**Build Server**

Operational Concept Document

CSE 681 Software Modelling and Analysis

Instructor – Jim Fawcett

Naga Rama Krishna

[nrchalam@syr.edu](mailto:nrchalam@syr.edu)

6th Dec 2017

Contents

[EXECUTIVE SUMMARY 4](#_Toc500337346)

[DEFICIENCIES IN ORIGINAL 5](#_Toc500337347)

[INTRODUCTION 6](#_Toc500337348)

[Key architecture and obligations 6](#_Toc500337349)

[Organizing Principle 7](#_Toc500337350)

[USES 8](#_Toc500337351)

[Instructor and Teaching Assistant 8](#_Toc500337352)

[Developers 8](#_Toc500337353)

[QA 8](#_Toc500337354)

[Managers 9](#_Toc500337355)

[Sales Team 9](#_Toc500337356)

[Customers 9](#_Toc500337357)

[APPLICATION ACTIVITES 10](#_Toc500337358)

[Client 10](#_Toc500337359)

[Repository 10](#_Toc500337360)

[Test Harness 10](#_Toc500337361)

[Build Server 10](#_Toc500337362)

[Failure(Versioning) 11](#_Toc500337363)

[PARTITIONS 13](#_Toc500337364)

[Client 13](#_Toc500337365)

[Mainwindow 14](#_Toc500337366)

[Repository 14](#_Toc500337367)

[ToolChainManager 14](#_Toc500337368)

[Builder 14](#_Toc500337369)

[Child Builder 14](#_Toc500337370)

[Test Harness 14](#_Toc500337371)

[IMPCommService 14](#_Toc500337372)

[MPCommService 15](#_Toc500337373)

[BlockingQueue 15](#_Toc500337374)

[FileManager 15](#_Toc500337375)

[Class Diagram 16](#_Toc500337376)

[USER INTERFACE 18](#_Toc500337377)

[CRITICAL ISSUES 19](#_Toc500337378)

[Communication 19](#_Toc500337379)

[Usability 19](#_Toc500337380)

[Availability 19](#_Toc500337381)

[Performance 20](#_Toc500337382)

[Complexity and Flexibility 20](#_Toc500337383)

[Security 20](#_Toc500337384)

[Safety 20](#_Toc500337385)

[Demonstration 21](#_Toc500337386)

[CONCLUSION 22](#_Toc500337387)

[REFERENCES 22](#_Toc500337388)

LIST OF FIGURES

Figure 1: Activity Diagram 11

Figure 2: Package Diagram 13

Figure 3: Class Diagram 16

# EXECUTIVE SUMMARY

This development will create a Build Server, capable of building C# and C++ libraries, using a process pool to conduct multiple builds in parallel. The implementation is accomplished in three stages.

The first, Project #2, implements a local Build Server that communicates with a mock Repository, mock Client, and mock TestHarness, all residing in the same process. Its purpose is 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, develops 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, completes the build server, which communicates with mock Repository, mock Client, and mock TestHarness, 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.

Critical issues include: building source code using more than one language, scaling the build process for high volume of build requests, and using a single message structure for all message conversations between clients and servers. All of these issues have viable solutions.

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. Building these other Federation parts is beyond the scope of this development.

# DEFICIENCIES IN ORIGINAL

While developing 3 projects I can see some deficiencies in my original thoughts. Important deficiencies are like main builder to handle all incoming build requests, creating multiple child builders to handle main build activity, usage of message passing communication to efficiently transfer data or files between processes.

Also, during original design I haven’t clearly distinguished between build and test requests. Also, who is going to build it. As project got evolved I distinguished build request is to have only source code file names and project name under which they belong to. And the test request to be built by child Builder after build is successful and containing list of dll or jar files it built.

I have also learned better code organizing to handle multiple build requests by having multiple Child Builders. Also, to use lambdas handle simple requests to retrieve files and directories.

Under users, initially, I thought a QA will interact with TestHarness directly. But Developer and QA will interact with the application via same GUI. A QA will select a build request by selecting one from the list of available build requests already saved.

The original activity flow is serial. It can’t handle multiple build requests coming at a time. By introducing message passing communication framework, we can now handle multiple build requests, by forwarding the request to any free Child Builders. The receiver will be receiving incoming request message and add to corresponding process queue to do processing based on the message. Once the processing is done it will reply to the requester. All these will happen seamlessly without one waiting for the other.

While developing initial document, haven’t thought about importance of Blocking queue in a multi-processed application, which will improve performance of the application greatly by handling multiple requests.

Also, the idea of a sender and a receiver for each process. Dividing communication this way will ensure a process to be responsive for every incoming message. Even the structure of message passed between different processes so that all vital information is exchanges in a seamless manner.

Communication place important role while multiple interfaces interact, so that one message won’t go to which it is not intended. Also, GUI should be responsive to multiple actions at any time. It shouldn’t be non-responsive to the user, while user is trying to do some action.

# INTRODUCTION

## Key architecture and obligations

In today’ agile world, the requirements of the software changes rapidly. At the same time, the time to deliver these rapid changes is short. This makes every team in the industry wants to design, develop, build, test and deploy the changes in agile way. Every step in this process is very important. But as the software development has to be done in a short time. We look for ways to reduce time for at least one step in the development process. Design and development are time consuming steps. As lot of brainstorming should happen for design and lot of man hours will go for development. We can hasten the process by introducing automation in build, test and deployment phases of software development.

In this project, we will be automating this continuous integration process, which includes building the code and testing the built code. Once build and test of a module is successful, we can integrate this module with other modules. Again, repeat this process of build and test on this merged module. We can repeat this process until we go up to the root of the project tree. All this can be done with minimum supervision of Developer or QA. Streamlining this process will be very helpful for faster delivery of a project.

All this process will come into picture after a developer check-in their code and a QA check-in their created test request for the code developed by a developer.

For this, we are going build a federation of servers, namely, Client, Repository, Build Server and Test harness. The Build Server will be the central theme of this project, while rest of the servers will be developed as mocks running on different processes.

Each server will be obliged to do its own assigned task. Executive Client will submit code and build request. Repository will store code and build request. Build Server will build the code. Test Harness will test the built code using test request built by Build server after build is successful. In addition, we have few more obligations as discussed below.

* This project should be able to handle the scenarios where Executive Client submits the code which is not supported. With this project, we target to build C# and Java files but it is designed in such a way that makes it easy to extend this functionality for other languages. The code submitted by the Client should be the one supported by this project. If the user submits the code other than the one not supported by our project, we should respond with a friendly message.

* Project should be able to handle improper test request. Usually it should be in proper format containing all the required information like author name and source code we are should use for build. All these information should be in a XML.
* The build or test should be in isolated process. This is so because during build or test if an exception happens it could kill whole process. So, by doing this in isolated manner we are mitigating the worst scenarios.
* We are going to demonstrate that we are meeting the project requirements by printing every status on the console and log file

## Organizing Principle

The way we are going to organize the whole project will have high impact on how we are going to extend the project in future with changing requirement. We are going to group all logically related tasks into one package. We are going to have separate packages for communication, database and logging. So that other packages will do the task they should do. We will discuss it further in Partitions section.

# USES

We will divide our application uses into 4 groups. Instructor and Teaching Assistant uses in one group. Developers use in the second group, QAs use in the third group and the rest into the final group. We will see the impact on our design in this group wise.

## Instructor and Teaching Assistant

Instructor and Teaching Assistant are the prime users of the application, initially. They would want to see whether our application have all the required functionalities set by them.

This requirement would be make us to have proper log messages for every important task. These log messages should be redirected to console for visibility.

## Developers

Developers are the main users of this application. They would be submitting their code to check whether their checked-in code successfully built after integrating with other team members code. Even, they would like to test their code remotely before releasing to QA team, by testing in Test Harness.

In a team, there can be many team members, who could be raising request to build at the same time. If all the requests are on the same code base there will not be any problem. Otherwise, server should keep the request in queue, to be picked next.

There can be a scenario where different teams asking the server to build at the same time. We should be able to handle it by creating separate process for reach request. The same applies when they are requesting to test the code.

## QA

QAs are other main users of the application. They would use the application to test the code using Test Harness. They would do this after Developer submitted the code and Build Server built the libraries successfully.

They would request Test Harness by creating and submitting a test request. This test request is a XML file. Not everyone is good in creating XML files and there will be few who like to upload their own XML file. For this we should give 2 options. One option is to give upload feature to upload XML file. Other is UI should have selection of options which will build XML automatically in background.

Another important point is that they would ask the Test Harness to test during the night. This would have an impact on our design. If any test case failed we shouldn’t stop, we should just log that information and continue with other testing. Also, we should have a service which checks whether any testing to be done.

## Managers

Managers would be interested in the status of the overall project or each stage of the project. They wouldn’t be interested in detail until something go wrong.

## Sales Team

Sales Team, like Manager, would be interested in the overall status of the project to communicate with Customers how the project is going on and when it could be delivered.

## Customers

Customers would like to have a quick glance on the overall status of the project, whenever they feel it.

All these three users will have same impact on our design, especially on UI. As they need minimal information and we have one UI for all users. By default, we will display only minimal information about build on UI. UI’s clickable features, when clicked, will display complete details like build start time, end time, some logs.

These uses will have an overall impact on our application. This impact will be like how many users can access our application at a time, how many requests our application can handle, how many builds we can do parallelly. We can handle each of these by creating separate process for every user access, every new request and every build. However, there will be a limitation to these based on our system capability like CPU, memory and other applications running in parallel.

# APPLICATION ACTIVITES

Our whole application activities will be divided among Executive(client), Repository, Build Server and Test harness. These will make federation of servers. And every server has a different role, which can be seen in the below activity diagram. Next, we will discuss every part of this activity diagram.

## Client

The client will select source code files, create a build request and save it in repository. Then it will ask the repository to trigger build by selecting specific build request. While build is in progress it can receive notifications about build status from Build Server and test results from Test Harness.

## Repository

This will send build request message to builder. After Build Server and Test Harness complete their jobs, repository will cache the build and test log.

As many clients submit code and test request at the same time, Repository should be able to handle multiple inputs at a time.

## Test Harness

Test Harness will start executing tests, on receiving test libraries and command from Build Server. During the test process, it will log all the activity. At the end, it will send result notification to client.

## Build Server

Build Server is the central activity of this project. It will build the source code files it receives and forward to Test Harness.

In detail, this Server, initially, accepts Build Request message from Repository and forward it one of the free **Child Builders**. It will parse the received request to check how each project is configured. Then it will get source code files from Repository after creating a temporary folder. Then it will parse the received source code files to decide which tool chain to use while building. Using the tool, it will attempt to build the source code files.

Based on build result it will send pass or fail notification to Executive Client. Parallelly, it will send built libraries and logs to Repository. At the same time, if build is successful, it will send Test Libraries and Test Request to Test Harness for it to test.

All this can be seen in the below activity diagram of Build Server.

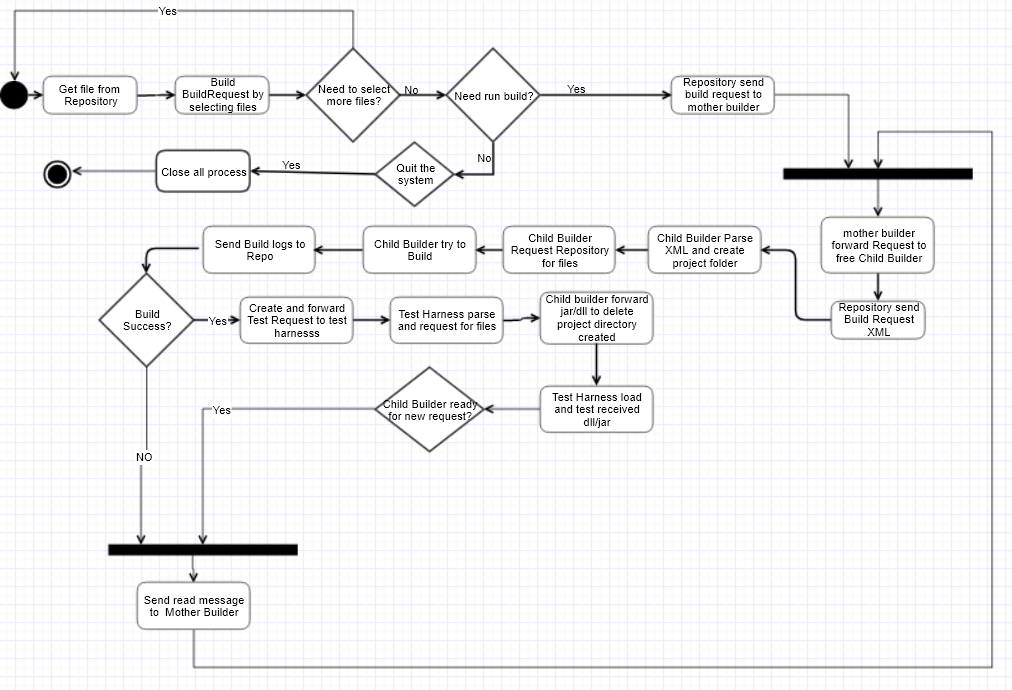


Figure 1: Activity Diagram

In this project, we are going to develop Build Server mainly and rest of the servers will be developed as mock. Basically, we are building mock servers to demonstrate functioning of Build Server. These mocks will be built in step-by-step process. In project 3 we are going to build the mocks as single process. On moving to project 4 we will have separate process for each mock.

## Failure(Versioning)

Now let’s discuss how we are going to handle failure scenario. We will discuss this in combination of versioning.

Developers and QAs will be checking-in their files and command Build Server to build and Test Harness to test the libraries generated by the Build Server. After this there will be 2 scenarios we should handle. One scenario is where Developer/QA will check-in new versions of already checked-in files. Other is a negative scenario where build or test fail. We will discuss these one after the other.

Developer/QA will be checking in many files and they could be checking in same file multiple times. We should handle the new version of the same file scenario such that when we build again we use this new version. For this we should have a serial number, which gets generated automatically and every new number should be greater than previous generated number. The new version file will replace the file in main repository and we will be moving old version to back-up location. Back-up is very important so that Developer/QA can fall back to this when new version is not meeting requirement or change in plan of delivery.

When something goes wrong with build or test, we will be informing Executive Client. The client will take some action and check-in new code or test request. Then again, we should build or test the corrected one. How we are going to do this is very important. If we are going to do complete build or test, then it would be waste of resources. We should go in this approach, when the build or test of the very first package on which other packages depend is failed. Otherwise, we should continue to build or test from the point where build or test failed earlier. For doing this we should save the build or test information in repository when build is failed and use it for the next time.

It may be difficult to implement few of the features discussed here because of time constraint and our project scope. But these are good points to be considered while implementing in real time.

# PARTITIONS

Today, most software is developed as modular. So that we can remove or enhance any specific module without impacting any other in the system. In this project too, we will develop modular approach. During modularization, we should group logically similar tasks into one module.

Following is the Package Diagram we will follow for this project and below we will discuss each package.

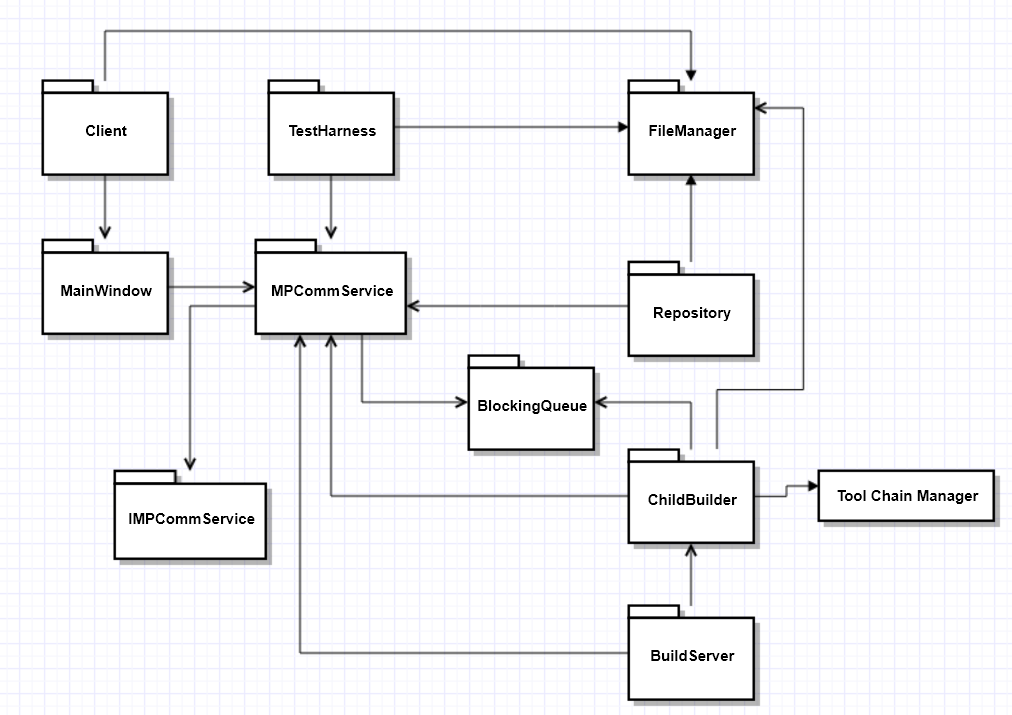


Figure 2: Package Diagram

## Client

The Client will request all source code files available in Repository to build XML files and send to Repository for later usage. It also get information about build and test status from Builder and Test Harness respectively. It can also trigger multiple Child builder via Builder.

## Mainwindow

A user going to interact with the system via this package. It will take help of Client to do all the activities like displaying files, XML, build and test status.

## Repository

The main activity of this package is to store source code and build request of a Client, libraries and build logs of Builder Package, Test logs of Test Harness Package in a file system.

Also, this package will be responsible to send list of files available to Client so that it can select files and generate Build Request. This saved request will be sent to builder to initiate build process.

## ToolChainManager

This package will be responsible to decide which tool we are going to use to generate dll files.

## Builder

Builder will be responsible to create specific number of Child Builder and assigning different port numbers to each. It will also deviate all build requests coming from Repository to free Child Builders.

## Child Builder

This will build source code and generate respective dll or jar files. During this build process it will get build request from Repository. Once build request processing is done it will request source code from Repository and once all files are copied it will do the build process. Build status will be send to Main window, logs will be send to Repository. At the end, it will generate a test request and send to Test Harness to test the built files by copying to its location.

## Test Harness

On receiving libraries and test request from Builder, this package will test the library based on test cases in the test request. After testing is done, it will send test logs to Repository to save for later use. Also, it will send success or failure status to display on Main window via Client.

## IMPCommService

This package defines the service contract interface to be used for communication between different packages.

## MPCommService

The communication between main packages like Client, Repository, Builder, Child Builder and Test Harness will happen via this package. This will be responsible in sending any message from one package to another, also to receive message.

## BlockingQueue

This package will be responsible to hold messages while communication is happening between two different packages. This is make sure that one processing happening in serial fashion.

**Logger**

This package will display log messages on consoles in a specific format.

## FileManager

This package will be responsible for all file system related activities like create or delete directory, delete file, getting of files in a directory.

## Class Diagram

All these relationship between packages can be better viewed in a class diagram as below

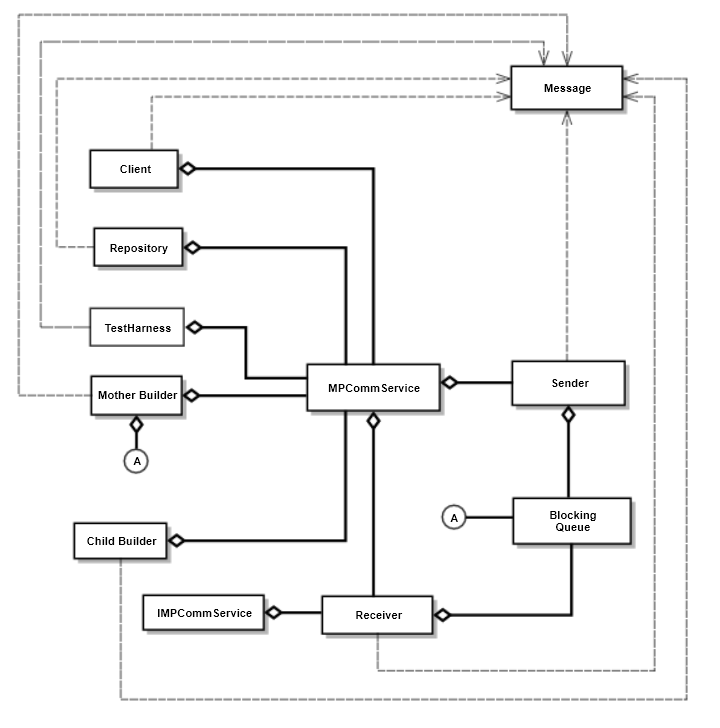


Figure 3: Class diagram

**IMPCommService** interfacewill define service contract to be implemented.

**MPCommService** class will implement the service contract interface as defined in IMPCommService interface. Based on this definition all classes will interact with each other.

**Sender** class will define methods to invoked while sending a message from one class to another class.

**Receiver** classes will define methods that would get invoked when a class receives a message from another class.

**Blocking Queue** classdefines queue to be used in a multi-threaded environment. So that no thread will waste the computing resource.

**Message** class defines the message structure to be exchanged by different classes.

**Client** class defines the user interaction activities like get list of files and folders under repository, creating build request XML, trigger build.

**Repository** class defines the storage related activities like get list of files and folders for Client class, source code files for Child Builders, stores build request, build and test logs it receives from other classes.

**Builder** class defines logic to route all incoming traffic to free Child Builder. It uses Blocking queue to maintain incoming build requests from Repository and free Child Builders.

**Child Builder** class defines methods to get build request XML and source code files from Repository, build the received code, send build logs to Repository, create test request and send to Test Harness along with successful dll or jar.

**Test Harness** class will define methods to parse test request and test the dll or jar file.

# USER INTERFACE

In any application, we can think of interface at different level. One is at module level, that is how different modules will communicate between each other so that they will together do the complete work. Other is at user level, that is how a user will use a whole application. We will discuss these one after the other.

Main modules, that is Client, Repository, Build Server and Test Harness, in our application will not interact with each other directly. They will communicate via Communication Interface provided by the Communication Package.

All modules logging or printing to the console will be done via logging interface provided by the Logger module. This will make sure that the logging is happening in proper format with date.

Repository module which going to store source code and test request will do via interface provided by the Database module. Doing this way will make sure that establishing and closing the connection to database is happening properly.

Builder, which is the main module, will interact with File Parser and Build Decider directly.

Coming to user level interface we are going to consider different ways the user will use the application. For our application, up to project 2 user is going to interact via console. From project 3 we will be developing GUI. For console interaction, we are going to display proper formatted messages so that user will respond back promptly like letting the application know which files to be compiled. For GUI, we will be displaying all code check-in files and test request, by which user can select and command to build.

# CRITICAL ISSUES

So far, we have seen the how our application can be used and its impact, different activities it can do, how we are going to partition the application and different interfaces we will come across. Now, let’s discuss various issues we must consider before we start implementing our design.

## Communication

As the application is multi process, intra process communication is very important. How we are going to define the message passing structure is very important. It is so because message should carry every important detail for the

## Usability

Our application is going to be used by Managers, Sales team and Customers, in addition to Developers and QAs. These people will use to get very little information and they expect simple user interface. So, initial screen of the application should be simple with minimal information. At the same time, we should provide options on UI to get more details for other users.

QAs and Developers will have to submit Test Request to test the code. This Test Request will be in XML format. Few people will not be interested to create XML file as it is hectic task. So, our UI should provide options such that when they select or enter values and XML created automatically. At the same time, they should be able to download the created XML file, so that they no need to enter the values again when they want to do same testing again. This make us to have upload feature in our UI.

Developers may be uploading lot of source code files. Once they submitted the upload files they would like to know to status of the upload. So, we would be displaying the status with percentage of upload done.

At the same time when build or testing is happening in the back end. Developer or QA may want to know the status of it. We will be displaying percentage as status on UI. Development of this feature will depend on feasibility and time constraint.

## Availability

Build server will take time to build. The time it takes depends on the volume of the code in the repository. If the code is in Kilo Bytes or few Mega Bytes it will take less time to build. But if the code is huge it will take a lot of time, which may be hours. During this time if someone else wants to build by checking in the code, they will have to wait for the current build to complete. In the agile development environment, this is not acceptable.

There are different ways to handle this issue. One is to spawn a new build process every time we need going to build. Other is to notify Clients that a build is about to happen. This way a Client can defer the build. That means our application should have a feature to defer build. We will give this feature via User Interface.

## Performance

Build is a resource demanding task, it will take lot of CPU time to build and memory to keep files that should be build. This can make the server to be non-responsive for other tasks. But it should be responsive all the time. We are going to handle this by process pool and sharing of resources between process.

We can even handle this by building at module level. That is, we are going to build each module separately. At last we will build the whole project by combining each module output. Building this way going to help in building dependent package first. That is, if there are two packages A and B, and A depend on B. We are going to build B package first so that A will not fail while its building is happening.

## Complexity and Flexibility

After few enhancements, many software will be hard to maintain or enhance because of complexity in their design and inflexibility of the code for extension or replacement with different module. To avoid this, we are going to follow modular and interface design. Each important task we are going to be developed as a separate module. So that we can replace that module with different one using different approach. Interface design will help us to abstract the internal implementation from user. This will make us to replace internal logic without worrying user.

## Security

In a project, every team member may not know when it is right time to trigger a build. He/she may not aware of other unchecked files. If the person has access to Build Server and triggered build, which may fail because of dependency, it will waste the time and resources. So, our application should have capability to check whether the triggered person is authorized to do so. This feature will even help when a member out of the team try to trigger build.

## Safety

We should consider the safety of our project in terms of data it holds. Data could source code, test request, log files, build files and libraries. Safety will be in terms of retrieval of data when system crashes or data theft. This will have high impact on our design. We should consider redundancy of data, but this application scope and time is limited. So, this is not feasible. In real time, this feature should be considered.

## Demonstration

Demoing to instructor and teaching assistant that we have met all the requirements is also critical issue. For this we are going to display stepwise every important message on console.

# CONCLUSION

From this whole discussion, we can understand that we can build a federation of servers, such that we can do continuous integration. This will in turn help us in better and faster delivery of a project. However, we should make sure that we are handling all critical issues we have noticed, while building a robust application.

# REFERENCES

* <http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/lectures/Project1-F2017.htm>
* <http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/lectures/StudyGuideOCD.htm>
* https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/MidTerm/MTF17/
* Helper code of Prof. Jim Fawcett