**CONTINUOUS BUILD AND INTEGRATION SYSTEM**

**ARCHITECTURE DOCUMENT**

**TANMAY FADNAVIS**

**SUID: 971141760**

SOFTWARE MODELING AND ANALYSIS/ CIS-681/FALL-2014/ 12/08/2014

Contents

[**1.** **EXECUTIVE SUMMARY** 3](#_Toc405823424)

[**2.** **ARCHITECTURE OF THE SYSTEM** 4](#_Toc405823425)

[**3.** **REPOSITORY SERVER** 6](#_Toc405823426)

[3.1 IMPLEMENTATION 6](#_Toc405823427)

[3.2 ACTIVITY DIAGRAM 15](#_Toc405823428)

[3.3 PARTITIONS 17](#_Toc405823429)

[**3.3.1 EXECUTIVE PACKAGE** 18](#_Toc405823430)

[**3.3.3 COMMUNICATION PACKAGE** 18](#_Toc405823431)

[**3.3.3 DEPENDENCY ANALYZER PACKAGE** 19](#_Toc405823432)

[**3.3.4 REPOSITORY** 19](#_Toc405823433)

[**3.3.5 FILE MANAGER** 20](#_Toc405823434)

[**3.3.6 CHECK-IN PACKAGE** 20](#_Toc405823435)

[**3.3.7 METADATA BUILDER PACKAGE** 20](#_Toc405823436)

[**3.3.8 SENDER PACKAGE** 20](#_Toc405823437)

[**3.3.9 RECEIVER PACKAGE** 21](#_Toc405823438)

[**3.3.10 AUTHENTICATOR PACKAGE** 21](#_Toc405823439)

[**3.3.11 MESSAGE DECODER** 21](#_Toc405823440)

[**3.3.12 ERROR HANDLING PACKAGE** 22](#_Toc405823441)

[**4.** **BUILD SERVER** 22](#_Toc405823442)

[4.1 IMPLEMENTATION 23](#_Toc405823443)

[4.2 ACTIVITY DIAGRAM 27](#_Toc405823444)

[4.3 PARTITIONS 29](#_Toc405823445)

[**4.3.1 COMMUNICATION PACKAGE** 30](#_Toc405823446)

[**4.3.2 RECEIVER PACKAGE** 31](#_Toc405823447)

[**4.3.3 SENDER PACKAGE** 31](#_Toc405823448)

[**4.3.4 MESSAGE DECODING PACKAGE** 31](#_Toc405823449)

[**4.3.5 DLL BUILDER** 32](#_Toc405823450)

[**4.3.6 ERROR HANDLER** 32](#_Toc405823451)

[**4.3.7 CACHE** 32](#_Toc405823452)

[**5. TEST HARNESS SERVER** 32](#_Toc405823453)

[5.1 IMPLEMENTATION 35](#_Toc405823454)

[5.2 ACTIVITY DIAGRAM 37](#_Toc405823455)

[**6.** **CLIENT** 39](#_Toc405823456)

[6.1 CLIENT ACTIVITY 40](#_Toc405823457)

[6.2 CLIENT PARTITIONS 42](#_Toc405823458)

[**6.2.1 EXECUTIVE PACKAGE** 43](#_Toc405823459)

[**6.2.2 GUI PACKAGE** 43](#_Toc405823460)

[**6.2.3 FILE MANAGER PACKAGE** 44](#_Toc405823461)

[**6.2.4 CHECK-IN PACKAGE** 44](#_Toc405823462)

[**6.2.5 CHECK-OUT PACKAGE** 44](#_Toc405823463)

[**6.2.6 TEST SUITES BUILDER** 44](#_Toc405823464)

[**6.2.7 SENDER** 45](#_Toc405823465)

[**6.2.8 RECEIVER** 45](#_Toc405823466)

[**6.2.9 COMMUNICATION** 45](#_Toc405823467)

[**7.** **VIEWS** 46](#_Toc405823468)

[**8.** **ADDITIONAL IMPLEMENTATION/FUNCTIONALITIES** 51](#_Toc405823469)

[**9.** **USERS** 58](#_Toc405823470)

[**10.** **CRITICAL ISSUES** 63](#_Toc405823471)

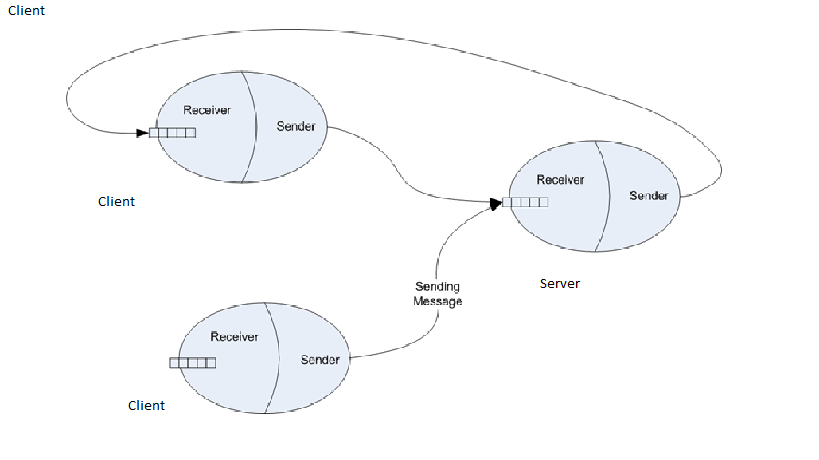
[**11.** **CONCLUSION** 68](#_Toc405823472)

[**12.** **REFERENCES** 68](#_Toc405823473)

# **EXECUTIVE SUMMARY**

This document describes the architecture of Continuous Build and Integration System, which we will refer to as CBIS henceforth in the document. In a large project development environment, the integration and testing of the code becomes very difficult. The project manager needs to check on the progress of development of the project, the software developers need to integrate their modules with other project modules making sure that the integration is successful and does not break the system. The crux is the successful communication between the developers and testing of the entire system whenever a new code is added to the baseline keeping the quality of the product at its top level. The CBIS is exactly the tool which aims to organize and automate the integration and the testing process. The CBIS would be having many arbitrary number of clients, i.e. on which the development process would take place. There would be three servers initially present namely the repository server, the build server and the test harness server. Each of which would be described in details later in the document.

We can think of each entity as a peer to peer system.



The above image is provided by Dr. Fawcett. Above is a very general communication system. Each entity would be having a receiver queue. This queue would be storing messages which it receives from other clients or servers. Each entity is a multithreaded entity, it could process multiple requests at the same time. This has been discussed in details later in the document. Asynchronous message passing would be used for the communication.

Few of the users of the system would be software developers, software testers, project managers, quality assurance engineers, software architects, maintenance engineers. How would the system be used by each user is discussed later on.

Few of the critical issues of the system are,

* The testing of the entire source code stored on the repository which we refer to as a baseline.
* The size of the messages being transferred between the various among the entities.
* File contention issues on the repository server.
* Cache coherency and cache replacement issues on the build and the test harness servers.

These all and many more would be discussed in the appropriate sections.

# **ARCHITECTURE OF THE SYSTEM**

The CBIS system will have the following architecture. This image is taken from the Pr5 requirements, provided by Dr. Fawcett.



The overall flow of the application would be something like below.

The development would take place on all the clients i.e. all the users would communicate with the server using the client GUI. The user can check-in or check-out the code from the repository. Also, the user can contact the test- harness server if he/she wants to conduct tests on some/all of the modules, or the entire baseline. The repository server would be storing the source code along with the test files for that code. A package must have its corresponding test file. Additionally, the repository would also store the test results that have been generated by the test harness. The build server is obliged to build the source code and the test files and output the DLLs for the same. The build server could get request for build either from the repository server or the test harness server. The build server would be having a cache which would store the recently build images. The test harness server is obliged to carry the testing on the images provided by the build server.

The above is a very naïve basic architecture, each entity is described in details in its individual section.

# **REPOSITORY SERVER**

The repository server stores the source code and its corresponding test files. All the developers would check in their code/test files on the repository. Thus our repository will act like a SVN, you can consider it as a GIT repository which is used to integrate the various modules of a large project. But, just the integration of the code is not the final outcome. The integration should be a successful one i.e. adding new code in the existing baseline of the project should not break the project in any case. So prior testing is necessary. Hence, the dependency analyzer of our code repository will come into the picture. The job of the dependency analyzer is to find out the package dependencies among the various packages of all the projects stored in the repository. Based on these dependencies, whenever a new package or a module needs to be check-in, prior testing would be performed. The new module, and the dependencies module would be given to the build server to build the DLLs, these built images would be given to the test harness, which would perform the required testing. Once, all the tests are successful, only then the check –in would be permitted. Another flow could be, if a tester wants to tests some modules, he could contact the test- harness and the flow could be reverse, this we would be discussing in the later section.

## 3.1 IMPLEMENTATION

* The repository is said to be storing the baseline. What is a baseline?

Baseline here is referred to the source code which is already present in the repository. This source code is been tested and has successfully been integrated. Only after that the source code could be added to the baseline. Thus, the up and running completed project is referred to as a baseline.

* The project would be consisting of modules, which are defined as a list of packages. What exactly are these modules?

Modules are a set of packages which when developed, defines a functionality of a project. A software project is made up of many modules. A team of developers is given the responsibility of developing the entire module. “The module is a system block that allows us to assemble a system in such a way that any of these parts could be changed and re-built without requiring the entire system to be rebuilt.” (Dr. Fawcett). Thus, a module are packages which export only the object factory and the interface. The interface could be used to access the package service. The advantage about this design is that, the interface and object factory would be packaged as DLLs. These DLLs would need to be rebuilt, if any changes are made in the module. These could be easily tested by the “Test Harness” server. Thus, each project would be made up of various modules. Also, once the object factory and the interface has been defined, it could not be changed without the consent of the project manager or the software architect.

* Will the repository server perform authentication check?

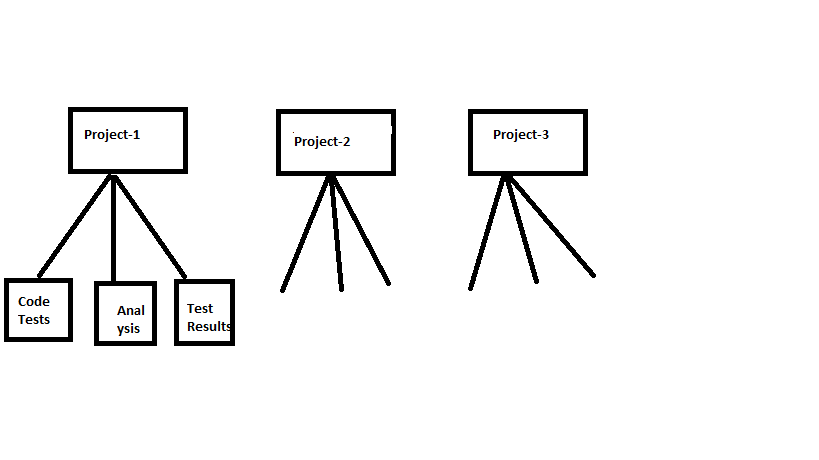
Yes, the repository server has to perform the authentication check on who is accessing the contents. Repository server is responsible for storing large number of projects. Each project will have hundreds of developers working on them. A particular developer should not have access to the files which are not part of the project he/she is working on. Thus authentication is a must. Authentication is also required for check-in check-out policy.

* What is this check-in, check-out policy? Who can check-in the code to the repository? Who can check-out the code from the repository? What exactly does check-in and check-out mean?

Check-in means, once the developer has completed coding his module/package, he needs to integrate his code with the rest of the system. Thus, the code needs to be integrated and kept at a single place. Putting the code onto the repository is referred to as check-in the code. We will follow the single-ownership policy i.e. each package would be having a single owner, in most of the cases that would be the developer of that package. Only the owner of the package can check in the code into the repository. One implementation of this would be, a team would be assigned a common login. Using this login only, the code could be checked-in. This login would be the responsibility of the team lead. Thus, if any developer wants to check-in the code, he needs the team lead’s consent. This is useful in scenario where, a newly hired intern is given the responsibility of a module. He/she should not be directly allowed to check-in his/her code. Thus, all the code check-ins are routed through the team lead. Who can check-out the code? Any developer who is a part of the project team can check-out the code from the repository. Again the check-out could only be done using the team login id. Check-in files means to upload the files onto the repository server. The upload feature would be present on the client GUI. Using the GUI, check-in could be performed. The check-out could be done again from the GUI. The developers could use command line to perform the check-in, check-out too.

* How exactly will the repository store information? What all information would be present on the repository server? What is the organizing principles/directory structure of the repository?

As the repository would be storing information about many projects, each project would be consisting of large number of modules, the directory structure of the repository would be based on projects i.e. on top level, there would be different projects, and the second level would be consisting of various modules which make up the project. It will also consist of various test results pertaining to that project. Along with that the dependency analysis results would also be stored. Below would be the directory structure.



* What will be the steps for a successful check-in of a new package?

A new package cannot be check-in just be uploading it to the repository. Below would be the tests necessary for the successful check-in.

Once a new package has been uploaded to the repository, first the authentication of the user would be checked. Once the user is authenticated, the new package would be stored in a dummy region, virtual server concept would be used here. The new package will be referred to as an orphan package hence forth. This orphan package might be dependent on many other packages. The dependency could be other way round i.e. many other packages would be depending on this new package. Thus, we have to make sure that this new package does not break the dependency chain. Dependency chain here refers to, say a package A depends on package B which intern depends on this new package. If adding this new package to the baseline breaks something in package A, this could lead to lot of trouble. Thus, we need to make sure that such type of errors does not occur. Here dependency analyzer would come into the picture. This new package, once it has been uploaded on the repository, the dependency analyzer would do incremental analysis, and find out all the dependencies of this particular package. Once, all the dependency has been found, all the packages would be given to the build server to build DLLs. Only if the build is successful, the build server would forward them to the Test harness server. The test harness server would carry the testing. Only if the testing is successful, the check-in would be done. But, not only would the dependent packages of this new package be tested. The packages which depend on the dependent packages would also be tested to make sure that the whole dependency chain is not broken. If the test results are successful for all the tests, only then the check-in would be successful, else the user would get the notification that the check –in has failed and the package would be deleted from the repository. The user would also be displayed the particular build error/ test failure error on the client GUI, if check-in fails and the success error if the check-in is successful.

* What exactly is incremental analysis? How is the dependency analyzer carrying out this incremental analysis?

The dependency analyzer developed in Pr4 would be used to carry out the analysis. This dependency analyzer analyzes all the packages of a particular project, based on the file names, and store the types, their relationships and the various type and package dependencies in a directory structure. Incremental analysis refers to the process of continuous analysis, i.e. there would be a thread call it the analysis thread which would continuously be running on the system. As soon as any new package arrives, this thread would carry out the analysis and update the type/ dependency table. Thus, at a given point of time, we would be having the dependency analysis result. This dependency analysis is useful because, as we had discussed, the CBIS system is a continuous integration system, thus whenever a new package is been added, we need to test all the dependent packages/modules to make sure nothing breaks in the project. Thus, dependency analyzer would be used.

* Is the repository server a multi-threaded server? How is multithreading being implemented?

As the repository server would be contacted by many clients/servers, multi-threading is necessary. There would be a main thread, the task scheduler, which will collect the message from the receiver queue of the repository. A new thread would be spawned based on the message type. The message catalog and the communication system is discussed later in the document, in the same section. Based on the message type, the corresponding action would be taken by the child thread. As this is a multithreaded environment, file contention would need to be taken care of. Locking should be implemented, this could be implemented by using C# locks.

* What is an orphan package? When does the orphan package gets added to the baseline?

Orphan package is the package which has been uploaded by the developer recently to the repository to add it to them baseline. The orphan package does not get added to the baseline until it passes all the tests as described above.

* How will the orphan package concept works?

The orphan package would be treated as a normal package, but it won’t be checked in, into the repository before the testing successes, thus all the dependencies would be found out, and given for the build server to build images.

* What is the meta-data of the package/module? How and when it is used?

Metadata is a file attached to each and every package. The meta-data contains all the information pertaining to its package. It is an XML file which has information like, who is the owner of the package, the package belongs to which module of which project, when was the package uploaded to the repository, when was the package being added to the baseline, when was the package last tested. Along with these, it also has dependency information, i.e. which all packages does this package depends on, which packages depends on this package. This dependency information would be useful during testing as mentioned before.

* What if a new version of the package is added to the repository? How will the new dependencies be affected? Are the older versions deleted from the repository?

No, the older versions cannot be deleted from the repository. Once the code is added to the baseline, the code is immutable. This is necessary under scenarios, in which a client/ or a developer wants to refer to some older version of a code, he could easily go back to the older version if it is present on the repository. If we delete the older version, it would be very difficult to migrate the system to older version. Thus, whenever, a new version of a particular package is checked-in, a new meta-data file would be created. After the successful check-in, the new meta-data file would add the dependencies that this package depends on. This could be done by referring to the older version of the meta-data. But, as a result of this new version being check-in, the packages which depended on the older version of the package needs to change and point to this new version. The metadata file of those packages would need to be changed simply and point to this new version. As the interfaces are not changed, nothing else need to be changed except the metadata file. Thus, the packages versioning system would be implemented in this way.

* What communication system would be used by various clients/ servers to communicate with the repository?

Message passing would be used for communication. WCF would be used to implement the message passing system. The repository would be having a receiver blocking queue as discussed. All the messages would be dumped into that queue. The main thread scheduler would take the message from the queue and spawn a new thread each to serve a new message.

* What are all the various messages which will be exchanged by the repository? What would be the format of the message?

The messages would be in an XML format for universal use of application. The repository would be contacted by the clients for check-in and check-out of code. It would also be contacted by the build-server. The message catalog would be as follows. There would be various type of messages that the repository would have to handle.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. | Sender address | Receiver address | Project information | Message body | Additional information |
| 2. | Sender address | Receiver address | Module information | Message body | Additional information |
| 3. | Sender address | Receiver address | Check-in information | Message body | Additional information |
| 4. | Sender address | Receiver address | Check-out information | Message body | Additional information |
| 5. | Sender address | Receiver address | Dependency information | Message body | Additional information |
| 6. | Sender address | Receiver address | Error information | Message body | Additional information |
| 7. | Sender address | Receiver address | Test results | Message body | Additional information |

Above is the message catalog for some of the messages which the repository server would be receiving. The 4 column is the type of the message. The second and the third is the sender and the receiver address, the fifth column is for some additional information.

1. The client can ask about the project information of the project he is working on. This information would consist of the project start-date, the number of modules and such metadata information.
2. The client can also ask the information about a particular module, e.g. a module the developer is working on.
3. If the developer wants to check-in few packages, he will send the check-in message to the repository. The message body would be consisting of the actual message i.e. the package source code and the test file associated with that.
4. If a developer wants to check-out few packages, he will send the check-out package information.
5. The builder server could send the dependency information, asking for all the dependent packages of a particular package.
6. If there is some error, error type message code would be sent.
7. The test results message type would be sent by the test harness to the repository server. The message body would contain the actual test results in an XML format.

* How will a developer revert back to the older version of a package?

Simple. The metadata file could be used in this scenario. We can just use the old metadata file that point to the older version of the package.

* How will a source file be matched with its corresponding test file?

Every source code file should have a corresponding test file associated with it. This would be the first check the repository server would perform. If there is no associated test file with the source file, an error message would be sent to the client. One way of associating the source code with its test file is the name. The name could be the same for both the source code file and the test file with TEST\_package1.cs as the name. But this is not a re-usable model. Hence, we would be using a standard naming conventions. The naming convention would be the package name followed by the package id, e.g. parser\_p1, the corresponding test file would be TEST\_parser\_p1. In this way, we could associate the source code file and its test file.

* When will the metadata file be generated for a package?

Once, the check-in is successful i.e. once the test results return success, the source code file, along with its test file would be stored in the corresponding project directory. The thread which receives the success message could do the work of creating the metadata file, storing the information. It will also fetch the package’s dependency information and store it in the metadata file.

* Response time from the build server. How much time the repository server would wait for a response?

All the communication would be asynchronous i.e. once the repository sends a message, he can continue with the other processing rather than waiting for a message. But the question arises of how much should the repository server waits a response? One solution to this would be to use an agent. The agent’s work would be just to send the messages to other entities and wait for the response. The agent could start a timer whenever it sends a message for e.g. to the build server. Once the timer expires, he can notify the repository so that the repository could again re-send that same message.

* How will the server find out the dependencies using the dependency analyzer?

The dependency analyzer developed in Pr4 would be used, it finds out the package and type dependencies based on the various rules and actions. The detailed information regarding this could be found in Operational Concept Document-Pr3.

* What if another client makes a new request to the repository server, while the server is processing earlier client’s request?

The processing at the server end is of multi-threading. As discussed earlier, the main thread would be spawned into multiple threads. These child threads would be responsible in sending the response to the client. The client information is stored in the message which the client sends to the server. The message contains the receiver information as seen in the message prototype. Using this information, the child thread would come to know which client’s request it is processing and thus, the response would be sent to the appropriate client. Even if a new client request arrives, that request would be having the client address and a new child thread would be spawned. Thus, the server would be able to distinguish between various clients/servers.

* How will the authentication be performed?

When the project is first checked in into the server, the admin could include a configuration file associated with each project. The configuration file would include the required username and password, assigned to each team of that project. Using this, the authentication could be performed.

* How will the initial baseline be built?

The design of the repository server is such that, no new activities need to be performed. When a package/modules arrives into the repository, the dependency analyzer would do the analysis, as it’s the only package/ that would be sent to the build server and thus the incremental analysis and continuous process would take place. Eventually, more packages would be arriving and integration would take place.

## 3.2 ACTIVITY DIAGRAM

Below are the activities that the repository server would do.



1. The repository server would open connection to accept the messages from the server/clients.
2. Also, the repository server would assign a thread for incremental analysis. The details of how the dependency analysis is being carried out has been discussed in Pr3 OCD.
3. The analysis would be carried out for each project.
4. Once a message has been received from any entity i.e. the client or the server, the task scheduler would dequeue the message from the blocking queue.
5. Once a message has been received, a new thread would be spawned by the task scheduler.
6. The very first task, the repository server would perform is that of the authentication. How the authentication would be performed is described above in the document under the implementation section.
7. Once the request has been authenticated, based on the message type, the action would be performed.
8. If the message type is that of check-in, the repository would get the dependencies of the orphan package, take all the modules and send it to the build server.
9. If the message type is that of check-out, the repository would get the files, prepare a response and send it to the client.
10. If the message type is that of error message from the build or the test server, the appropriate message would be sent to the client that the check in has failed.
11. If there is success message from the build server, the corresponding file would be checked in, metadata for that file would be created and added to the baseline.

Thus, these are the activities that our repository server would be carrying out.

## 3.3 PARTITIONS

The repository server is partitioned into the following packages.

Below is the package diagram for the repository server and the interfaces between them.



The above diagrams depicts the various packages of the repository server. We will discuss them in details.

### **3.3.1 EXECUTIVE PACKAGE**

This is the main starter package of the repository server. This package has two main responsibilities. Firstly, it will start the communication server of the repository server, i.e. it will initialize the blocking queue, and the main thread i.e. the task scheduler would start listening. Secondly, it would also start the dependency analyzer thread for incremental analysis.

### **3.3.3 COMMUNICATION PACKAGE**

This package main obligation is just for communication. The communication package would be contacted by two more packages i.e. the sender package and the receiver package. The communication package, after the repository server receives the message would pass the message to the receiver package. Also, the sender package would contact the communication package and the communication package would send that message to the client/server based on the destination address.

### **3.3.3 DEPENDENCY ANALYZER PACKAGE**

The dependency analyzer package would be finding the various type and package dependency analysis between the files of various projects. This has been shown as just a single package here. The detail diagram of the dependency analyzer has been discussed in Pr3- OCD. The dependency analyzer package would receive the files to be analyzed from the file manager package. Based on the input from the file manager, the dependency analyzer would analyze the files and build the type table. The type table would contain the type dependencies, package dependencies among various files of a particular project. This table would be stored in the corresponding project directories. Also, the thread which carries on this task of finding the dependencies would be started by the executive package, so that incremental analysis could happen.

### **3.3.4 REPOSITORY**

The repository package is the main crux package. It would contain all the source code files, the test results and the test files for each and every project. As discussed in the directory structure, all files would be kept project wise and all the corresponding modules would be kept in that project directories along with the test results. The repository package would be communicating with two other packages. One would be the check-in package. This check-in package would carry out the task of check-in the source code and the test files, if the testing and building is successful. Other would be the file manager package. The file manager package would take the dependent files from the repository and give it to the dependency analyzer for finding out the dependencies. The orphan packages would also be stored in the repository but in a separate directory.

### **3.3.5 FILE MANAGER**

The job of the file manager package is just to get the files from the repository and give them to the dependency analyzer for finding out the dependencies. Also, once the testing is successful of an orphan package, this package would be given to the file manager, from which the check-in package would then take this new file and send it to the repository for adding in the project baseline.

### **3.3.6 CHECK-IN PACKAGE**

The job of the check in package is to do the final check-in once the testing has been completed. It will take the package from the file manager, which needs to be checked in. Also, the metadata of that package would be taken from the metadata generator package. Thus, once it has all the information, it would store it into the repository into the corresponding project directory.

### **3.3.7 METADATA BUILDER PACKAGE**

The metadata builder package is obliged to just build the metadata of a new orphan packages which has passed its tests. This package would get the success notification from the message decoder package. Once this notification has been received, the metadata package would get the package information from the file manager package, the dependency information from the repository. With all this data , a metadata file would be built for the package. The check-in package would take this information which would be used during the check-in.

### **3.3.8 SENDER PACKAGE**

The sender package’s job is just to create response messages and give them to the communication channel which would then send them to the appropriate client/server. The sender package would be contacted by the check-in package to notify the client that the check-in has been successful. It could also be contacted by the authenticator package if the authentication of a client or a server fails.

### **3.3.9 RECEIVER PACKAGE**

The only job of the receiver package is to receive the message which would be sent to it by the communication package and forward the message to the authenticator package. The authenticator package should be the first step as if the client/ server is not authenticated, then that particular request should not be proceeded.

### **3.3.10 AUTHENTICATOR PACKAGE**

The authenticator package does the work of authentication of the request. It checks for the project id for which the request has come and checks the corresponding login details from the repository. If the authentication is successful, only then the request would be sent to the message decoding package, else an error response is sent to the sender package, which sends the response to the communication package, which sends it back to the client/server.

### **3.3.11 MESSAGE DECODER**

Message decoder package is obliged to decode the message type and take appropriate actions based on the message type. If the message type is a check-in message, the request would be forwarded to file manager, which would then forward it to the repository. If the message is a check out message, again the repository would be sent the message via the file manager. If the message is a success message from the build server, which implies that the new orphan package has successfully completed its testing. Thus, the message would be forwarded both to the metadata builder and the file manager. If the message is an error message, it would be forwarded to the error handling package, which would then tell the repository to delete the orphan package and send the message to the sender that the check in has failed.

### **3.3.12 ERROR HANDLING PACKAGE**

This package has the responsibility of just accepting the error messages from the message decoding package. It then tell the repository to delete the orphan package and then sends the check in fail message to the server. Also, if there is any error in the decoding of the message itself, it would be forwarded to the error handling message, which will take necessary actions, and thus the system won’t be affected.

# **BUILD SERVER**

The build server is obliged to build the DLLs which are required for the test harness for the testing. The build server is the one responsible for building the modules/packages which are dependent on each other. Whenever a developer develops a package/module, he does the independent building of the package and the independent testing on his local machine. But it is important that the building should succeed when the code is merged into the baseline. If a developer’s code is added and if the build fails due to some error in the coding, then the entire project would be failing. Hence integrated building is very necessary for the system to run and for the test harness to run the tests. The build server could get requests from both the repository server and the test harness server. The client won’t be communicating with the build server.

If a new package has been requested for check-in, in the repository server, the repository server would be finding all the dependent packages and giving these packages to the build server. The build server would thus be building these packages and creating the corresponding DLLs. If the build is failing, the build server would send the error response to the repository server regarding the failed build. If the build is successful, the build server would be sending this successfully built DLL to the test harness server for running tests. If the test harness returns successful results, the build server would be returning the success message to the repository, else it would be returning the error message to the repository server. Thus, in this way, build server would be communicating with the repository server.

The build server could also get messages from the test harness server. Consider a scenario, where a tester wants to test some packages. He would be directly contacting the test harness server. The test harness server would be sending the request message to the build servers to get the images for the packages and its depending packages requested by the user. The build server would then send the request to the repository, the repository would be sending the response to the build server with the requested packages. The build server would be building those packages, and if the build is successful, it will send the success message along with the image to the test harness server. If the build is not successful, build server would be sending an error message to the test harness server.

Cache would be playing an important role in the build server. The build server would be storing the recently build images in the cache. How will the cache be useful would be discussed in the next section i.e. the implementation section.

## 4.1 IMPLEMENTATION

* How exactly would the build server be useful in the CBIS system?

The build server would do the work of building images and sending them to the test harness server. The test harness server could carry out testing on these DLLs. Thus, the build server would be helpful in making these DLLs. This building could also have been performed on the repository, but then it wouldn’t have been a very re-usable design. As the build server’s only functionality is building the files, this adheres to the object oriented concept.

* How will the repository server communicate with the build server?

The repository server would be sending all the modules which it needs to be tested to the build server for building the DLLs. The build server would accept these modules/packages and build the images. If the build is successful, the image would be sent to test harness server, else if the build fails, an error message would be sent back to the repository server. This would be the scenario where a new package wants to be checked-in.

Also, if the build server gets a request from the test harness, the build server would be sending the package/module name for which it wants all the dependent packages to the repository server. The repository server would then send all the packages for building. Thus, in the above scenario, the build server and the repository server would communicate.

* How and when will the test harness server communicate with the build server?

Again, similarly to the repository server, the test harness server would be communicating with the build server in two scenarios. One, if the build server gets a request from the repository server, the build server would build the images and send it to the test harness server. If the testing is successful, the test harness would return the success message along with the testing results. If the test fails, the test harness would be returning the failure message along with the failure error. Also, the test harness could send the request to the build server for a specific package and its dependent packages which are requested by the user. The build server would forward the request to the repository. The repository would then be sending the modules/packages along with its test files to the build server. The build server would then build the images and forward them to the test harness. If the build fails, the notification would be sent to the test harness.

* Will the build server also be multithreaded like the repository server?

Yes, the design of build server would be very similar to that of repository server in terms of the multithreaded design. There would be a blocking queue, which would be storing all the incoming messages. The task scheduler would spawn a new thread for every new message just like the repository server.

* How will the caching in the build server be useful?

The build server would be having cache of all the recently built images. This would consist of approximately 1 % of the repository information. The cache would be useful when a request from the test harness arrives. If the test harness is requesting an image which is already present in the build server cache, the build server would send the image of the cache rather than requesting it to the repository. Thus, the cache would be very helpful in reducing the network traffic. This is an important decision in a project where huge number of data is continuously been exchanged between servers/clients.

* Size of the cache? What should be the size of the cache present on the build server?

The size of the cache is again a very important decision. If the cache is too small, less number of images could be stored on the cache, which would not be of a great use to reduce the network traffic. If the cache size if big, then again time would be lost in searching the cache. So the latency would again be the same. Thus, the cache size should be big enough so that it could store a lot of images, but it should not be that big that the time is lost searching the entire cache.

* What will happen if the cache is full and if a new image is just being developed?

The cache replacement policy would be very helpful in clearing out the old images. The best policy would be to use least recently used i.e. LRU policy. The image which has not been used since a long time would be deleted from the cache of the build server.

* Where will the images be uploaded to the test harness?

The test harness would only test the DLLs which have been uploaded on a specific path provided by the test harness server. Besides that path, the images could not be uploaded onto the test harness. This is done for the security purpose. Only that particular directory would be available for the build server to upload the images.

* What would be the types of messages the build server would be receiving? What will the build server do after he receives the message?

The message catalog for the build server would be as below. Column 4 is of the message type.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. | Sender address | Receiver address | Build images | Message body | Additional information |
| 2. | Sender address | Receiver address | Modules information | Message body | Additional information |
| 3. | Sender address | Receiver address | Success/ Test results | Message body | Additional information |
| 4. | Sender address | Receiver address | Error messages | Message body | Additional information |
| 5. | Sender address | Receiver address | Module request. | Message body | Additional information |
| 6. | Sender Address | Receiver address | Images to test | Message body | Additional information |

Above would be the messages which the build server would be receiving.

1. The message would definitely contain the sender address and the receiver address. The message type and the body of the message would change based on the request. First type of message is the “build images” message type. This would be sent by the repository server when it is requesting the build server to build the image. The message body would contain the various packages (dependent packages), for which the repository wants the images to be built and tested. If the build is successful, the build server would send the image and send the “Images to test” type of message to the test harness, which is requesting it to carry the testing on the image.
2. The “modules information” type of message would be sent by the build server to the repository, when it gets the request from the test harness for a particular module. First, as discussed, the build server would check its cache, if it has the image. If the image is not present, this type of message would be sent to the repository server asking for the module/ and its dependent’s information.
3. The “success/test results” message would be sent by the test harness server to the build server, after the image which the build server has provided has been successfully tested. These results would be forwarded to the repository server.
4. The “module request” type of message would be sent by the test harness server to the build server, if it’s requesting some module and its dependency images to be build.
5. The error messages would be sent by the build server if the build fails or the test fails on the images.

Thus, the build server would be running continuously, as it gets the images it would be building DLLs, caching the latest images and sending the results to the repository server and the test harness servers.

## 4.2 ACTIVITY DIAGRAM

Below are the activities that the build server would be performing.



1. The build server would start listening to the connections.
2. Once a message is received from any server (repository / test harness), the message would be put in the blocking queue.
3. The task scheduler would retrieve the message from the queue and check the message type.
4. Based on the message type, appropriate action would be taken.
5. If the message is received from the repository server, the build server would carry on the building of image. If the build is successful, the image would be forwarded to test harness server. If the build fails, the error message would be sent to the repository server.
6. If the message is received from the test harness, that of the successful implementation, the test results would be forwarded to the repository and the particular image would be stored in the cache.
7. If the messages is received from the test harness about a particular package, the build server would first check, if the corresponding image is present in its cache. If yes, that image would be sent to the test harness.

If no, the request would be sent to the repository to get the corresponding module/packages along with its dependent packages. The build server would build the image and then send it to the test harness server for testing.

Note that the build server won’t be doing any authentication on the requests. The reason being, the requests from both the test harness and the repository server would already have been authenticated.

These are the activities which the build server would carry out.

## 4.3 PARTITIONS

The build server would be having the below partitions.



Above is a very simple implementation of the build server. It won’t be having many packages as it is just building the images.

### **4.3.1 COMMUNICATION PACKAGE**

This packages, does the same work as the communication package of the repository server. Its job is to start the communication channel, using which the build server could send and receive the messages. This package would be communicated by the sender package and the receiver package. The received messages would be given to the receiver package. The messages to be sent would be received from the sender package and sent it to the appropriate destination.

### **4.3.2 RECEIVER PACKAGE**

This package again has the same functionality as that of the receiver package of the repository server. It would just do the task of receiving messages and forward them to the message decoder package.

### **4.3.3 SENDER PACKAGE**

This package has the responsibility of receiving the messages from the error handling package and the message decoder package and build the message which needs to be sent. The sender package then just forwards the message to the communication package which sends the message to its destination. If any error occurs during the building the DLL phase, the error would be sent to the error handling package, which would take appropriate actions and send the error message to the sender. The message decoder would be sending the message, once it has decoded the success message from the test harness server. The message decoder package would then send the test results to the sender package.

### **4.3.4 MESSAGE DECODING PACKAGE**

This package, as the name suggests, decodes the message type received and takes appropriate actions. If the message is from the repository server to build the modules, it would forward message to the DLL builder package. If the test harness sends a success message, the message decoder would first store the DLL in the cache and then forward the test results to the sender package. If the test harness requests for a module/package, the message decoder would first check in the cache if the image is present. If not, the request would be forwarded to the sender package to send it to the repository server.

### **4.3.5 DLL BUILDER**

The DLL builder does the work of building the images. If the build is successful, the image would be sent to the sender package. If the build gives an error, the error is been handled by the error handling package.

### **4.3.6 ERROR HANDLER**

The error handler package would be consulted by the DLL builder package if any error occurs. The error handler would then just kill the thread, as it should not affect the entire system. It would then send the error message to the sender package.

### **4.3.7 CACHE**

Cache is not a package per se, it’s a memory area on the build server which would be used for storing the recently built images, which have successfully pasted the testing. If the cache is full, LRU replacement policy would be used.

# **5. TEST HARNESS SERVER**

The test harness does its work of testing the DLLs which it receives from the build server. We would be using the test harness provided by Dr. Fawcett. As this tester is provided by Dr. Fawcett, we would be just briefly describing the functionality of the tester in general and with respect to the CBIS system.

The test harness has the following class structure.



The above diagram has been provided by Dr. Fawcett.

* Thus, it has an ITest interface. An important point to notice here that each test should implement the ITest interface and its test () method. If the test is not inheriting the ITest interface, that test is rejected at the repository itself. Another constraint of the repository is that if there is no test file, the source code is not allowed to check-in. Thus, each tests needs to implement the ITEST interface and its test method.
* Secondly, there is a Tester class. This class/package creates a child Application domain for each test. What is an application domain? An application domain is a less CPU intensive process. There can be a channel between two application domains which could be used for communication. But the pointer of one domain could not point to other application domain. The biggest advantage of using the application domain is that, even if something crashes in the domain, the rest of the system is not affected.
* Thus, the tester class creates child Application domains. The loader package does the work of loading the test assemblies into the child application domains. This is also referred to as Test Domain. The libraries are loaded from a specific path. It has this constraint that only in the destination directory the libraries should be loaded. The libraries are loaded directly into the child domain. The tests are run on all the libraries that support the ITest interface. This functionality is important so that the primary application domain of the test harness server won’t be needing any information about the test types it will invoke. Below is the Test harness prototype.



The above diagram has been provided by Dr. Fawcett.

* The test results are returned as XML strings. The test harness also has a logger class which provides logging facilities for all the tests that have run. Thus, our test files would be having many functions to be tested. Each function could be defined as a test function which returns bool. The main test() function which has been inherited from the ITest interface would only return true if all the functions which needs to be tested returns true. Else it returns false.
* Below are the step the test harness would follow.

1. First, a child application domain would be created in which individual tests would be running, each would be running its own thread.
2. Assemblies would be loaded into the child domain which needs to be tested.
3. An instance of the class which implements the ITest interface would be created.
4. The tests would be run by calling the ITest.test() function.
5. The tests would report whether the test succeeded or failed.
6. All the test drivers in the loaded assemblies would be tested.
7. After the tests are done, the child application domain would be unloaded. This also unloads the libraries.

The above test harness is provided by Dr. Fawcett which has these functionalities. We would be using this test harness in our CBIS system.

## 5.1 IMPLEMENTATION

* Who will call the test harness? How the test harness is used in CBIS?

The test harness would be used to run the tests on the libraries. The test harness would be contacted either by the Build Server or by the Client directly. The client user could be a tester who wants to run few tests on selected packages. He would call the test harness and give him the test suites which are a bunch of test cases which he/she wants to run. The test suites are defined by XML messages. The test harness would accept this messages. The authentication similar to that on the repository would be performed on the test harness to ensure security. This authentication has already been discussed for the repository server. The test harness would then first check in its cache if he has the test results of the tests the client is requesting. If the results are present in the cache, it would return the results directly. If the results are not present, the test harness would send the request to the build server. The build server would first check in its cache whether he has the image for the tests the test harness is requesting. If not, the build server would forward the request to the repository. The repository would then respond with the corresponding and all the dependent modules. The build server would then build the image and then forward them to the test harness. The test harness would carry out the testing and send the results to the client.

The test harness could also be contacted by the build server in the scenario where a new package needs to be checked-in. The build server would then send the image to the test harness server for testing purpose. If the tests are successful, the test harness would be storing the results in its cache. It would then send the results to the build server which would then be sending them to the repository.

* Can multiple tests be run at the same time?

Yes, as discussed, the test harness would create an application domain for all the tests, and each test would be running on its own thread.

* Will the test harness have cache?

Yes, the test harness would be having cache similar to the build server. The test harness would store all the recently run successful test results in its cache. This could help in reducing the network traffic to a great extent similar to that of the build server.

* What would be the messages the test harness would exchange.

Below is the message catalog.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. | Sender address | Receiver address | Test Suites | Message body | Additional information |
| 2. | Sender address | Receiver address | Run tests | Message body | Additional information |
| 3. | Sender address | Receiver address | Built image | Message body | Additional information |
| 4. | Sender address | Receiver address | Test results | Message body | Additional information |
| 5. | Sender address | Receiver address | Error messages | Message body | Additional information |

1. The first type of message would be given by the client, for e.g. a tester who needs to run tests for some arbitrary number of packages. The message type would be test suites and the message body would contain the information of all the packages, the tester needs to run the tests on. The test harness would first check in its cache as discussed above. If not, the message would be forwarded to the built server.
2. The “run tests” type of message would be received from the built server to the test harness. The message body would contain the images to be tested.
3. The “built image” type of message would also be sent by the build server to the test harness. This message body would contain the images which was requested by the test harness from the built server.
4. The test results would be sent by the test harness to the client requesting the results. The test results would also be sent by the test harness to the build server, if the images which the build server had sent has returned success.
5. The error messages would be used to send the message to the client/ or the build server if any of the test fails.

## 5.2 ACTIVITY DIAGRAM

Below is the activity diagram for the test harness server. The test harness would be carrying out the following activities.



1. The test harness would be starting its service and listening for the requests.
2. Once the request has been received, it could be either from the build server or directly from the client.
3. If the client has made the request, first perform the authentication.
4. If the authentication is successful, then continue with the task, else send error notification to the client.
5. Get the message. Depending on the message type, the test harness would take the appropriate actions.
6. If the message is from the client directly, test harness will check whether the requested test is in its cache. If yes, it will return that else it will forward the message to build server.
7. If the message is from the build server, it will carry out the testing send the test results.
8. If the build server returns the built DLL which the test harness had requested, it will perform the testing and send the test results to the client.

# **CLIENT**

Client in out CBIS system would be the GUI developed using Windows Presentation Foundation. It would present the interface necessary to communicate with the repository server and/or the test harness server. The various users and how will they use the system has been discussed later in the “Users” section. The developers would be developing their part i.e. say a package or a module. They will open the CBIS client if they want to check-in their code. The CBIS client will also be used if the developers wants to check-out code from the repository. Authentication would be used. For this, the request from the client would go to the repository. Another use would be, if a developer/tester wants to test his/her or any code. He just needs to provide test suits to the system, and the test harness would return the test results. It will display which all tests failed and/or passed. This is the overall view of the CBIS client. There would definitely be many arbitrary number of clients. Next we will discuss the client activities and the partitions.

## 6.1 CLIENT ACTIVITY

Below are the activities the client would be performing.



1. The user would first start the CBIS client.
2. The user would be prompted to login. The login would be checked on the local machine.
3. If the login is successful, the CBIS client would display various options, else appropriate error message i.e. invalid login would be displayed.
4. The user could do one of the many things.
5. If the user wants to check-in, he would select the files to be checked-in. The client would then display the appropriate message to the user after the user click the check-in button. If the check-in is successful, success message would be displayed, else appropriate error message would be displayed.
6. The user could check out files from the repository. The client would send the requests to the repository. If the requests is successful, the client would start downloading the files. It will display to the user which files are being downloaded and display the path/directory where the files are being saved.
7. If the user selects the test suites, the client would build the message and send it to the test-harness and display the appropriate pass/fail message to the user.

These are the activities the CBIS client will perform. We have kept the client simple and smart.

## 6.2 CLIENT PARTITIONS

The CBIS client would be partitioned into the below packages.



### **6.2.1 EXECUTIVE PACKAGE**

The executive package is the main start-up package for the CBIS client. It will call the communication package and start the client service. The client would also start listening for the incoming requests. Similar to all the servers, the client would be having a receiving blocking queue. Also, the client CBIS too is indeed a multithreaded environment. The main thread i.e. the task scheduler would retrieve the messages from the queue and spawn a new thread and take the appropriate actions. As the communication is asynchronous, once the client makes a request, it need not wait for the server to respond back. It can continue with its own processing. Thus, the job of the executive package is to start the CBIS client.

### **6.2.2 GUI PACKAGE**

The graphical user interface package, as the name suggests would be doing the job of displaying the results to the client/users. It would also display the functionalities needed for the user to communicate with the repository or the test harness i.e. to upload the module functionality, to display the test results, to display the error messages and to send test suites. It would be communicating with the test suite builder package and the check-in package based on the action user selects. The receiver package would be communicating with the GUI package to display the results.

### **6.2.3 FILE MANAGER PACKAGE**

The file manager package just performs the work of bringing the files from the path specified by the user and sends them to the appropriate package which has called it. The file manager package would be called by the check-in package. The user would input the path of the files which he/she wants to check-in. The check-in package would then contact the file manager package to retrieve the files from the specified path. Also, the test suites builder package would contact the file manager package to retrieve the file names on which the user wants to run tests.

### **6.2.4 CHECK-IN PACKAGE**

If the user selects the check-in tab on the client, (the GUI views are discussed in the next section), and the check-in package would be called. The job of the check-in package is simply to contact the file manager and get the files the user needs to check in, and send these to the sender package. We have kept the functionality of each and every package simple, so that the package design is such that they are re-usable.

### **6.2.5 CHECK-OUT PACKAGE**

The check-out package would be contacted by the receiver package. The receiver package would send the checked-out files to this package. The check-out package is obliged to download the files and store them to a directory location.

### **6.2.6 TEST SUITES BUILDER**

The test suites builder package, as the name suggests, builds the test suites for the user. The user when he/she selects the testing tab, this package would be called. The user would provide the path on which the testing to be carried out files would be present. The test suites builder would then contact the file manager package and retrieve the files. The test suite builder would just requires the file names. It would then build the test suite i.e. the XML file and send it to the sender package.

### **6.2.7 SENDER**

The sender package as the name suggests, is the same in all the entities of the CBIS system i.e. the repository server, the build server. The sender package just does the job of preparing the message and sending it to the communication package. The sender package would be contacted by either the check-in package which would contain the files to be checked-in or the test suite builder package, which would contain the test suites.

### **6.2.8 RECEIVER**

The receiver package would be receiving the messages from the communication package. Based on the message type, either the receiver package would forward the message to the display package, to display the test results, or the error messages to the user. The receiver package could also send the response received to the check-out package, if the user had requested for some check-out of the files.

### **6.2.9 COMMUNICATION**

The communication package has the job of starting the client service. It is just used for the communication to and by the client. The sender package would send the messages to be sent to the communication package. The communication package would then send the messages to the destination. The communication package after receiving the message would then forward the message to the receiver package.

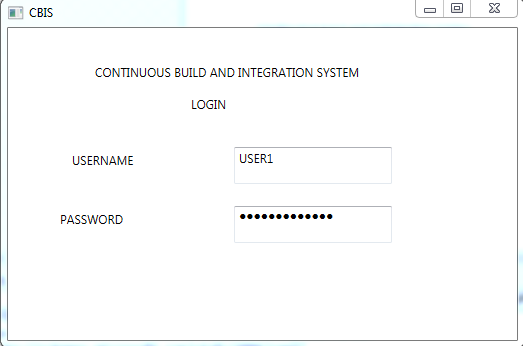
Thus, the CBIS client would be partitioned into the above packages. The above packages are designed keeping the object oriented principles in mind and their reusability.

# **VIEWS**

In this section, we would be discussing the prototype GUI views for the CBIS client. Please note that these are just the prototype to discuss the functionality. Many modifications would be required in terms of design and the UI.

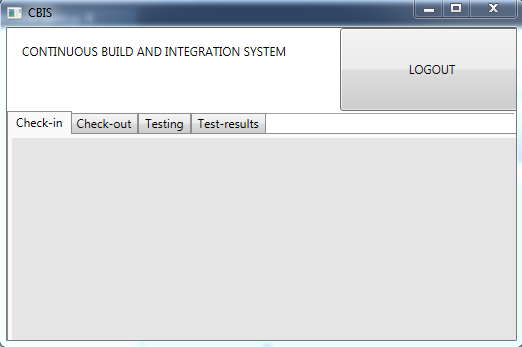
Below are the views.

1. Login view



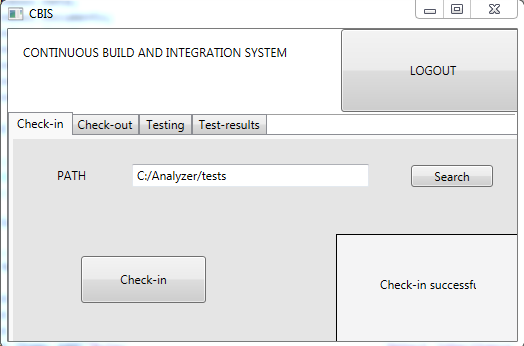
Above view shows the first login screen when the CBIS client opens. The user would be asked to input his/her login credentials. If the login is successful, only then the user would be allowed to proceed like any other login. The login would be checked on the local machine itself.

1. Login successful view



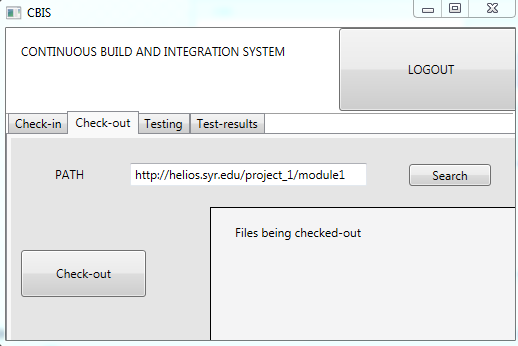
This screen would be shown when the user successfully logs in. There are four tabs each for one function. The check-in tab would be used by the user when the user wants to check-in some code which he/she implemented to the repository. The check-out tab would be used when the user wants to check-out some code from the repository. The testing tab would be used when the user wants to carry out some testing, run few test cases. The test-results tab would be used to display the test-results. The test results tab would only be enabled if there are some test results to be shown. The user can select any of the tab based on what action he needs to perform.

1. Check-in view



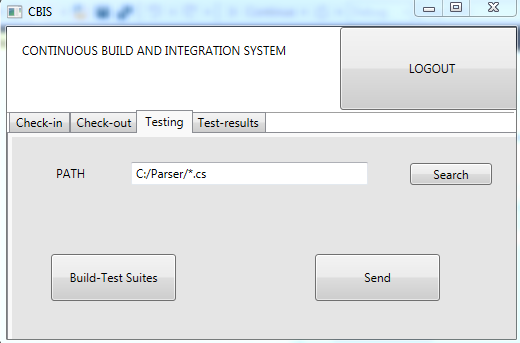
This is the check-in tab. As we can see, the user would be asked to enter the path where his files are residing which he wants to check-in. There also would be a search button, where he can search his local machine. After the user selects the path and clicks on the check-in button, the request would be sent to the server. If the check-in is successful, the check-in success message is displayed. If the check-in fails, the failure message would be displayed on the same tab. The messages are being displayed on the same tab to improve the user experience of the CBIS client.

1. Check-out view.



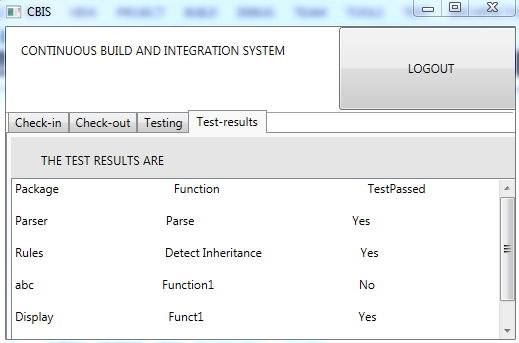
This is the check-out tab. It is very similar to the check-in tab with slight modification. Instead of giving the path of the local machine, the user would input the path of the server from where he needs to check out the files from. The user would be knowing where all the files are residing in the repository. This could be seen from the repository interface (we would be discussing this shortly). Thus, once the user puts in the server path, he click on check-out. If the check-out is successful, the display box would display what all files are being checked-out in a dynamic way. After the files are checked-out, the path would be displayed where the files have been checked-out. If the check-out has thrown some error on the server side, the corresponding message would be shown.

1. Testing view.



This is the testing tab. The user navigates here when he wants to have some files tested. In this path, he would specify the location of the files which he needs to be tested. The files are already on the repository, the files won’t be sent, but just the test suite would be sent to the test harness. When the user clicks the Build Test Suites button, the test suite would be built. After pressing the send button, this XML would be sent to the server. Mostly, testers would be concerned regarding this functionality.

1. Test results view.



This tab would be displaying the test results. The tab would be disabled at the start, only when there are some test results to be displayed, the tab would be enabled. The results would be displayed as shown above. The packages which are tested would be displayed, the functions of those packages would be displayed and whether the function passed the test or not would be displayed. Thus, this gives the summary of the running the tests. The user could come to know based on this result which functionality is failing and thus could make the necessary changes.

Above we have discussed the GUI the CBIS client would be having.

# **ADDITIONAL IMPLEMENTATION/FUNCTIONALITIES**

In this section, we would be discussing the additional functionalities/ implementation details of the CBIS system and features that could make the system continuous.

* Config.xml file present on each of the four entity of the CBIS system.

The configuration file would be present on the client machines, the repository server, the build server and the test harness server. This file would be having the IP address of all the entities present in the system. Using these address, the entities can talk with each other.

* How are sessions implemented?

WCF implements sessions for us and we don’t have to handle it explicitly.

[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession)]

When defining the service behavior , we can determine the activation policy.

* Milestones could be defined for the project and stored on the repository.

One very good implementation would be to define the milestones for each and every project initially, before the project starts on the repository. We can define the modules and which team is working on which module. Additional details like, the due time, when the module is due, the team members working on the module, everything could be defined. Thus, we can run a thread on the repository, which checks and sends notifications to the team members if a due date is approaching. Thus, we could automate this process which would be of great help in finishing the project on time and not violating the service level agreement.

* Additional logs, details stored on the repository.

The metadata of each and every package/module is very important. In this metadata, we could also store details like, who has last accessed the file, who has last checked-out the file, when was the file being modified, when was the file/package added to the baseline. These details would be helpful for security purpose. Also, in case of failure, we could back track the individual package’s activities and figure out the issue in case of any.

* Logging system on the repository, build and the test harness server.

Logging activities on the servers are very important. The server logs could help us understand the activities performed by the server, the number of requests, and any errors that might have occurred. These details are very useful for server maintenance. The server needs to be maintained, backed-up periodically, the server logs would be helpful in these scenarios.

* If a server is processing the client request, and if a new request comes-in, how can the server distinguish between the requests?

The message would be containing the address of the sender. Using this sender address, various requests are being distinguished.

* Could we extend the functionality to show the dependency outputs to the client?

Suppose a developer wants to check on which all packages does his package depends on? So, instead of just using the repository server for check-in and check-out purposes, we could extend its functionality and show the various package and type dependencies results to the developers, definitely after authentication. This could be a good addition to the repository’s functionality. Not many changes would be required, as the repository already has a dependency analyzer which is running and storing the results. We could use the same results to be sent to the developers.

* Caching is an important aspect of both the build server and the test harness server. The build server caches the recently built images, the test harness caches the recently tested test results? One question would be, should these caches be synchronized?

If you think carefully, the test harness and the build server almost caches very similar results. So, the synchronization of these caches would be a good design decision. The reason being, if an image is built, that image is sent to the test harness server for testing. If the testing is successful, the test harness server would cache the test results, also the build server would cache the image. Thus, both the caches would eventually be storing the same data. So, we could improve the performance, in a scenario, where if a tester is asking for the test results, if its not present in the test harness cache, that image would definitely won’t be present in the build server cache. Thus, the build server could instead of checking its cache, would directly send the request straight to the repository server. Thus, this could improve the performance a bit. Also, both the caches are using the least recently use algorithm for cache management. Thus, both the caches will be in synchronization.

* What exactly would be stored in the test harness cache? How can a build server cache entry and the test harness cache entry match?

Authentication which we are performing would be very useful in such scenarios. If a tester requests some modules to be tested, the cache could store all the module names, the test results for those modules, the image id which the build server sent, and the tester/user who had requested the test and when was the test carried out.

Struct cache

{

Int userid,

Int imageId,

Int timeOfTest,

String testResults,

String modulenames,

Int projectId,

String data

};

An advantage of this would be, if another tester/user runs similar tests, he could be notified about the last person who has run the same tests. Thus, if anything suspicious about the results could be discussed offline. The build server cache and the test harness cache could be matched using the imageId, which would be the same on both the servers.

Thus, this would be a developer friendly addition to the CBIS system.

* Test harness directory. The test harness would only be testing the images which have been uploaded in its fixed directory. Can there be a scenario where the build server uploads the images in the wrong directory?

No, the test harness would be only exposing a single directory where the build server could upload all the images. If the build server tries to upload the image/ access any other directory, the test harness would be sending an error message and it won’t allow the build server to upload to any other directory.

* How will the test dependencies be resolved? What if a test result depends on any other test result?

This is a very interesting scenario, where there is a test dependency, the output of one tests, could act like an input to the other. In such a scenario, there can be three cases. One the test which the current test is depending on has already been run. Thus, the test harness would be having the results prior to running the dependent test. There won’t be any conflict in such situations.

If both the depending test, and the dependent tests are running simultaneously, the depending test would have to wait until the dependent tests finishes running, only after this test has finished, can the dependent test complete.

This is a very intuiting scenario. This could happen when a package is depending on the results from another package. That package could not be present in the repository, might be because the developer has not yet finished developing that module.

Solution: A very simple solution to this would be, while designing the project milestones, the modules, main modules on which many other modules depend on should be given the bigger priority. Thus, all the important modules should be coded first. If this standard is followed, the above situation would rarely occur.

* When will the testing be done on the entire baseline stored in the repository? Is it feasible?

The testing needs to be carried out on the entire baseline, once a day. The perfect time for it will be the mid-nite hours when the load on the servers are far less as compared to the day time. This is necessary to keep the project up-to date with the recent tests and make sure that the baseline is behaving and giving the expected output. This is necessary to avoid any last minute surprises when the project is nearing the go live phase.

* Notification system, How can it be used in CBIS?

The notification systems would be very useful in the CBIS environment. If a test fails, or if the test gives erroneous results, the notification system could notify the developers on whose package/modules the test has failed. Thus, they can take immediate actions and look into the matter with priority. This will truly ensure the continuity of the project as, if any bug is to be found, it would/ should be replaced with highest priority. The other testers could also be notified, if any test fails, which will make them test a particular module in a more rigorous way. If a dependent module fails in the testing, the developers could have a meeting and discuss the issue before it exponentiates.

* What is the test file itself has been coded erroneously?

This could be a very genuine issue. This issue will be covered in the critical issues section.

* What if there is a change in the test file itself?

There can be a scenario where a test needs to be changed. How will this change affect the system? It won’t be affecting much. If the test file itself is changed, we can just add the new test file in the repository and let its source file point to this new test file. As the code is immutable, this would be a better approach, it’s also similar to the versioning system we had discussed before.

* How will the authentication system be set-up?

The repository would be storing a lot of projects. It is important that authentication be performed of the requests from the clients to ensure security. The authentication would be set-up initially, when the project is created on the repository. An authentication.xml file would be having all the valid usernames which basically are the team ids and their corresponding passwords. These values are decided at the very beginning. Based on these values, the authentication would be done.

* What exactly are team-ids.? How are they used in a production environment?

Usually, in production environment, each team is given the common team-id which we had discussed earlier. Using this team-id only the check-in and check-out could be done. This ensures that no random check-ins or check-outs are performed.

* What about scaling the CBIS system? Can it be scaled?

Scaling the CBIS system would be very easy because of the design of the system. A proxy is used by the client, or by any entity to connect to other entity. If suppose there are multiple test harness servers, the proxy could be connecting to all of those, get the results from the servers, build the results and then send it to the client. Thus scaling of the CBIS system is very easy. The test harness server could possibly be the most heavily loaded server. We could just copy the code and replicate it to create another test harness server. It would be having a different IP definitely. Thus, in the config.xml file, a new test harness server would be added and thus the client and the other servers would come to know about this new addition.

* Repository UI access? What exactly it is? How is it used?

Repository server would be the one storing all the project files. If we know the IP of the repository, we could see what all files are currently present on the repository. These files could be viewed from the web-browser. This is called as the repository UI view. Only the team leads and the project manager would be having this access. Sometimes, a developer would not like to check-out the entire package/module, but he just wants to see a small code implementation. This could be done by viewing the repository from the web, but obviously with the team lead’s permission. Also, this view is just the read-only view. The user won’t be able to make any changes in any of the files, just view them. This is helpful in many scenarios where an individual just needs to skim through the code, and not check-out the entire module.

# **USERS**

In this section, we would be discussing the user base which could use the CBIS system and how they would be using the CBIS system.

The CBIS system is used to support continuous integration and building of very large projects. Thus, this is a useful tool which would support large development teams involved in huge projects. The potential users of the system would be.

1. Software Developers.
2. Software Testers.
3. Project Manager.
4. Quality Assurance Engineer.
5. Software Architects.
6. Customers.
7. Maintenance Engineer.

Use case: Below is the general activities which all the users would follow.



The users would follow the above general flow.

1. The user would first start the CBIS client.
2. The user would login in.
3. After the successful login, he could upload the files to the repository, i.e. check in the code developed by him/her.
4. The user would select the path from where he wants the files to be uploaded to the repository and then checks-in the files.
5. The user will select the path from where he wants to check-out the code from the repository.
6. The user will view the checked-out files.
7. The user will also select the test files and create a test suite and send it to the test harness.
8. The user will check the test results displayed on the GUI.
9. The user will also see the various error messages/ success messages given by the server on the GUI.
10. **SOFTWARE DEVELOPER**

Software developers are given the responsibility of developing a part/ module of the huge project. Each developer works on its own independent machines. The code needs to be merged/ integrated step by step. The CBIS would exactly help the developers in doing so. After the developer has completed its part, the developer will upload its code to the repository. The code would be integrated into the baseline. Thus, CBIS would then test this integrated code. Thus, for the software developers, CBIS system would be used to integrate their codes, test the integrated codes and complete the project on-time. This ensures smooth delivery of the project.

1. **SOFTWARE TESTER**

The job of the software tester is to test the entire system as a whole. Not only that, they need to test the individual modules and the dependent modules. The CBIS system would them in doing the same in a very efficient manner. The testing part of the project has been separated on a separate test harness server. The testers would directly be contacting the test harness providing it with the test suites. The test suites would contain all the modules the testers need to test. The test harness would then carry out the function of testing and return the results to the testers. The testers could then check the results and see if any test case is failing. Ideally, the test case should not fail because, the check-in is not allowed unless the tests succeed. But the job of the testers is to test each and every possible values, extreme boundary conditions need to be tested. Thus, the test files need to have all the test conditions. The testers would be able to test the dependent modules too, as the repository server has this functionality included. The entire system would be tested once a day, i.e. the baseline would undergo testing as discussed. Overall, this would be a very useful tool for software testers and would easy the testing process. Notification system as discussed would add more functionality to the testing process.

1. **PROJECT MANAGER**

The job of the project manager is to ensure that the project is going on in a smooth way. All the teams are following the schedule so that the project won’t be delayed to the client to follow the SLA. The repository server, as discussed would be storing the milestones for each and every team. By checking the milestones, and the progress in a timely fashion could help the project manager in ensuring smooth running of the system. The project manager could also in a timely fashion, run the test cases and see the overall results of the baseline, as all these details would be logged into the repository server.

1. **SOFTWARE ARCHTECT**

Once the software architect creates a design, i.e. the architect of the entire project, he checks whether the system is been developed as expected. The code should be of high standards which needs to be given to the client. The architect could also check on the test results, he could check-out the code from the repository, he could analyze the test results, and if he finds out something not right, he could immediately contact the respective team, sit with them, discuss the design and re-design the system in worst case.

1. **QUALITY ASSURANCE ENGINEER**

The job of the quality assurance engineer is to ensure that a good high quality code is been delivered to the client. The CBIS system would help in doing the same. The quality assurance engineer could check out the code from the repository and analyze the code being developed. Whether all the coding standards are being followed. He could also check the statistics of each and every module from the repository. He could do the analysis on the test results, how many of the test results have failed over a period of time. All these analysis would help the quality engineer to analyze the project as a whole, in the real time system, i.e. as the project is been developed and help in ensuring good code development.

1. **MAINTENANCE ENGINEERS**

Once the project goes live, it is the job of the maintenance engineers to support the project. The client might face some issues which was not found in the development/testing phase. Many a times when a system goes live, issues are created on the system in real time. The maintenance engineer’s job is to fix those issues. The maintenance engineers, as it has the whole system, could take a look into the issues, make the corresponding changes to the system, and integrate with using the CBIS and test it using the test harness. Thus, this could largely help the maintenance engineers and could greatly reduce the resolution time of the issue.

1. **CUSTOMERS**

The customers could be considered as a secondary users. It might so happen that the client wants to know the status of a project. Using CBIS, the manager could just provide the walkthrough of the project, by showing the repository and running the integrated tests, ensuring smooth development of the project.

The extension for the repository of showing the dependencies would be very useful for all the users. The developers could tweak their codes and check the dependencies on/of their packages. The software architect could ensure that there are no cyclic dependencies of packages on one another. The quality assurance engineer could also come to know about the code quality based on the dependency. If there are more dependencies, there might be a necessity to change the code. The maintenance engineer could easily understand the project structure and ensure smooth maintenance of the system. Many more uses have already been discussed in Pr3-OCD. Thus, our CBIS system would be an ideal system for the IT world.

# **CRITICAL ISSUES**

While designing the system, we need to take into consideration the critical issues that might arise, and how will the critical issues be handled. The success of any system depends on how well that system addresses these issues. We would be discussing them in this section.

* **Cache coherency problem at the build and the test harness server.**

As the build and the test harness servers are maintaining caches, consider a situation where a package is changed, and its old image and test results are stored in the caches of the build and the test harness server. How will the system cope up with that? The answer is in the design of the system. As it is a continuous build and integration system, any changes in the package/module would be directed to the build server for building the images and the test harness server for carrying out the testing. Thus, the test harness cache and the build server cache would be refreshed and would be storing the latest images and the test results.

* **For how long the data should be in the cache of the build server and the test harness server?**

One design is, until the cache is full, we will keep the data in the cache. If a new data, data here refers to the images and the test results, arrives, the least recently used algorithm would be use to empty the cache. Another aspect of the cache could be, to define the Max\_Cache\_Time, the time for which the data would remain in the cache. If the Last\_Used\_Time value exceeds the Max\_Cache\_Time, then that entry should be removed out of the cache. Adding this constraint would allow a better cache management technique.

* **What if the output is huge? How should we map the test results data and display it to the client?**

There is a good possibility of a test suite returning a huge amount of result on the client side. A module, might be depending on many other modules. The design of the test harness is that, it would running tests on all of these modules which might result of generation of a huge number of test results. The problem might occur on the client end, as to how to effectively display this data to the user.

Solution: One solution would be to break the data module wise. Display each module’s test results separately. This will ensure that the output is in the readable format, and the tester or any other user could get good conclusions out of the same.

* **How will the repository server manage file contention?**

As the repository server, or other servers are multi-threaded, there is a good possibility of two or more than two threads accessing the same resource/file. If multiple threads are given simultaneous access to the same file/resource, it might result in wrong output.

Solution: As discussed, locking is the solution to file contention problem. Whenever a thread accesses a resource, it would lock the resource so that no other thread could access it. Once the thread is done with its processing, it would unlock the resource and other threads could use it. If a thread tries to use a locked resource, it would be blocked until that lock is released. Locks can be efficiently implemented in C#.

* **What if a client is trying to access the server, e.g. a client is trying to check-in to the repository server and the server is currently down?**

There might be a situation, where a client is trying to access the repository server to check-in or check-out the modules from the repository server, but the server is currently unavailable.

Solution: The client would try to establish the connection to the server. If there is no response, the client would wait for some time and try again. If the client gets no response, after a specific number of attempts, the client would display the appropriate error message of server unavailable at the moment.

* **What if the message size exceed the maximum limit?**

This issue might occur due to huge size of the output. The test results could be of huge number, if lot of dependent modules are being tested. These test results in a rare scenario, could exceed the maximum size of the WCF communication message decided while writing the service. How to handle such scenarios.

Solution: One of the solution would be, break the messages into many messages. The first message would contain a part of the result. The message would also indicate that rest of the result is in the following messages. The message would be combined at the receiver and displayed it to the user.

* **What about the security of the messages shared between various entities?**

As the messages between the various CBIS entities are communicated over the network, the security would be a major concern. How can the system implement secure communication between the various servers/clients.

Solution: One of the solution would be to encrypt the message before sending. At the start-up an encryption scheme could be decided which would be implemented on all the servers/clients. Before sending the message, the message would be encrypted. As all the entities have the knowledge of the encryption scheme used, the message could be easily decoded once it has been received.

* **What will happen if the communication between the client and the server is terminated?**

The connection between the client- server and any server-server could get terminated or dropped due to unexpected reasons. How will the communication be re-gained?

Solution: The solution is in the messages that have been exchanged. If for e.g. the repository wants to send the files to a client who is requesting a check-out, and if the communication breaks, the message won’t be delivered to the client. But, the repository would be having the client address who has requested the check-out. So once the communication channel is back and working again, the repository would be able to send the message. Thus, we could consider this CBIS system as a loss less system.

* **When should the server data be backed up, if it should be backed up?**

Backing up the server data is an important aspect. The reliability of the system increases if we have the back-up of the server. But another aspect would be the cost of the system. As we have three servers, backing up those servers would increase the cost to a great extent. If cost is not an issue, we can set up a job which copies the data from one server to another server. As this is a very heavy operation, we could set up that back up job to be run once a month.

* **Issues due to multithreading**

As the servers are multi-threaded environment, there could be a lot of issues which are related to multithreading such as race conditions, starvations, deadlocks.

Solution: The simple solution to these would be to implement the system in a very simple way, i.e. to keep the thread model as simple as possible. Only by writing a good multi-threaded code can these issues be terminated.

The above are the general client-server issues which could happen when more than one entities are involved.

* **Testing the entire baseline?**

The entire baseline needs to be tested, say once a day, to ensure the integrity of the code and check whether the current baseline is running as per the expectation. This is a very heavy task, and it needs to be carried out every night, say post mid-night. We can again use jobs to do this work. We would be heavily using such server running scripts referred to as server jobs to do such tasks. Thus, if the job has been successfully implemented, i.e. all the tests have passed, the job would pass, else the job would fail. In both the scenarios, notification would be sent to the team leads and the project manager on a daily basis.

* **Creation of metadata.**

We had discussed earlier that metadata would play an important role in the CBIS system. The metadata creation could also be assigned to a server job. This job would be running all the time, and it would log the results related to each file, i.e. how many times this package has been tested, who have last tested the package, how many test cases have failed. These data would be useful in understanding the overall quality of the development of the project and ensure a good quality code being delivered to the client/ or a good product has been developed.

* **What if the test file itself is wrong?**

Test files are also eventually source code, written by the developers/testers. It could also contain bugs. If a test is continuously failing, it could indicate two things, either the code is wrong, or the test code is wrong. Even if after rigorous testing, if a test case is failing, then there might be a situation where the tester code itself has been wrongly written.

Solution to that would be to change the test code, integrate it with the system and then again re-run the tests.

* **Time required to perform the testing?**

The test harness server also uses a high resolution timer, again provided by Dr. Fawcett to measure the time taken to run the tests. What if the time taken is too long?

If the tests take a lot of time to run, it means that the code has a lot of complexity because of which the test timing is increased. This calls for re-designing, writing a better simpler and less complex code, an optimized code which runs quickly and correctly.

# **CONCLUSION**

This document described the architecture of the continuous build and integration system. We described each of the four entities of the system, i.e. the repository server, the build server, the test harness server and the CBIS client. Each of these were described in detail and how the implementation would occur. We also described the various multi-threaded environment of the servers, the partitions of the servers. Also, few server jobs which would be running on a daily basis to collect the data were also described.

The communication system between the entities were described, which included the message catalog. Some of the parts were very similar to the Pr3- OCD. References to that were made whenever necessary and only an overall view of that topic was discussed here, for e.g. the dependency analyzer. The prototype for the client GUI was described in detail.

Few enhancements to the base CBIS system were also discussed. The users and the critical issues along with the solution were also discussed. Thus, to conclude, CBIS as the name suggests supports continuous building and integration of the projects which is very useful in a development environment. It runs test cases on integrated codes, checks the authenticity of the user, stores useful data in metadata file which could be used by high level managers to know the health of the system being developed.

Thus, CBIS is a very useful tool in the IT industry specially when then the complexities of the applications are growing day by day along with the team size and the code size.

# **REFERENCES**

1. Project 5- CBIS requirements, provided by Dr. Fawcett.
2. Notes from <http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/CSE681Note7.htm> - Dr. Fawcett’s website.