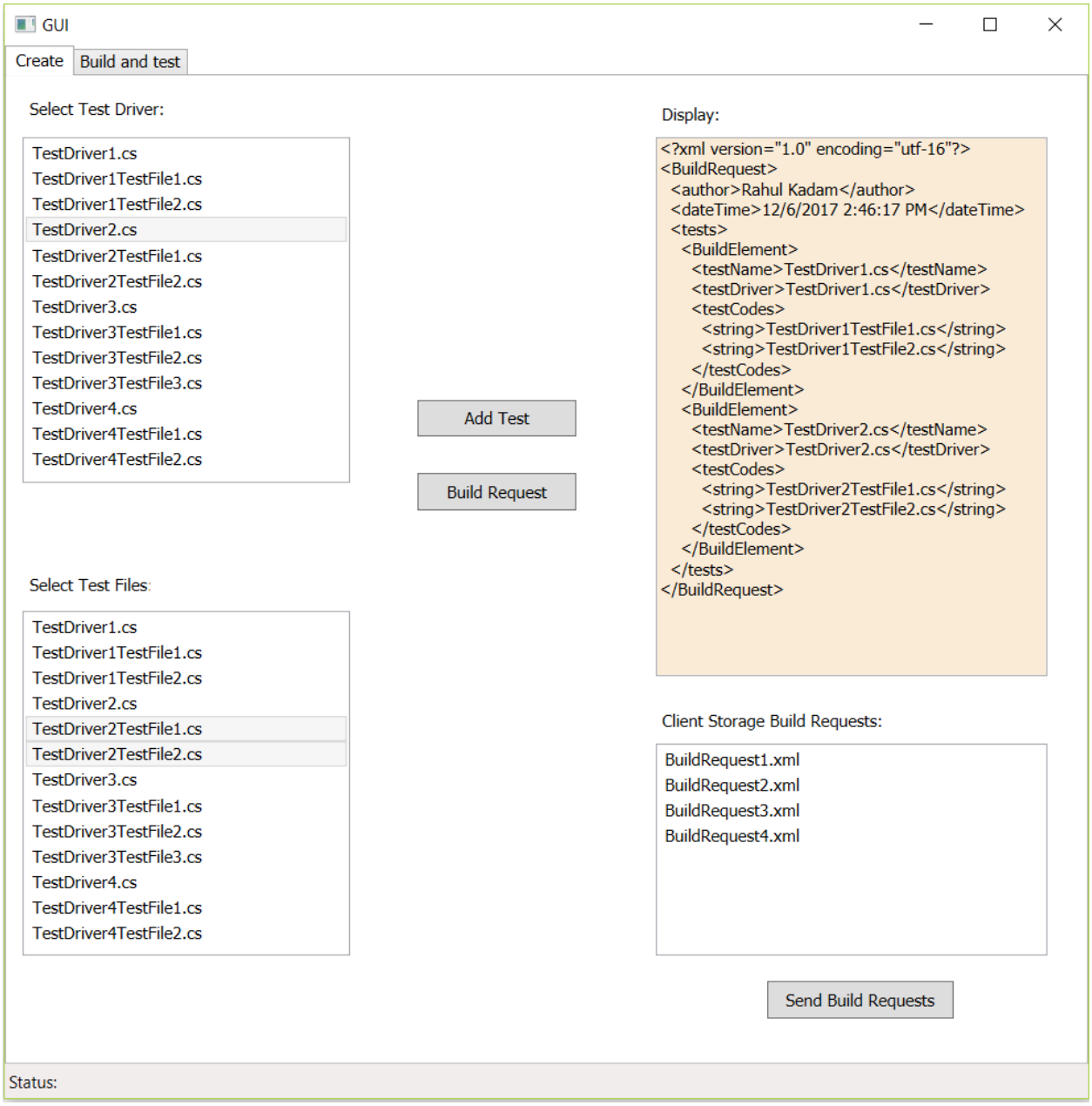

Fig 14. User Manual Diagram 3

User can then press “Build Request” button to finalize the build request and create a new build request xml file in client storage reflected below:

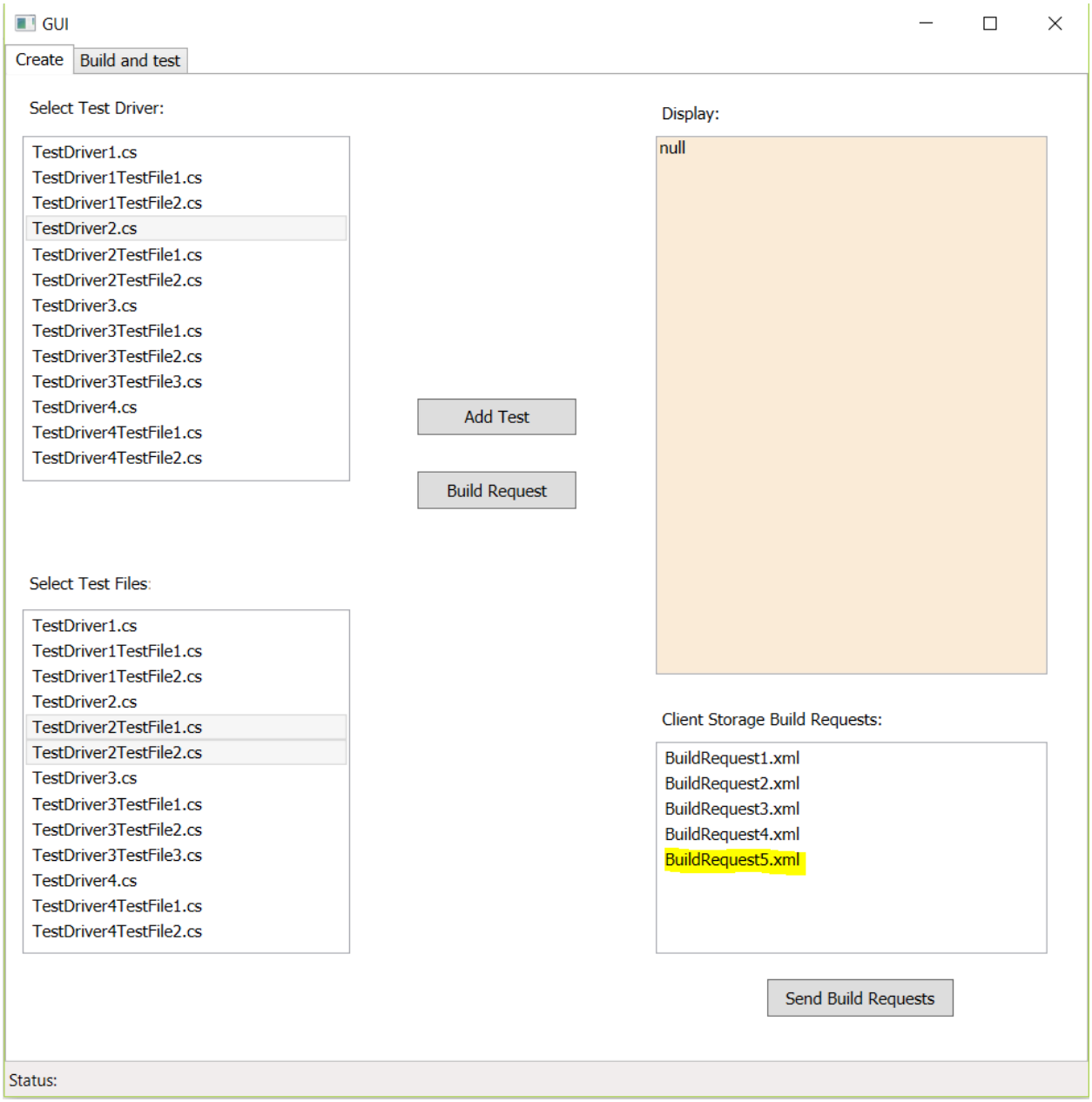

Fig 15. User Manual Diagram 4

“BuildRequest5.xml” highlighted above was created in this process. User can click this “BuildRequest5.xml” file and click “Send Build Requests” button to send this file to Repository for storage.

Tab 2 - Building/Testing Build Request, Viewing Build Logs and Test Result

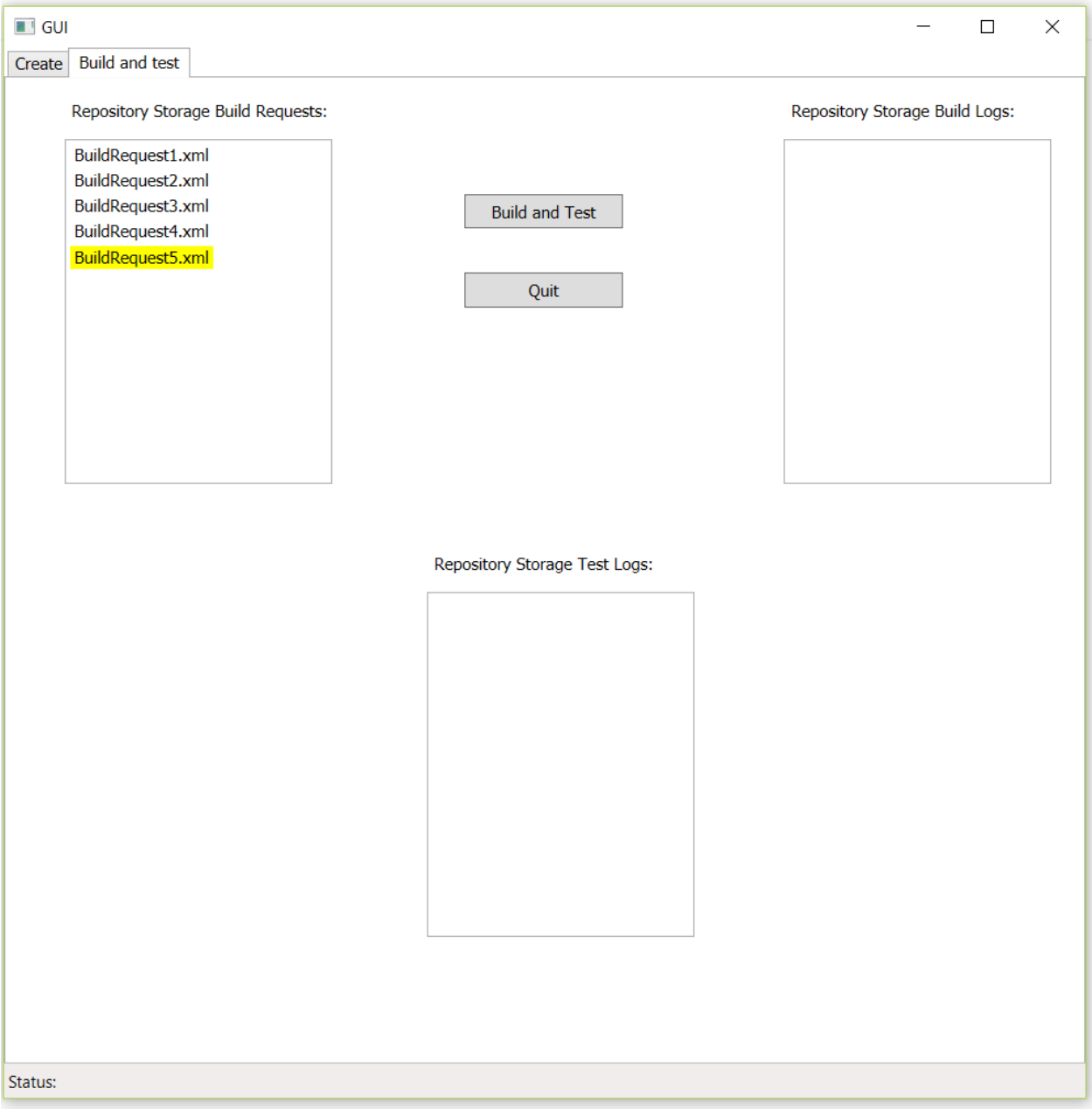

Fig 16. User Manual Diagram 5

"BuildRequest5.xml" highlighted above was sent from Client Storage to Repository Storage previously.

User can select multiple Build Request in top left listbox and press “Build and Test” button to start building and testing process.

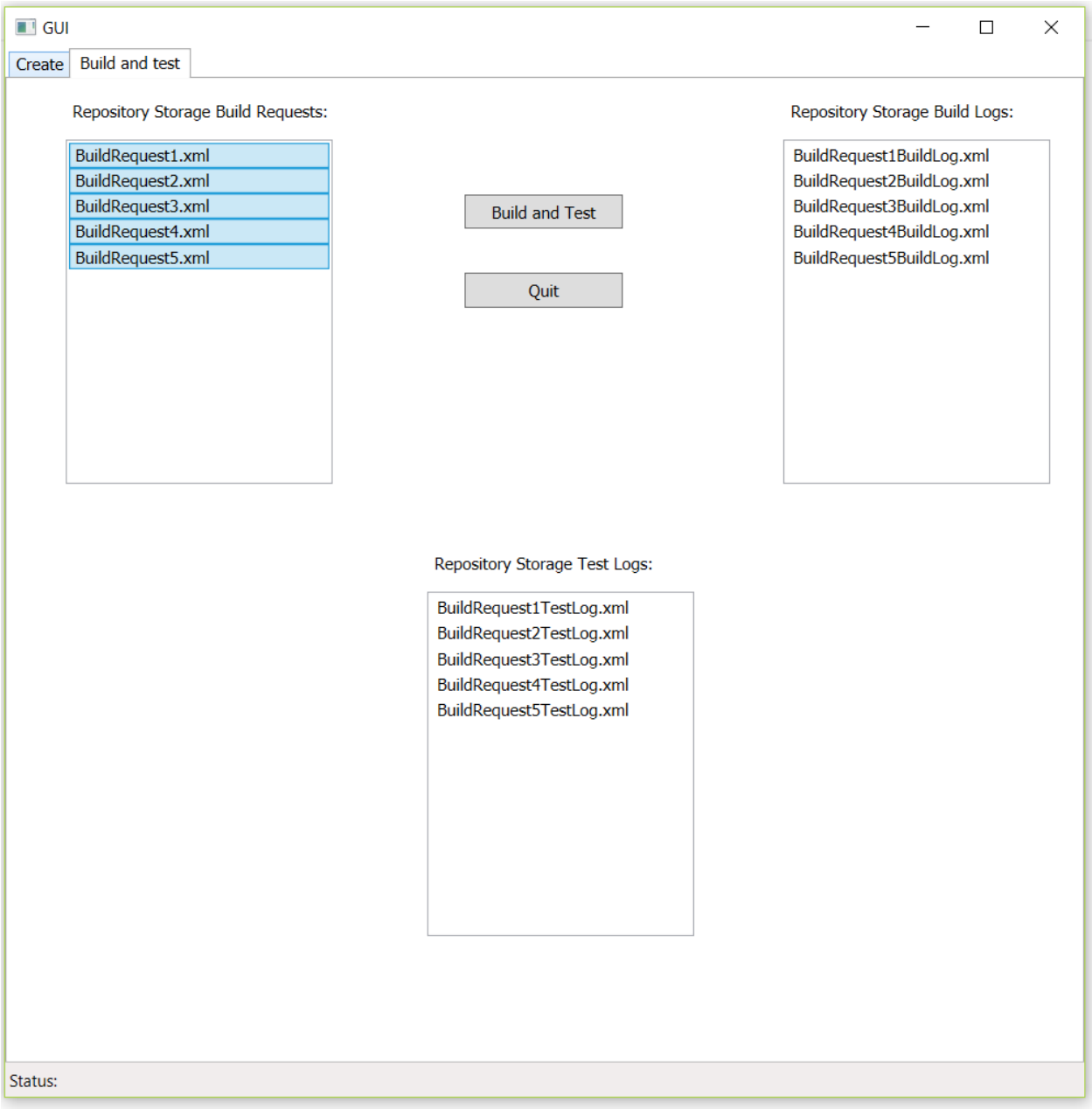

Fig 17. User Manual Diagram 6

Here all 5 build requests were selected for build and test and ask can be seem for each build request a corresponding build log and test log file was created.

User can double click on these build logs and test logs to view their contents.

Sample Build Log:

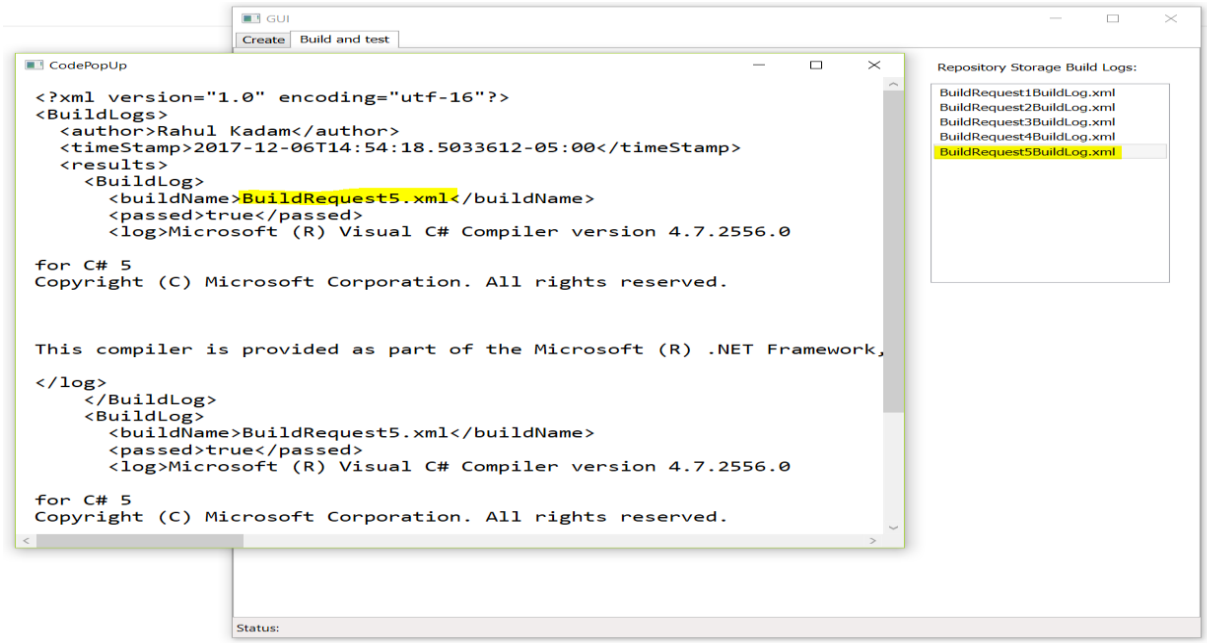

Fig 18. User Manual Diagram 7

Sample Test Log:

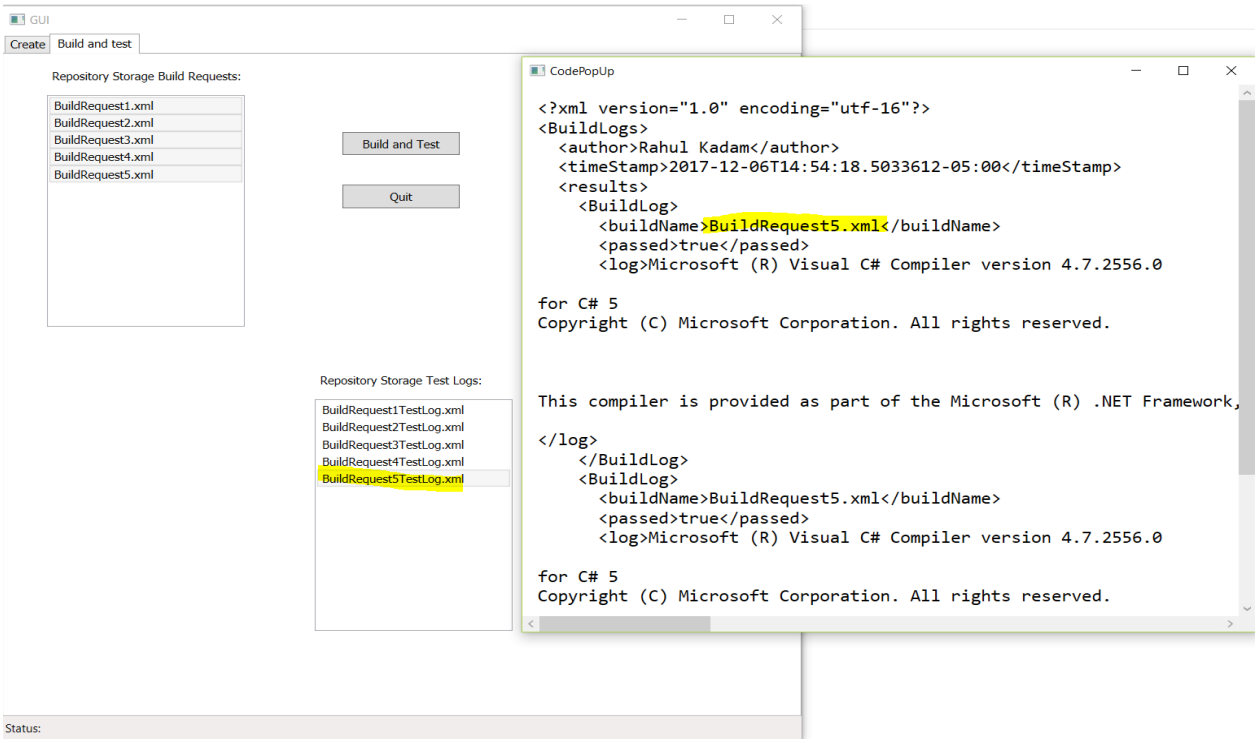

Fig 19. User Manual Diagram 8

10. Conclusion

10.1 What I like?

GUI Design:

- Design is simple and precise to operate upon.
- GUI has a display section which shows the build request (xml string) while adding new tests to final Build Request.
- Using Multiple tabs to separate Creation of Build Request and Testing Build Requests and viewing Build and Test Logs.
- Automated process with only few number of Buttons, only 5 buttons used for all the requirements.
- Double Clicking any filename on GUI will Popup a window with the file contents.

Request Manager:

- Request Manager package has separate classes for Build Request, Test Request, Build Logs and Test Logs, which are all serializable to and from xml.
- Above classes consists of list of certain things with each thing described in classes such as Build Element, Test Element, Build Log and Test Log at lower level.
- Such design made creation and browsing of build/test requests and build/test logs efficient.
- Additionally, being serialized into xml string meant these could be added on body section of CommMessage and passed around the system.

10.2 What I want to improve?

There are many things to improve upon as stated in the deficiencies section, but below are some of the top things I would like to improve in next development cycle:

- Building of C++ and Java source code files.
- Test Harness with Thread Pool of Child App Domains.
- Loading EndPoint information from xml file on startup for all client and servers.

11. References

1. http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project1-F2017.htm
2. http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project2-F2017.htm
3. http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project3-F2017.htm
4. http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project4F2017.htm