**Remote Build Server**

**Operational Concept Document (OCD)**

**Zheng Zhan (SU ID:825530128)**

**Instructor- Dr. Jim Fawcett**

**CSE-681 SMA Project 4**

# Table of Contexts

[Table of Contexts 2](#_Toc500363519)

[1. Executive Summary 4](#_Toc500363520)

[2. Introduction 5](#_Toc500363521)

[3. Users and Uses 7](#_Toc500363522)

[3.1 Developers 7](#_Toc500363523)

[3.2 Managers 7](#_Toc500363524)

[3.3 Quality Assurance (QA) 7](#_Toc500363525)

[3.4 Instructor and Teaching Assistant 8](#_Toc500363526)

[3.5 Student in CSE681 – Software Modeling and Analysis 8](#_Toc500363527)

[4. Partitions 9](#_Toc500363528)

[4.1 Mother Builder 9](#_Toc500363529)

[4.2 Blocking Queue 9](#_Toc500363530)

[4.3 Child Builder 10](#_Toc500363531)

[4.4 File manager 10](#_Toc500363532)

[4.5 CommMessage 10](#_Toc500363533)

[4.6 Comm 11](#_Toc500363534)

[4.7 Sender 11](#_Toc500363535)

[4.8 Receiver 11](#_Toc500363536)

[4.9 XML Parser 11](#_Toc500363537)

[4.10 Client 12](#_Toc500363538)

[4.11 Repository 12](#_Toc500363539)

[4.12 TestHarness 12](#_Toc500363540)

[4.13 Storage Mgr 13](#_Toc500363541)

[4.14 MainWindow 13](#_Toc500363542)

[4.15 RequestBuilder 13](#_Toc500363543)

[4.16 AppDomain Mgr 13](#_Toc500363544)

[4.17 C# Tester 13](#_Toc500363545)

[5. Tasks and Activities 15](#_Toc500363546)

[Activity process 17](#_Toc500363547)

[6. Critical Issues 20](#_Toc500363548)

[6.1 Ease of Use 20](#_Toc500363549)

[6.2 Message Structure 20](#_Toc500363550)

[6.3 Exception Handling 20](#_Toc500363551)

[6.4 Synchronization 20](#_Toc500363552)

[6.5 Unreadable Input 20](#_Toc500363553)

[6.6 Deadlock Situations 20](#_Toc500363554)

[6.7 High Volume of BuildRequests 21](#_Toc500363555)

[7. Conclusions 22](#_Toc500363556)

[Reference 23](#_Toc500363557)

[Appendix 24](#_Toc500363558)

[1. Changes To The Core Concept As Design Evolved 24](#_Toc500363559)

[Process Pool 24](#_Toc500363560)

[More specific details of other modules such as TestHarness Client Repository 24](#_Toc500363561)

[Packages of Build Server change 24](#_Toc500363562)

[Communication Channel (using WCF) 24](#_Toc500363563)

[Didn’t think about a single message structure for the whole server 24](#_Toc500363564)

[Detailed GUI 24](#_Toc500363565)

[2. Deficiencies 24](#_Toc500363566)

[Without Process pool 24](#_Toc500363567)

[Can’t build source code using more than one language 24](#_Toc500363568)

[Unclear GUI design 24](#_Toc500363569)

[Didn’t have a well-designed message structure 25](#_Toc500363570)

[Without Communication Channel Using WCF 25](#_Toc500363571)

# Executive Summary

Nowadays, software projects are much larger than before, even a small project would range from 10000 source lines of code. If every developer need to manage their own code and keep building and testing, project cost budgeting will be a huge burden. Furthermore, as project scale grow larger and larger, communication between developers is also a huge problem, because software projects are often too large for any one person to understand completely. Therefore, a federated server for group work is necessary.

Our main goal is finish a Remote Build Server. The Remote Build Server is an automated tool, capable of building C# libraries, using a process pool to conduct multiple builds in parallel. Then provide test libraries which are built into Dynamic Link Library for test process. So, it retrieves files and help user run test, this is the core of the whole automated system.

The Build Server will function as one of the principle components of a Software Development Environment Federation, the others being Repository, TestHarness, and Federation Client. Build server allows the automation of simple, repeatable tasks. We want to just send a BuildRequest to accomplish the whole test and get feedback. So, the key functions of build server are:

* using a process pool to conduct multiple builds in parallel
* creating test libraries automatedly
* communication with other modules like mock Repository, mock Client, and mock TestHarness

The primary users for this system are the developers, instructors, teaching assistants, QAs, managers and Student in CSE681 – Software Modeling and Analysis.

A few critical issues shall be considered while developing this product:

* Ease of Use
* Message Structure
* Exception handling
* Synchronization
* Unreadable input
* Deadlock
* High Volume of BuildRequests

# Introduction

To successfully implement big systems, we need to partition code into relatively small parts that are easy to understand and easy to test. This is important because we need to thoroughly test each of the parts before inserting them into the project's software baseline. But if every developer need to manage their own code and keep building and testing, project cost budgeting will be a huge burden. So, we need a Software Collaboration Federation to help us improve the productivity.

And software projects today are too large for anyone to understand it, and developers may use different programming language. So, we can’t ask QAs to understand every line of code and make test. Furthermore, as project scale grow larger and larger, communication between developers, manager, and QA will waste too much time.

Therefore, our new Software Collaboration Federation should be able to ease the work of users, the core of the federation is build server, an automated tool, which is our main work.

**A Process Pool:** The build server may have very heavy workloads 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 will use a "Process Pool". That is, a limited set of processes spawned at startup. The build server provides a queue of build requests, and each pooled process retrieves a request, processes it, sends the build log and, if successful, libraries to the test harness, then retrieves another request.

Malformed code may cause one of the processes to crash, perhaps by a circular set of C++ #include statements which overflow the process stack. This however, won't stop the Builder, which simply creates a new process replacement, and reports the build error to the repository. Note that the process pools will need to communicate with the mother Builder process.

Each pooled process has the functionality of the Core Builder.

**A Blocking Queue:** Using Blocking Queue is definitely a good way to keep synchronization and control work load. After receiving BuildRequest from repository, it will wait for ready message from Child Builder to match. The more specified details are placed in the part 4.2.

**Parsing BuildRequest:** After client send test request, build server can get information for building and testing from XML files, such as which test drive do we need and which files we want to test. And because users may use different programming language, the product we design should have the ability to choose correct tool chain for different programming language.

**Automatedly build test library:** After retrieving those files we need, build server should have the ability to build test drive and source lines of code into test libraries.

Besides, build server can play a role of communication with other modules. So, there should be an application not only can retrieve source codes and test drives from repository, but also can command test harness run test. Build server can also create build logs of building process.

The build process begins with reading input as XML files and ending in sending test files to test harness. It’s the place where the code for testing is built into libraries. Test requests and test libraries are then sent to the Test Harness.

Build server’s key functions are as below:

1. It provides logging facilities to create log and send it to repository.
2. Can parse BuildRequest to get information for testing.
3. Building source code into test libraries.
4. Can post file to TestHarness and send command to run test.
5. Play a role of communication with other modules of the system, smooth integration of these modules.

Other packages of function that build server should have:

1. It should have two Blocking Queue for input message to manage high incoming load.
2. It should have a friend Simple and friendly Graphical User Interface (GUI).
3. It should have an exception monitor to kill some process may cause crush and create logs for unhandled exceptions.
4. It should notify user when BuildRequest is incorrect.
5. It should have a Process pool to help solve high volume of build requests.
6. Building source code using more than one language

# Users and Uses

One of the basic thing of design is to think about use cases. Without users, the program is useless, and all the effort turn into a waste of time. Therefore, functions of the product must satisfy some specify requirements of different users to help them with their work. And ease of use is also a key point to attract more users. This section will discuss use cases and how users’ requirements impact on design.

## 3.1 Developers

The developers can use the build server to automate their building process and test process, and generate logs of the code they have developed. The developers can send BuildRequest through an XML file to the build server. The build server will build test libraries for TestHarness. Then uses TestHarness to verify code as part of check-in process and defect analysis.

**IMPACT ON DESIGN:**

First, a simple and friendly user interface is necessary, because this product aims to free users from repeatable tasks and make their developing process easier. This work is in the main flow of software development and it is very important to make it as easy to use and productive as is practical. We do this by focusing on the Client interactions with a well-designed GUI, by making the build server as efficient as we can, and by automating generation of messages by selecting items on the client GUI[[1]](#endnote-1)[1].

Second, exception handling ability is needed, we want the whole system stable.

Last but not the least, build process and test process should create detailed logs automatedly. And the product should allow them to submit all these codes and send BuildRequests before going off work and check these logs on next day morning. Project 4 implements the GUI and also the remote access facilities which would be very useful to the developer.

## 3.2 Managers

Managers will need summary level information about testing. Besides, managers can constantly monitor the progress by the submit date in logs to track their activities and make schedule.

**IMPACT ON DESIGN:**

It is important to build in easy to use query mechanisms that can select results to summarize testing and to build effective displays of that information. And build process and test process create enough detailed logs automatedly.

## 3.3 Quality Assurance (QA)

The QA’s job is to insure the product should be suitable for the intended purpose and mistakes should be eliminated. But for such a big project, they may need to keep building and run thousands of tests to reach that purpose. Therefore, a build server is necessary, they can just ask developers to submit their code to repository and then they send BuildRequest to test the specified function of the product. The automated process thus saves a lot of time. QA personnel focus on the testing process in preparation for acceptance testing and on maintaining code quality.

**IMPACT ON DESIGN:**

QA will be submitting BuildRequest messages for large bodies of code. It is important that the collection of dependent code packages be automated and that the build server operates as efficiently as possible. Test log should tell every detail, and provide identification of requirements, show how many of them have been achieved.

## 3.4 Instructor and Teaching Assistant

The instructors and TA can use this product to examine the system and check if all the requirements are met or not. They just need students submit their code and let them to test and grade.

**IMPACT ON DESIGN:**

The ease of use is primary concern, the goal is to help instructors and TA deal with many similar homework. They can use the project to evaluate student performance, e.g., check requirements, evaluate structure and design, and analyze code metrics.

## 3.5 Student in CSE681 – Software Modeling and Analysis

Uses the project to learn how to structure complex code, handle threading problems, and use the WCF framework to pass messages and WPF framework to develop a GUI.

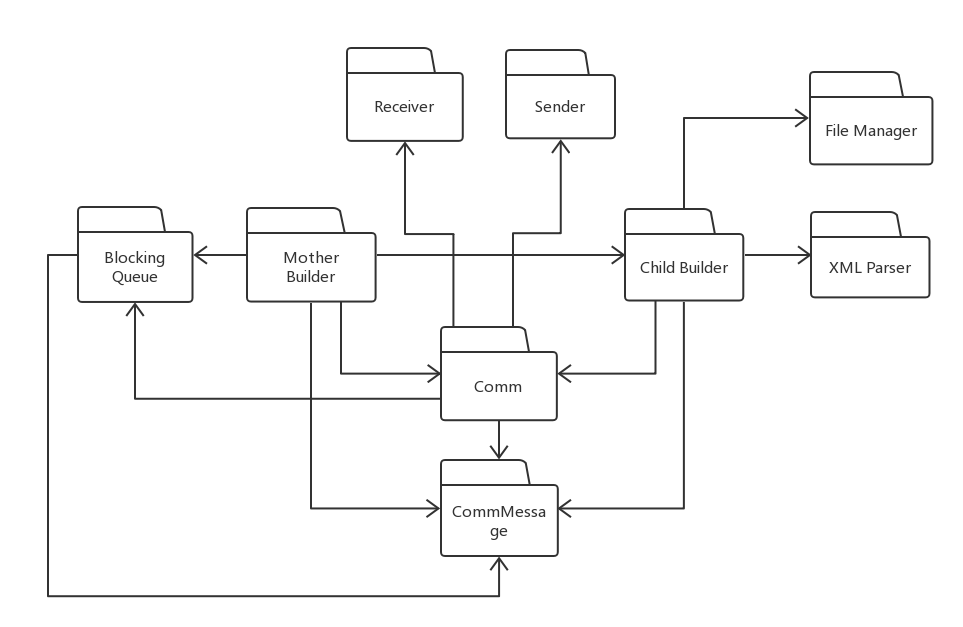
**IMPACT ON DESIGN:**

Will provide a console along with the GUI window and in the build server to demonstrate requirements. Will design a message catalog with the smallest number of messages needed to meet all the requirements. Will structure the design based on demonstration code provided in Project4HelpF17 (if they want to finish on time).

# **Partitions**

Structure of Build Server is a critical part of developing the operating concept on this system.

The Remote Build Server is divided into various packages which perform a set of functions and interact among each other. The package diagram for our system is illustrated below:



*Figure 4.1. Build server package diagram*

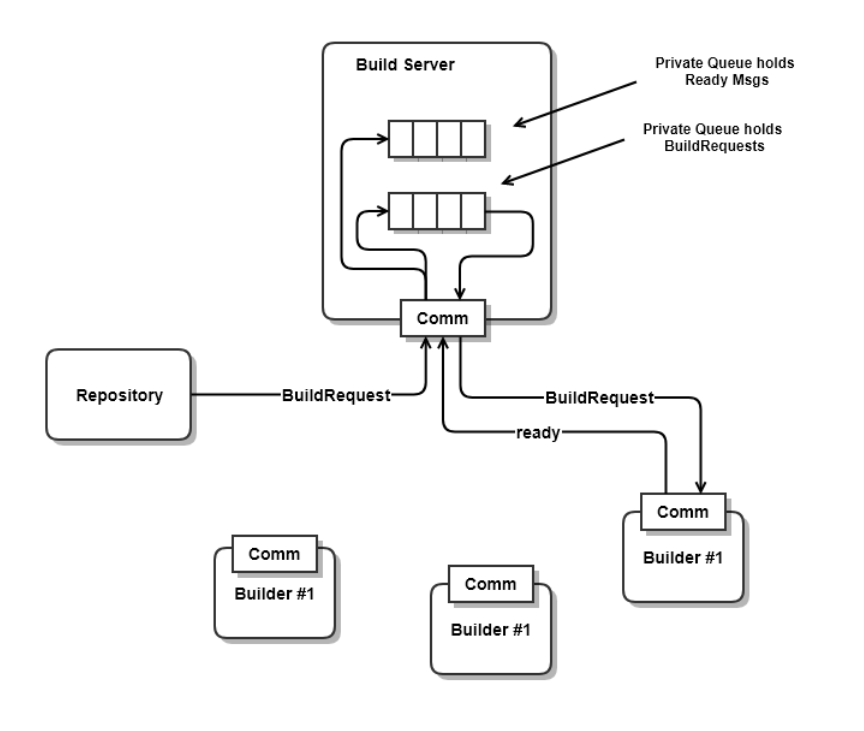
The basic functionalities of all the packages are described below:

## 4.1 Mother Builder

Mother Builder is the primary package of the whole system, it responsible for the main control flow of the build server, create Child Builder. It also commands test harness to execute test. It uses Comm package to communicate with Mock repository, Child Builder and TestHarness. It creates two Blocking queues for build request message from Mock repository and ready message from Child Builders. Match a pair and send build information to Child Process.

## 4.2 Blocking Queue

Using Blocking Queue is definitely a good way to keep synchronization and control work load. After receiving BuildRequest from repository, it will wait for ready message from Child Builder to match.



*Figure 4.2. diagram of Mother Builder using Blocking Queue*

For example, Child Builder sends ready message to Mother Builder, and repository sends a BuildRequest to Mother Builder, they are pushed into two separate queues, and when both queue are not empty, deQueue both queue for one message and send the BuildRequest to the Child Builder whose ready message was deQueue by Mother Builder.

## 4.3 Child Builder

Child Builder is created by Mother Builder and it creates temp subdirectory for storing \*.dll files and build logs. Its main work is to build Dynamic Link Library and send Dynamic Link Library and TestRequest to TestHarness.

## 4.4 File manager

File manager create a temporary directory to manage test libraries and build logs. It has a temporary storage and can manage files.

## 4.5 CommMessage

CommMessage is a package which define message flow between each module.

Here is the well-formatted list of messages for Project #4. It includes name, source and destination, purpose, and contents and every message needed for that project.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Source | Destination | Purpose | Contents |
| FileRequest | Client | Repository | Get file list | Command |
| FileList | Repository | Client | Return file list | File list |
| BuildRequest | Client | Repository | Send XML string | BuildRequest |
| SendBuildRequest | Client | Repository | Command | Command |
| BuildRequest | Repository | BuildServer | Send XML string | BuildRequest |
| BuildRequest | BuildServer | ChildBuilder | Send XML string | BuildRequest |
| Ready | ChildBuilder | BuildServer | Notification | Ready status |
| FileRequest | ChildBuilder | Repository | Get file | File name |
| File | Repository | ChildBuilder | Send file | File contents |
| BuildLog | ChildBuilder | Repository | Send build log | Log string |
| TestRequest | ChildBuilder | TestHarness | Send XML string | TestRequest |
| FileRequest | TestHarness | ChildBuilder | Get file | File name |
| File | ChildBuilder | TestHarness | Send file | File contents |
| TestResult | TestHarness | Client | Notification | Test status |
| TestLog | TestHarness | Repository | Send test log | Log string |
| Complete | TestHarness | ChildBuilder | Notification | Complete status |

*Table 4.1. list of messages for Project #4*

For example, Client sends FileRequest to Repository for getting file list. The content is stored in msg.command. When Repository get the command, it start getting file list in its directory and those files’ name are stored in msg.argument which is a List<Argument>. And then Repository send the message back to Client, so Client can view Repository now.

## 4.6 Comm

Each module should have a WCF Communication channel to communicate with each other. Comm package includes sender and receiver class to receiver and send message. Sender package contains postfile() method and receiver package contains openFileForWrite() method for file transfer. Comm is used for sending and receiving messages from above table and for files.

## 4.7 Sender

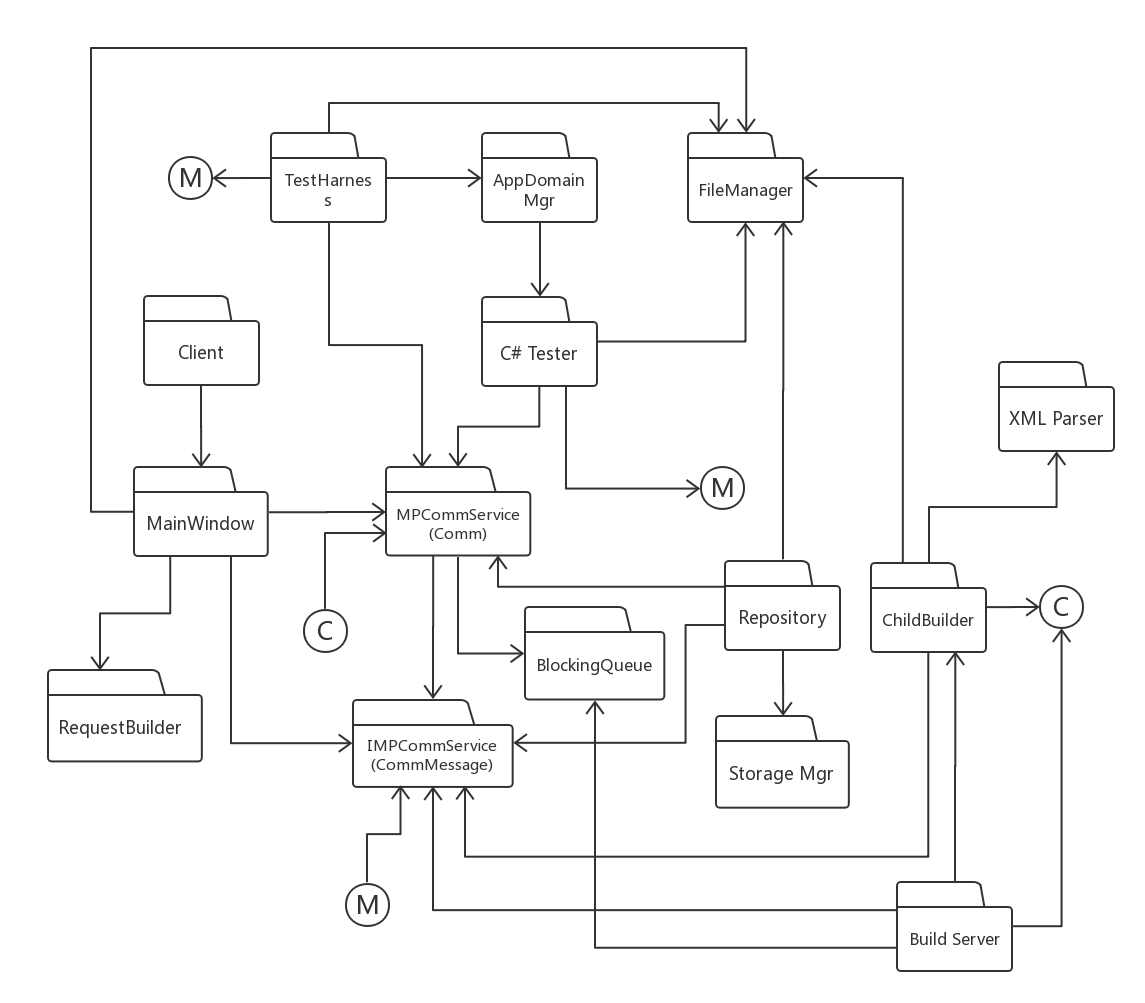
It monitors the whole system, if anything goes out of control, it can kill the process and notify users. Sender package contains postfile() method.

## 4.8 Receiver

Display is used to receive message from XML parse and exception monitor and display notifications to console. Receiver package contains openFileForWrite() method.

## 4.9 XML Parser

XML Parser is a tool to transform XML files into some readable files for build server executive. First, it can get test information from XML files, such as test drives and test files that we need. Second, it can choose correct tool chain for build process. If encounter incorrectly formatted test requests, XML Parser should notify the users.



*Figure 4.3. diagram of Project #4*

## 4.10 Client

The user's primary interface into the Federation. Client is responsible to submit source code to repository and send BuildRequest to Repository, it holds the starter of the whole automation process. Besides, it will receive notification from build server and Test Harness no matter the process succeeds or fails. It uses WPF.

## 4.11 Repository

A storage for logs, codes, executable files, any file that flow between others can be stored in here. Mock Repository relies on its File Manager to manage all its files. There may be some files missing.

## 4.12 TestHarness

Runs tests, concurrently for multiple users, based on test requests and libraries sent from the Build Server. Clients will checkin, to the Repository, code for testing, along with one or more test requests. The repository sends code and requests to the Build Server, where the code is built into libraries and the test requests and libraries are then sent to the Test Harness. The Test Harness executes tests, logs results, and submits results to the Repository. It also notifies the author of the tests of the results.

## 4.13 Storage Mgr

Storage Mgr manage all code and documents for the current baseline, along with their dependency relationships. And can get these information for repository to send to Client.

## 4.14 MainWindow

MainWindow package can be shown as the window we see directly. All our operations rely on this window.

## 4.15 RequestBuilder

RequestBuilder is a class which can use the filename that Users choose to build a XML file. The structure may look like below. It also provides the capability of repeating the process that add other test libraries to the build request structure.

<?xml version="1.0" encoding="utf-8" standalone="yes"?>

<BuildRequest>

<author>Zheng Zhan</author>

<test name="test1">

<library>plus.csproj</library>

<library>td.cs</library>

<library>testcode.cs</library>

</test>

</BuildRequest>

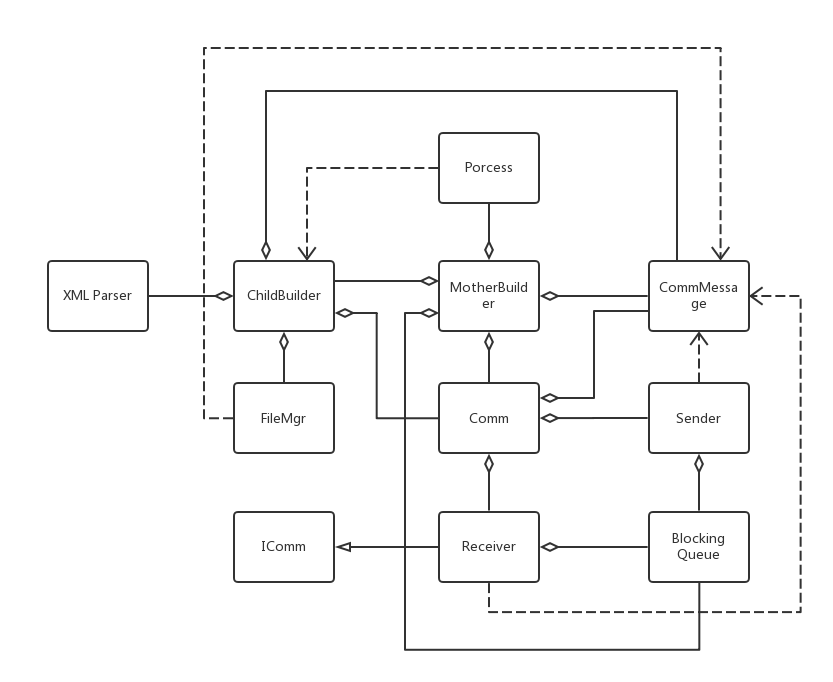
## 4.16 AppDomain Mgr

This package provides operations to create a child Application Domain, load libraries into it, and run tests on all loaded libraries that support the ITest interface.

In order to load libraries without requiring the Tester to bind to the types they declare, a Loader library is defined that is loaded into the child domain, and loads each of the test libraries from within the child.

## 4.17 C# Tester

C# Tester is



*Figure 4.4. class diagram of process pool*

- MotherBuilder manages ChildBuilders, passing them BuildRequests when ready

- ChildBuilder loads files, matching BuildRequest, from Repository and builds them into libraries and sends them to the TestHarness

- FileMgr is used to create temp directory, find the libraries for Comm.sendFile, and delete directory.

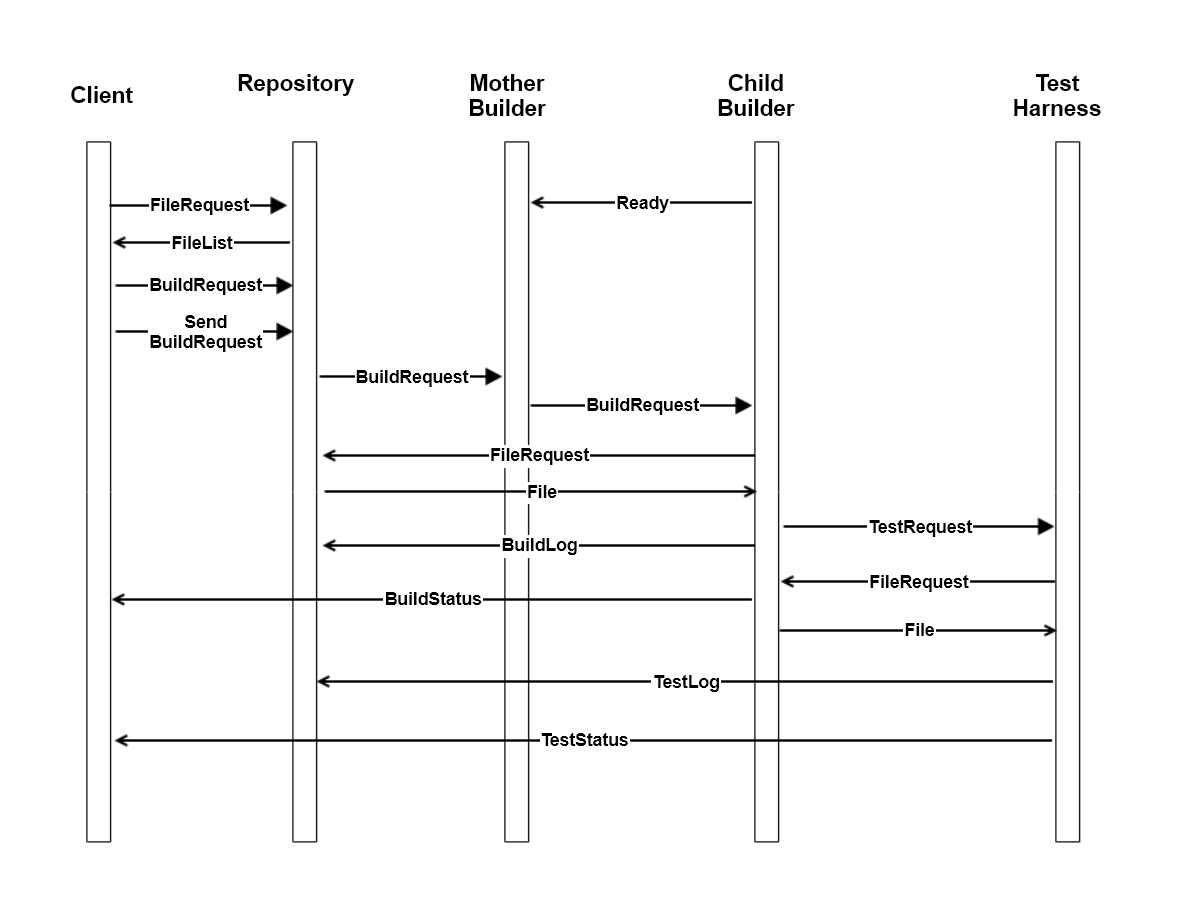
- System.Diagnostics.Process is used to support starting ChildBuilders and tracking their exit events.

- Comm and CommMessage support communication between MotherBuilder, ChildBuilders, Repo, and TestHarness

- XMLParser parses BuildRequest message for source code file names

# Tasks and Activities

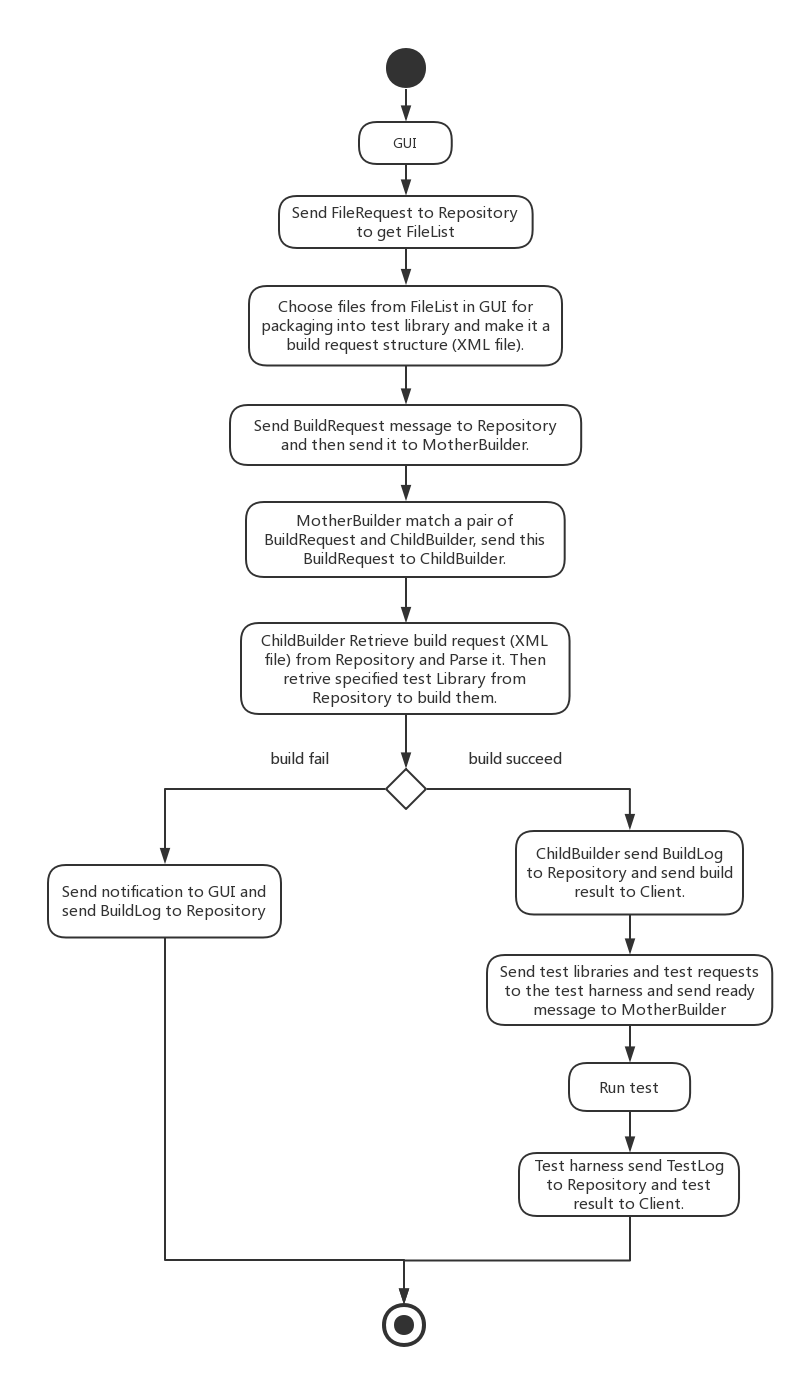
Here is the sequence diagram of Project#4. It shows the activities between each module.



*Figure 5.1. sequence diagram*

* **Client** send FileRequest message to **Repository**.
* **Repository** send FileList back to **Client**.
* **Client** choose files from FileList and make BuildRequest (XML file) in **Repository**.
* **Client** send BuildRequest message to **Repository**.
* **Repository** send BuildRequest message to **MotherBuilder**.
* **ChildBuilder** send ready message to **MotherBuilder**.
* With a ready **ChildBuilder** and a BuilderRequest, **MotherBuilder** send BuildRequest message to **ChildBuilder**. And **Repository** send BuildRequest (XML file) to **ChildBuilder**.
* **ChildBuilder** parse the BuildRequest (XML file) and send FileRequest to **Repository**.
* **Repository** send files back to **ChildBuilder**.
* **ChildBuilder** build Dynamic Link Library (\*.dll file). It also sends BuildLog to **Repository** and TestRequest message and Dynamic Link Library (\*.dll file) to **TestHarness**. At last, **ChildBuilder** send ready message to **MotherBuilder**.
* **TestHarness** do the test and send TestLog to **Repository** and test result to **Client**.

The Activity Diagram shows the processing of a single test request. Each client request will be similarly handled.

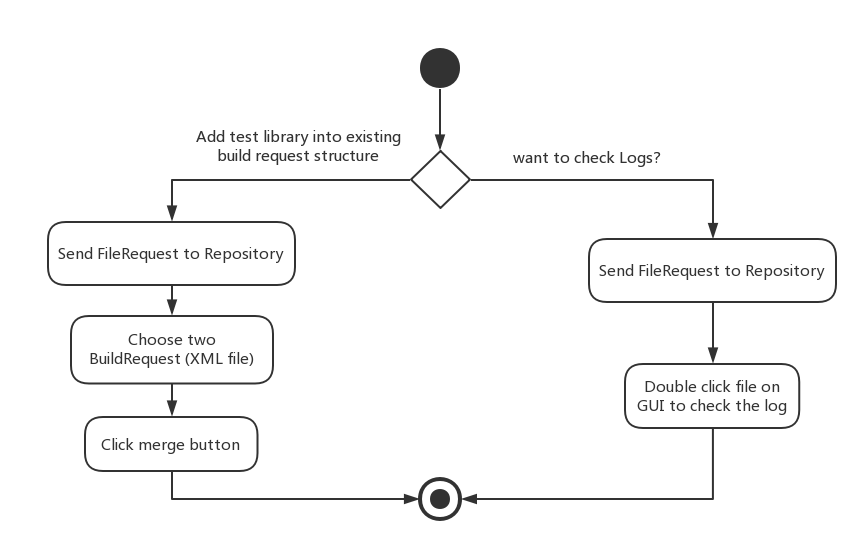


*Figure 5.2. Activity diagram for the build server system*

**Client:** The user's primary interface into the Federation. Client is responsible to submit source code to repository and send BuildRequest to Repository, it holds the starter of the whole automation process. Besides, it will receive notification from build server and Test Harness no matter the process succeeds or fails. It uses WPF.

**Mock Repository:** A storage for logs, codes, executable files, any file that flow between others can be stored in here. Mock Repository relies on its File Manager to manage all its files. There may be some files missing.

**Test Harness:** Runs tests, concurrently for multiple users, based on test requests and libraries sent from the Build Server. Clients will checkin, to the Repository, code for testing, along with one or more test requests. The repository sends code and requests to the Build Server, where the code is built into libraries and the test requests and libraries are then sent to the Test Harness. The Test Harness executes tests, logs results, and submits results to the Repository. It also notifies the author of the tests of the results.

****

*Figure 5.3. Activity diagram for other functions*

## Activity process

The Client have three main functions for users.

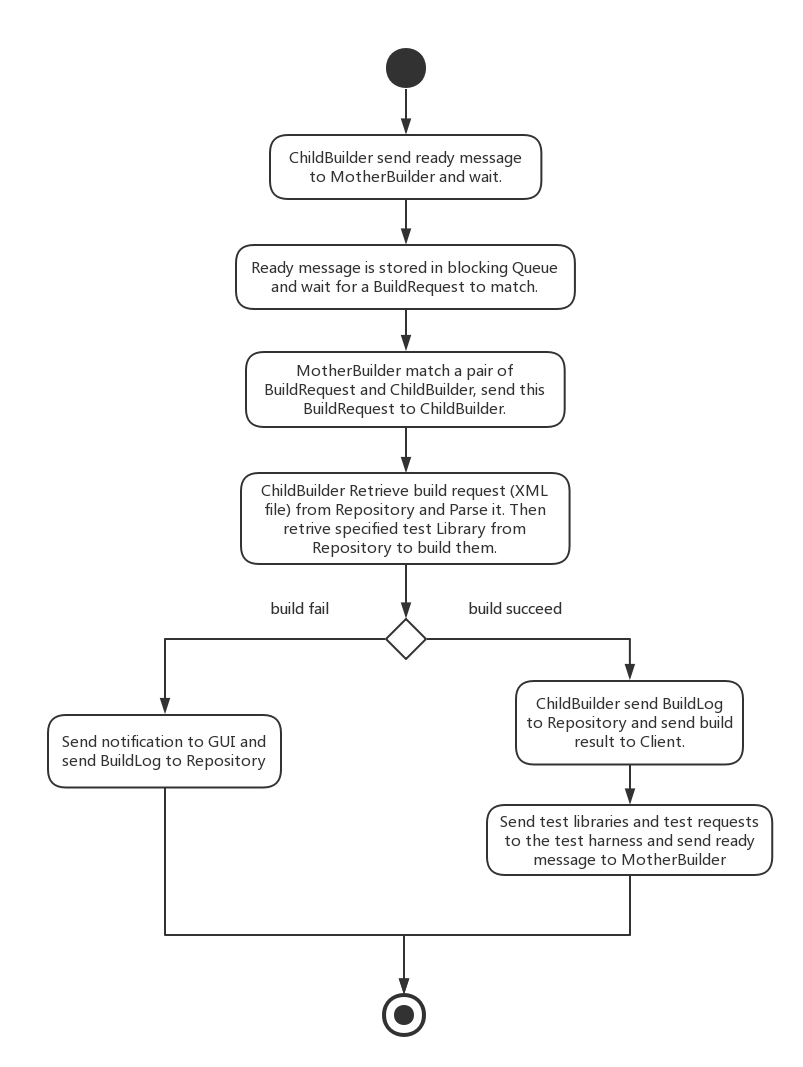
1. **Start a new test request:**
2. Client send FileRequest message to Repository to get FileList.
3. Client choose files from FileList and make BuildRequest (XML file) in Repository.
4. Client send BuildRequest message to Repository. Repository send BuildRequest message to MotherBuilder.
5. With a ready ChildBuilder and a BuilderRequest, MotherBuilder send BuildRequest message to ChildBuilder. And Repository send BuildRequest (XML file) to ChildBuilder.
6. ChildBuilder parse the BuildRequest (XML file) and send FileRequest message Repository. Repository send specified test library back to ChildBuilder.
7. ChildBuilder build Dynamic Link Library (dll file), sends BuildLog to Repository and build result to Client. If Build success, go to next step, otherwise, wait another BuildRequest from Client.
8. ChildBuilder TestRequest message and Dynamic Link Library (dll file) to TestHarness. At last, ChildBuilder send ready message to MotherBuilder.
9. TestHarness do the test and send TestLog to Repository and test result to Client.
10. **Check logs:**
11. Client send FileRequest message to Repository to get FileList.
12. Click filename to check the log.
13. **Add test library into** **existing build request structure**
14. Client send FileRequest message to Repository to get FileList.
15. Choose two BuildRequest (XML file).
16. Click merge button to add test library into existing build request structure.

**Activity diagram for build server:**

The build process begins with a BuildRequest with XML files input and a ChildBuilder will send ready message and wait until next BuildRequest. It’s the place where the code for testing is built into libraries. Test requests and test libraries are then sent to the Test Harness.

**The process**

1. ChildBuilder send ready message to MotherBuilder and wait.
2. Ready message is stored in blocking Queue and wait for a BuildRequest to match.
3. With a ready ChildBuilder and a BuilderRequest, MotherBuilder send BuildRequest message to ChildBuilder. And Repository send BuildRequest (XML file) to ChildBuilder.
4. ChildBuilder parse the BuildRequest (XML file) and send FileRequest message Repository. Repository send specified test library back to ChildBuilder.
5. ChildBuilder build Dynamic Link Library (dll file), sends BuildLog to Repository and build result to Client. **If Build success, go to next step, otherwise, wait for another BuildRequest from Client.**
6. ChildBuilder TestRequest message and Dynamic Link Library (dll file) to TestHarness. At last, ChildBuilder send ready message to MotherBuilder.
7. TestHarness do the test and send TestLog to Repository and test result to Client.



*Figure 5.4. Activity diagram for the remote build server*

# Critical Issues

## 6.1 Ease of Use

Ease of use is the first thing developer should think about. Our goal is to make users’ work become easier. First, the product shouldn’t have a complicated and confusing user interface. Second, just simple and accurate input can insure the automated build and test run correctly.

**Solution:** We do this by focusing on the Client interactions with a well-designed GUI, by making the build server as efficient as we can, and by automating generation of messages by selecting items on the client GUI.

## 6.2 Message Structure

How to define a single message structure that works for all messages used in the Federation.

**Solution:** A message that contains To and From addresses, Command string or enumeration, List of strings to hold file names, and a string body to hold logs will suffice for all needed operations.

### 

## 6.3 Exception Handling

Exception handling ability is needed, the whole system should be stable. Robustness is a key consideration that every system should think about.

**Solution:** There should be an exception monitor to monitor the whole system, if anything goes out of control, it can kill the process and automatedly create a log to notify the users.

## 6.4 Synchronization

It’s basically when and where to use BlockingQueue.

**Solutions:** We can use blocking Queue in MotherBuilder to handle it.

## 6.5 Unreadable Input

XML Parse may can’t parse the test request. One possibility is that test request isn’t a XML file. Another possibility is that the XML file format is incorrectly.

**Solution:** ChildBuilder should send users to notify him let him send correct BuildRequest.

## 6.6 Deadlock Situations

Since we make use of multithreading during programming on a build server and although they provide us with better performance the create a new issue, deadlock.

**Solution:** In this issue we can make use of thread pool. These thread pools reduce the number of application thread and provide management of worker threads. They also help in decreasing the complexity of the program.

## 6.7 High Volume of BuildRequests

The system should have the ability to handle with high load. When input is too much or too often, it may cause build server executive crash down.

**Solutions:** We can use process pool to ease the problem.

# Conclusions

This OCD shows the design concept of build server, also including use cases, package diagram, responsibility, class diagram, activity diagram and process activity. These detailed descriptions help reader access the concept of build server more easily.

Build server can automate build process and make preparation for test process, which improve developer and quality assurance productivity, and instructor, TA and manager can get information more easily.

# Reference

Operational Concept Document by Dr. Fawcett

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project1-F2017.htm>

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Code/Project1HelpF2017/Tasks2.png>

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Code/Project1HelpF2017/Tasks1.png>

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project3-F2017.htm>

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project4-F2017.htm>

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/MidTerm/MTF16/>

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/MidTerm/MTF17/>

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/MidTerm/Diagrams/>

# Appendix

## 1. Changes To The Core Concept As Design Evolved

### Process Pool

I add process pool in the whole server to ease the high workload

### More specific details of other modules such as TestHarness Client Repository

The project focused on build server and don’t have enough information for other module such as TestHarnss, Client, Repository. They all just mock package, only represented some easy function.

### Packages of Build Server change

Because the design is different, this edition add Comm and ChildBuilder package.

### Communication Channel (using WCF)

The first edition design doesn’t contain Message-passing Communication Channel. In project #4, all members of the Federation use Message-Passing Communication, implemented with Windows Communication Foundation (WCF). The Process Pool members will also communicate with the mother Builder using WCF.

### Didn’t think about a single message structure for the whole server

In first edition, there is a message package but didn’t provide a message structure which can be used in the whole project. This OCD provide a message structure and you can see it in Appendix.

### Detailed GUI

There is a detailed GUI and user manual now (in appendix), the first edition just talk about the importance of a friendly GUI, don’t have even a simple design.

## 2. Deficiencies

### Without Process pool

The first edition design didn’t have a Process Pool, it can’t deal with high volume of BuildRequest. Without Process pool, the whole build server is such a mess, message could lost, user need to operate very carefully to prevent crush.

### Can’t build source code using more than one language

It can only build C# libraries, can’t build C++ and Java libraries, which make this remote build server’s application scenarios very narrow.

### Unclear GUI design

The first edition just talked about the importance of a friendly GUI, didn’t have even a simple design. There is no way that Users can know whether it’s a friendly GUI. Ease of use is the most important issue, it should be discussed and show. So, the biggest problem ease of use didn’t have a solution in the first edition OCD.

### Didn’t have a well-designed message structure

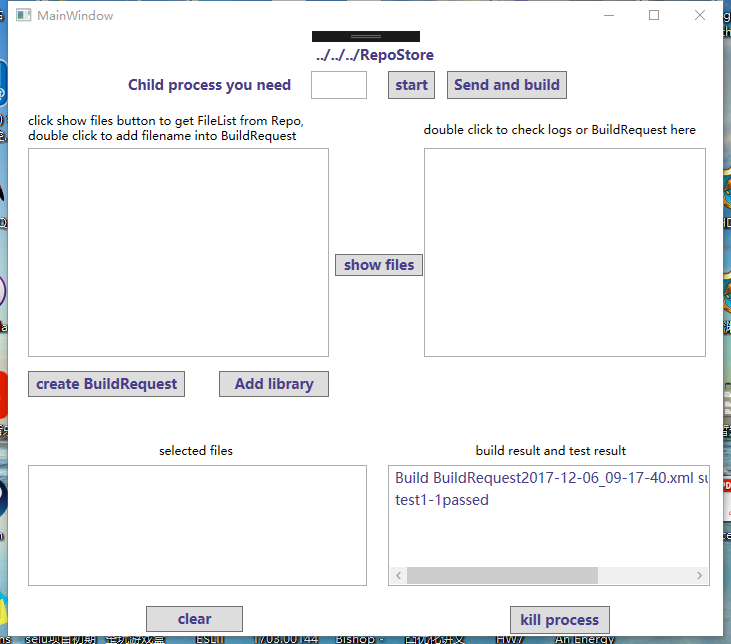
The first edition didn’t think about message structure. A awful message structure could make the whole communication system a mess. And lots of different message structure could confuse Comm channel and users.

### Without Communication Channel Using WCF

The first edition OCD didn’t have a communication channel using WCF, so it’s even not a remote build server, it’s just a design of local build server. Which made the application scenarios more narrow.

1. [1]

   ## Here is the GUI

   **“Clear” button is to clear selected files List<string>.**

   **“Add library” button is to add test library into existing BuildRequest, which is merge several xml files, Note that you must select xml files or it will crush!**

   **“Create BuildRequest” button is to create a buildrequest by selecting testcode and test drive files and csproj file for them.**

   ***\* Operations Instruction***

   ***\* ----------------------***

   ***\* 1.You need to enter a number, which is the number of child.***

   ***\* process you want.***

   ***\* 2.Click "start".***

   ***\* 3.Click "show files" to get FileList of code or logs.***

   ***\* 4.Choose file you want build into BuildRequest.***

   ***\* 5.Click "send and build".***

   ***\* 6.If you want to kill process, just click "kill process".***

   ***\****

   ***\* if you want to add test library into existing BuildRequest,***

   ***\* choose several xml files and click “Add library"***

   ***\****

   ***\* Sometimes, you may need to wait a moment for step 2 and 6.***

   ***\****

   ## Message structure

   [DataContract]

   public class CommMessage

   {

   public enum MessageType

   {

   [EnumMember]

   connect, // initial message sent on successfully connecting

   [EnumMember]

   request, // request for action from receiver

   [EnumMember]

   buildrequest, // request for action from receiver

   [EnumMember]

   testrequest, // request for action from receiver

   [EnumMember]

   reply, // response to a request

   [EnumMember]

   showfiles, // response to a request

   [EnumMember]

   closeSender, // close down client

   [EnumMember]

   closeReceiver, // close down server for graceful termination

   [EnumMember]

   close // close down server for graceful termination

   }

   /\*----< constructor requires message type >--------------------\*/

   public CommMessage(MessageType mt)

   {

   type = mt;

   }

   /\*----< data members - all serializable public properties >----\*/

   [DataMember]

   public MessageType type { get; set; } = MessageType.connect;

   [DataMember]

   public string to { get; set; }

   [DataMember]

   public string from { get; set; }

   [DataMember]

   public string author { get; set; }

   [DataMember]

   public string status { get; set; }

   [DataMember]

   public Command command { get; set; }

   [DataMember]

   public string info { get; set; }

   [DataMember]

   public List<Argument> arguments { get; set; } = new List<Argument>();

   [DataMember]

   public int threadId { get; set; } = Thread.CurrentThread.ManagedThreadId;

   [DataMember]

   public ErrorMessage errorMsg { get; set; } = "no error";

   public void show()

   {

   Console.Write("\n CommMessage:");

   Console.Write("\n MessageType : {0}", type.ToString());

   Console.Write("\n to : {0}", to);

   Console.Write("\n from : {0}", from);

   Console.Write("\n author : {0}", author);

   Console.Write("\n command : {0}", command);

   Console.Write("\n info : {0}", info);

   Console.Write("\n status : {0}", status);

   Console.Write("\n arguments :");

   if (arguments.Count > 0)

   Console.Write("\n ");

   foreach (string arg in arguments)

   Console.Write("{0} ", arg);

   Console.Write("\n ThreadId : {0}", threadId);

   Console.Write("\n errorMsg : {0}\n", errorMsg);

   }

   public CommMessage clone()

   {

   CommMessage msg = new CommMessage(MessageType.request);

   msg.type = type;

   msg.to = to;

   msg.from = from;

   msg.author = author;

   msg.command = command;

   foreach (string arg in arguments)

   msg.arguments.Add(arg);

   return msg;

   }

   } [↑](#endnote-ref-1)