

**CSE 681- Software Modelling and Analysis**

Project #5-Fall 2014

**Continuous Build and Integration System (CBIS) – OCD**

Document Version 1.1

By

Arunbalaji Sivakumar

SUID: 594227528

Graduate Student, Computer Engineering

Syracuse University

Date: December 08, 2014

Instructor: Jim Fawcett, PHD

Table of Contents

[**1.** **Executive summary** 3](#_Toc405846068)

[**1.1** **Introduction:** 3](#_Toc405846069)

[**1.2** **Overall architecture of the Continuous Build and Integration System:** 4](#_Toc405846070)

[**1.3** **Critical issues:** 5](#_Toc405846071)

[**2.** **Repository server** 6](#_Toc405846072)

[**2.1** **Concept:** 6](#_Toc405846073)

[**2.2** **Users and use cases:** 6](#_Toc405846074)

[**2.3** **Package diagram:** 7](#_Toc405846075)

[**2.4** **Activity diagram:** 11](#_Toc405846076)

[**2.5** **Critical issue:** 14](#_Toc405846077)

[**3.** **Build server** 16](#_Toc405846078)

[**3.1** **Concept:** 16](#_Toc405846079)

[**3.2** **Users:** 16](#_Toc405846080)

[**3.3** **Tasks:** 17](#_Toc405846081)

[**3.4** **Activity Diagram:** 18](#_Toc405846082)

[**3.5** **Package diagram:** 19](#_Toc405846083)

[**3.6** **Critical issues:** 21](#_Toc405846084)

[**4.** **Test harness server** 23](#_Toc405846085)

[**4.1** **Concept:** 23](#_Toc405846086)

[**4.2** **Users:** 23](#_Toc405846087)

[**4.3** **Task:** 24](#_Toc405846088)

[**4.4** **Package diagram:** 26](#_Toc405846089)

[**4.5** **Activity Diagram:** 28](#_Toc405846090)

[**4.6** **Critical issues:** 29](#_Toc405846091)

[**5.** **Clients** 32](#_Toc405846092)

[**5.1** **Concept:** 32](#_Toc405846093)

[**5.2** **Users:** 32](#_Toc405846094)

[**5.3** **Tasks:** 33](#_Toc405846095)

[**5.4** **Package diagram:** 36](#_Toc405846096)

[**5.5** **Activity diagram:** 38](#_Toc405846097)

[**5.6** **Critical issues:** 39](#_Toc405846098)

[**6.** **Prototype** 40](#_Toc405846099)

[**6.1** **Metadata generator:** 40](#_Toc405846100)

[**6.2** **Caching in build server:** 40](#_Toc405846101)

[**7.** **References:** 41](#_Toc405846102)

# **Executive summary**

## **Introduction:**

When a new project is being developed the software development team must have all the required tools to develop, store, build and test codes in a centralized environment so that the overall project is always kept integrated. When a new module is developed it must be placed in that central environment in such a way that it does not compromise the integrity of the whole project. To achieve this Continuous Build and Integration System (CBIS) is used to store and process a large amount of source codes either publicly or privately. CBIS is mostly used in multi-development project where different development groups work on different parts of the project and these parts are continuously integrated as the project developers.

Continuous Build and Integration System will not only build and store the files but also store information regarding the development process by means of having a metadata for every file that is uploaded in the CBIS. The information from the metadata can be used to measure the overall performance of the development process. Apart from metadata, dependency analyzer is also used to keep track of the dependencies among files stored in the repository.

This tool can help developers and testers to write and test codes effectively so they one code or test case does not conflict with the others. An important advantage of using the CBIS is that the codes within CBIS are always working and integrated and have no errors or conflicts among themselves. So developers should only worry about codes that are currently being developed and not on previously developed codes.

The main users of the CBIS tool are the following,

* Software developers
* Developer team leads
* Testers
* Test team leads
* Project manager
* Quality Assurance team
* Software architects
* Release team

## **Overall architecture of the Continuous Build and Integration System:**

The Continuous Build and Integration System uses a client-server architecture where the server performs the storage, build and test process and client performs the development process. The servers process the request from the clients such as check-in, check-out search files in the repository, find and analyze the dependency relationship between files under CBIS. The client uses a Graphical User Interface (GUI) which is implemented using Windows Presentation Foundation (WPF). The GUI displays the contents of the repository to the clients. It is also responsible for establishing the communication between the server and client. The communication between the server and client is made using the Windows Communication Foundation (WCF) which uses message passing mechanism for transactions.

The CBIS is the combination of servers and client which contains the repository server to store all code which are correct, build server which compiles the codes being checked-in to the repository under the request of other servers and clients, test harness server which perform various tests on the checked-in codes and the client where the development of the codes take place.

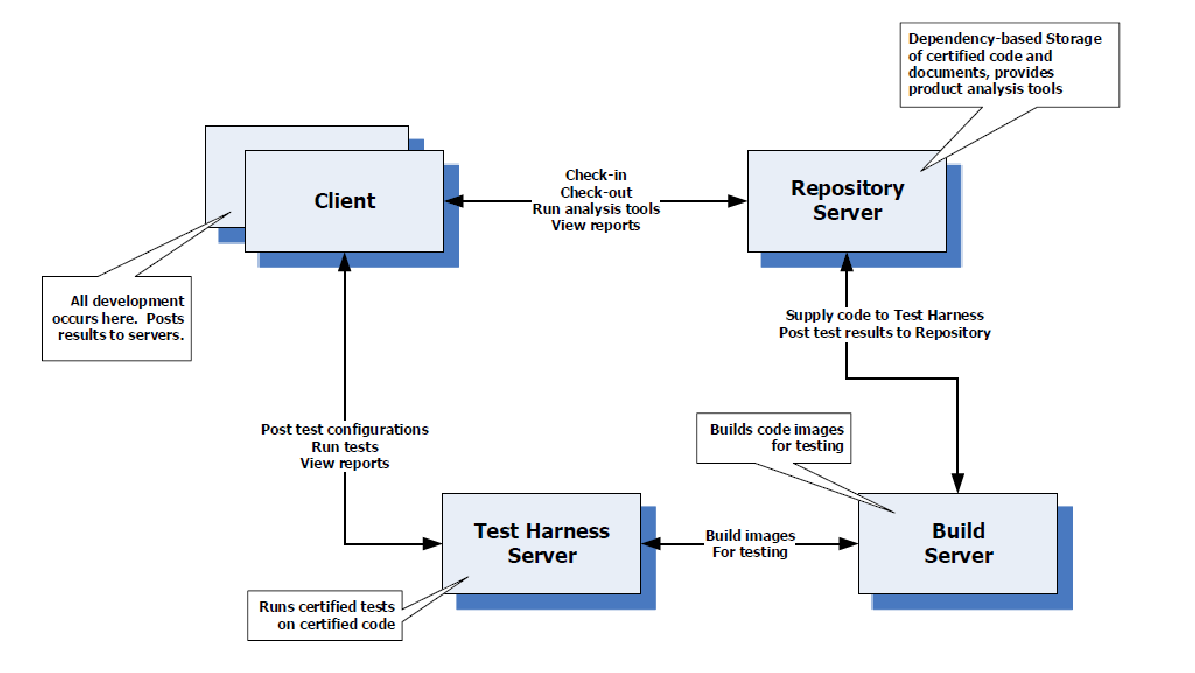


Figure 1.2.1 Continuous Build and Integration System

The document is mainly partitioned into following sections:

1. Repository server
2. Build server
3. Test harness server
4. Client
5. Prototypes

## **Critical issues:**

Here are some of the critical issues that will discussed in the later part of the discussion.

**Repository server:**

* What if the dependent files of the checked-in file are not available in repository server?
* What happens if the load balancing is very high?
* What happens if the transferred files are lost?
* What happens if two different client check-in different files with same file names?

**Build server:**

* What happens if the build server does not run?
* What happens if build time is longer than usual?
* How to debug the build server?
* What happens when the build server is upgraded?

**Test harness server:**

* How will the test harness test itself? Can the same test harness tool be used to test itself?
* What if the test cases become too large?
* What if the DLLs requested by the client does not satisfy the test harness interface?
* Can test be run on a single thread or multi-thread?

**Client:**

* What happens if the client’s system has a different version of DLLs that is not present or supported by the CBIS?

# **Repository server**

## **Concept:**

The repository server is the central server which is accessed by all the remaining servers. All developed codes, test logs, documents are stored in the repository server. The repository is always busy so it is important to handle load balance. The repository server has client-server architecture where the server is responsible for processing the request from the clients. It handles check-in, check-out, extraction and searching of text or metadata information from the repository. The server also manages the communication with the clients and routing requests.

## **Users and use cases:**

The following users interact with the repository server in various ways.

* **Software developers:**

As the project gets build up the developers have to deal with a large number of source code files. This tool is extremely useful for developers since they would need to search the code repository with keywords in order to search files and find specific functions. They can also use the parent child relationships to files to understand the dependencies of code files.

* **Project managers:**

The project managers use status reports and result logs which show the progress of the projects regarding the resources allocated to a particular project. The other servers use the code files on the repository to develop these reports. The managers can also perform code inspections where they search the repository for source code files and get its metadata file to view the last modified date and name of the developer who accessed it.

* **Continuous integration team:**

Continuous integration team is basically involved in creating build and deploying project on application servers. They take the code from the repository server and use the build server and the test harness server to build and test the codes and any identify bugs or dependencies between the source codes, then they can use the metadata file to contact the concerned developers about the report from the tested files. This tool is extremely useful to them to understand the dependencies of various files and metadata information of these files.

* **Testing team:**

The testing team perform tests on the codes in the repository server. As soon as the new check-in happens in the repository, testers carry out the integration testing on the code.

## **Package diagram:**

Depending on the activities performed, the tool is made up of the following different packages or partitions. The package diagram describes the dependencies of every packages on another.



Figure 2.3.1 Modules of repository server

* **Dispatcher:**

The dispatcher is responsible for interpreting the requests received from various other clients and servers and dispatching them to the appropriate modules for processing. It goes through the message and identifies the type of request by comparing it its message catalogue and then invokes the appropriate package. It has a shared queue which distributes the request to the desired module. A single thread is allocated to each request that comes out of the queue. If the queue is full then further incoming messages/ requests are dropped and an error message is sent to the sender.

* **Receiver:**

The receiver module is responsible for extracting the message from the service and sends it to the dispatcher module. The message is usually in XML format. It contains a shared queue. Whenever the repository finishes the clients request it enqueued it into the queue and the message is sent to the corresponding client via the communication channel.

* **Sender:**

The sender module posts the message received from the packages of the repository to the communication-service. It also performs chucking of large files so that chunks to one client is not mixed with the other client. Usually its l=the files that were requested by the clients are sent from the repository server. The sender has its own shared queue. Some results converted into XML file and sent to shared queue and then dequeued to the communication- service.

* **Communication-Service:**

The communication service is responsible for creating and maintaining an asynchronous communication between the server and the clients. It implements the HTTP Service for the communication. Whenever a new client tries to create a connection between the serve, the communication-service creates a new thread which has its own blocking queue and message passing is done using this blocking queue.

* **Navigation module:**

This module is responsible for creating and maintaining the dependencies between various files in the repository. When a file is checked-in into the repository the dependency relationship is of that fil is updated for all the files that are dependent on this file. The dependency relationship or dependency map basically stores the parent-child relationship between the filed using the dictionary data structure. The dictionary should be kept persistent so that parent-child relationship need not be obtained from the metadata file.

* **Check-in module:**

The check-in module handles the check-in request from the clients (other servers). Every file inside the repository which are the part of the project baseline has the metadata file which contains a version number, last modified date and the developer who last accessed and modified the code. Whenever a user wants to check-in a modified file/code, the version number of the file is incremented and a new metadata associated with this file is created. The older version of the same file is retained. The older version Meta data can be used by the developers to access the old configurations for the files that are still in service and get back to the older version if the new version code is incorrect and has a lot of bugs. All the dependent files will still point to the older version of the checked-in file and they are updated whenever they are ready to be integrated to the new version. A notification is also sent to all the parent files and the developers, who last accessed those files, about the new version being checked-in. The check-in has intermediate state called the open-check-in. when a file is checked-in to the repository, it is placed in an open-check-in state. This state indicates that the repository is waiting for all the dependent files of that file have not been checked-in. the file moves from the open-check-in state to the check-in state once all the dependent files are also checked-in.

* **Check-out module:**

The check-out module is invoked whenever the repository server gets a check-out request from the clients (other servers). Check-out is basically to transfer the requested file to the client and expecting that client to check-in the modified version of that file. The server maintains the list of users who have checked-out and the files which they have checked-out from the repository. File chunking is also done while transferring large documents. To prevent concurrent access of the same file the repository keeps a lock policy which locks the file while checking-out and no other user can check-out until the lock is released.

* **Authentication module:**

The authentication module handles the authentication activities of the repository. Whenever the user needs to access or submit the files he/she should authenticate being getting the access the tool. If the user fails to authenticate then an authentication error message is sent to that user. If there are multiple failure then a notification is also sent to the leader who is in charge of that particular part of the project.

* **File manager module:**

The file manager module is responsible for retrieving the files requested by the clients. It performs file chunking to merge received files and putting them into the desired files. It uses the .Net framework for file handling mechanism.

* **XML generator:**

The XML generator module helps in building XML documents. Notifications are sent to clients using the XML generator in XML format.

## **Activity diagram:**



Figure 2.4.1 Activity diagram for repository server

The overall repository server works as follows.

1. Repository server initializes its server objects and starts listening to any connection from the clients and other servers.
2. When a message is received from the client, the server interprets the message and dispatches it to the concerned packages or modules.
3. If the users sends authentication request using his/her user ID and password, the server matches it up with the entries present in the hash table in the repository server which maintains the user IDs and password of the entire project team.
4. If the user wants to view the file along with its dependency files then an information view request is received by the repository server and the server gets the desired file and its metadata from the repository. It then performs a dependency analysis on those files. The information is then sent to the client.

**Check-in process activity diagram:**



Figure 2.4.2 Activity diagram for check-in process in repository server

When the user wants to check-in his/her modified code back to the repository, a check-in request and the modified file is sent to the repository server. The modified code and its corresponding metadata file is given a new version number to differentiate it from its old version. It checks if the code is versioned or not. If there is no version number allotted to it then it identified as a new code and a new version number is assigned to it and placed in the repository. It the code is already has a version number then that version number is incremented to +1 and the same thing happens to its metadata file. The lock that was placed on that file when it was checked-out is released. The recently checked-in files are not yet part of the project’s baseline. The repository server keeps a list of all those files so when the test harness server requests those files they are sent through the build server. Only when the test harness successfully executes those code they are integrated into the baseline, or else the error notification is sent to the concerned client and to the repository server. If the repository receives any error message then that corresponding codes are marked faulty and the older version of the codes is used for any future check-out or extraction.

**Check-out process activity diagram:**



Figure 2.4.3 Activity diagram for check-out process in repository server

When the user wants to check-out the file, the server identifies if that user has an ownership on that code file. If that user is not the owner then the user can extract a copy of that file but cannot check-in with an updated/modified copy of that file. If the user is owner or he/she is in group which has the ownership on that file then that user can check-out. After the file is checked-out a lock is placed on that file so that others can only extract the file but cannot check-out.

## **Critical issue:**

1. **What happens if the load balancing is very high?**

Description:

As the overall project starts to progress the repository server will be more frequently be accessed by the various team of the project. In such a case the load on the repository server will be very high it may crash the server or make it unresponsive. The repository server can become the single point of failure in the system.

Solution:

Load balancing needs to be carried on the requests handled by the repository server by distributing the request across distributed servers. Thus by distributing the requests across multiple servers the repository will no longer be the single point of failure. For example, if one of the application server fails, the load balancer will simply direct all new application requests to other available servers. But replicating the data on these servers and keeping them in sync at any point of time will be a huge challenge.

1. **What if the dependent files of the checked-in file are not available in repository server?**

Description:

When files are checked-in to the repository the dependency between that file and the files in the repository server must be identified and maintained. If the dependency is not found then the integrity of the code might breakdown.

Solution:

A policy can be added that states that dependent files need to be present before check-in to the repository is made. The open-check-in state discussed under check-in process above will make sure that only when the dependent files are present in the repository the file state will move from open-checked-in to checked-in state. This will help the user to add the dependent files to the repository when the files are ready.

1. **What happens if the transferred files are lost?**

Description:

Being a server-client application there are always a chance of interruption during uploading/downloading from/to the server which might result in loss of data.

Solution:

A checksum can be maintained at the client side and the server side to verify if there was a loss of data during transaction. If so the client/server can request the other to resend the lost data.

1. **What happens if two different client check-in different files with same file names?**

Solution:

The file name can be prefixed with the owner’s name and his/her group name along with the file names.

# **Build server**

## **Concept:**

Developers check-out the codes from the repository server, work on them, modify them based on the requirements and add the updated/modified codes back to the repository server into the project baseline. When they do so there is a very high possibility that the updated/modified codes to cause problems to the overall system. So every time when the developers check-in their updated codes they should make sure that their codes can efficiently integrate with the rest of the codes in baseline without any error. Build server makes sure that the new codes does not break the overall structure of the system. Every time when there is a check-in made to the repository server the build server gets hold of that code and along with all the other codes that are dependent in that code (latest codes obtained from repository) and starts building that code. It also compares the build results with the previous successful build results of that code which was cached in the repository. Only if the build is successful the code is actually checked-in into the overall project’s baseline. If there is any build error or warning the build server will send an error notification to the respective developer listing all the errors and warnings and logging a report to the cache which holds the previous version of the code.

## **Users:**

The following users interact with the test harness server in various ways.

* **Developer team leader:**

Developer team leaders are one primary users of build server. When a project is being developed team leaders are responsible in making sure that their part of the project is progressing in the right phase. This is handled by adding required versions of right software and DLLs into the build server. Using the build server the team leader can know the progress level of every developers in under their team.

* **Quality Assurance Analyst:**

QAA is responsible for maintaining the integrity and quality of the overall project. If the modified codes tend to use any outdated software or DLLs then they cannot be properly integrated into the overall project. QAA can oversee and limit the projects capability to and prevent the entire project from crashing.

## **Tasks:**

* **Caching:**

Whenever a code is checked-out and checked back in the build server must make sure that the code does not have any errors and it does not breaks any dependency relations. So whenever a successful build happens the build server makes a list of all the modules that were dependent on this code. So when the code is modified/updated again it can perform analysis on whether any dependencies where broken.

* **Build request from Test Harness Server:**

When developers wants to test a code that they had modified they send the code to the Test Harness Server. Before the code is tested the THS sends a build request to the build server to compile the code and check whether the code integrates with the rest of the project. The build server will run the code under the desired software and DLLs along with all the dependent modules of that code. If the build is successful then the build server will the build image to THS that the code satisfies the overall project requirements and it can be tested. If the build fails the build server will send an error report listing all the error along with the information of why those errors occur. If there are continuous build errors on the same code then the corresponding team leader is notified about those errors.

* **Continuous build of Source Code Modules:**

It is the duty of the build server to maintain the overall project at the right phase. Whenever the build server is updated it gets a copy of the entire project from the repository and compiles them to check whether they are properly integrated with the build’s requirement. If the build fails then it send an error report to the respective developers to modify the code based on the requirements. If the build succeeds it updates the codes in the repository along with list of which modules are dependent on which modules.

## **Activity Diagram:**



Figure 3.4.1 Activity Diagram for Build Server

The build server performs the following activity.

* The test harness server requests the repository server for any recently checked-in codes through the build server. If there are any such files the THS will ask the build server to compile those file and build images on the compiled files.
* The repository server sends the recently checked-in code along with all files that have dependency on that file to the build server. The repository server gets the dependency list based on the user’s check-in information which lists all dependency and by running the dependency analyzer of that code.
* The repository also sends required libraries to perform compilation.
* The check-in code and dependency codes are integrated and compiled by the build server.
* If the build is successful then the build image and all the codes are sent to the test harness server.
* If the build is not successful then an error message is generated and sent to the concerned user. A report is also sent to the repository to remove that code from the baseline.

## **Package diagram:**

Based on the tasks designed for the build server the following packages can be identified in the application.



Figure 3.5.1 Package Diagram for Build Server

* **Receiver:**

The receiver gets the build request from test harness server when a modified code is to be built before being tested or recent codes from the repository server that were changed since the last build.

* **Dispatcher:**

The dispatcher handles the conversion of messages from the receiver into requests in the format in which the build server can interpret. It sends out any request message through the sender.

* **Sender:**

The sender sends message requested by the build server to the requested server/client. The sender sends message request to the repository server asking for the source code modules that were recently modified. It also sends out the build results to the test harness server and repository server for log information.

* **XML reader/Writer:**

When the code is compiled the build server requires a list of DLLs, and other dependent modules. This information is sent to the build server in a valid XML format and the server can get the required modules from the repository using this module. After the compilation is done it sends the error report or success report back to the servers in XML format using XML reader/writer.

* **Dependency analysis:**

The dependency analysis module handles all the analysis which needs to be done to identify whether the modified code has proper dependency with the other modules. It checks for data files and other codes without which the code cannot run.

* **Compilation:**

Compilation module runs the code under the desired DLLs and identifies if the code breaks and if there are any bugs during compilation. If the compilation is successful then a results are sent to the desired servers and if any error is encountered an error report is generated and sent.

## **Critical issues:**

1. **What happens if the build server does not run?**

Description:

The responsibility of a build server is to check whether the project is at the right path. There are many possibilities that the code that the build server runs can crash the build server. In such case the developer/ tester will be able to identify whether their code can properly integrate into the project.

Solution:

The team leaders and the Quality Assurance Analyst have to keep track of the performance on the build server. If they identify that the build server has crashed then immediate action should be taken to restore the build so that overall progress is not delayed.

1. **What happens if build time is longer than usual?**

Description:

As the number of lines of code keeps increasing during the project development the build server is run frequently. The build server has to make sure that each and every line of code is correct by running more test and provide legitimate output. This may result in longer build time which will lag the development progress.

Solution:

* The build server must kept upgraded.
* The build can be modified to support parallel build keeping in mind about the implicit dependencies. This can be handled by distributing the build process across different build machine with each handling a specific build action.
* Not every code needs to be built, only those codes that were recently modified/updated should be built again. This reduces the overall build time.

1. **How to debug the build server?**

Description:

Codes in build servers are not any different from ordinary codes. Sometimes they also have errors. This may lead to not identifying all the errors in the codes which are to be built resulting in poor build results.

Solution:

Developers and testers should be notified when the build server is being debugged so that they would not post codes to the build server. Various diagnostic tests should be run on the build server and make sure that the bugs are fixed.

1. **What happens when the build server is upgraded?**

Description:

It is not possible to have the same build server from the start to end of the project development. As the complexity of the project and its codes increases the build server should also be updated so handle those complex issues. If it is upgraded then all previous build process and result will not be valid.

Solution:

After upgrading the build server the previous version should be kept so that all necessary results and logs can be used.

# **Test harness server**

## **Concept:**

Test harness is a software or a collection of software designed to test a single or a set of codes under various test conditions (which are usually pre-defined), and identify whether they affect or improve the existing and working programs. When a software is being developed by a group of developers each one of them checks out different files or the same file (rarely), modify them and checks them back into the repository. After these codes are checked-in, their interaction with the other dependent files may be affected due to the changes made these recently checked-in codes. So the entire functionality of the code must again be validated after every check-in. The test harness tool allows the developers to test the functionality of their code before integrating them into the system. So whenever a bug is identified, it is fixed and the test case that exposed the bug is recorded and run again after every subsequent check-in of the codes. Test harness tool can be used whenever the user wants to add a new functionality or feature into the application.

## **Users:**

The following users interact with the test harness server in various ways.

* **Test Engineers:**

Testers are the primary users of the test harness server. They perform tests on the code specified by the team leader. Their duty is to find any bugs in the code and report it to the developers of that specific code. Testers usually do not have all the required codes during testing, so based on the available code which were pre-tested and previous test logs obtained from the repository server, they integrate the current code with them and find whether the code or the software meets the required technical, functional and security requirements. They also keep track of various test runs and store any bugs generated by that code and re-run the code again after future modifications. All results or faults that are obtained during these tests are logged in the repository server.

* **Developers:**

One of the important work of developers, apart from developing and updating the code, is to collaborate with the testers in developing various test cases that could break recently checked-in code and solve previous bugs or any new bugs that could break the overall system. Developers can also update the test cases which were written earlier for the application under test. Developers can integrate their newly checked-in code with the dependent codes in the repository and check if their behaviour is same even after updating the test cases.

* **Test team leader:**

The test team leader keeps track of all the test runs made by his/her team who used the test harness tool. The test team leader can also monitor the progress by looking into the outcome of the tests run by the tool.

* **Quality assurance analyst:**

The quality assurance analyst monitors the entire testing process handled by the test engineer and the developer. The QAA is in touch with the test team lead and developer team lead to make sure that the codes in the repository server are fully functional and reliable by identifying the weakness in their codes. QAA also notifies any deviations noticed in the testing process implemented by the other team.

## **Task:**

This topic describes the various functions handled by the test harness server.

* **Caching:**

The test harness server always keeps a cache of all compiled modules in repository server. Whenever a code is run in the test harness, any bugs or results is identified and logged in to the cache, so that when the updated code is re tested, it can be compared with the previous logged entries. All these cached modules have their own metadata containing the date and time of the last run in the test harness tool along with the information of all the other modules that were dependent on this module during the test. The successfully tested and cached modules are always a part of the baseline project.

Caching is mainly handled in the repository server and can be accessed by other servers. Testers and developers have their own cache folder in repository which stores all the previously compiled code’s bug report and log results. They can access their cache and verify the result of the updated codes with the previous logged results.

* **Test suites:**

The test suites is an XML file that contains the list of test cases that are used when testing in the test harness server. It is the duty of the test harness server to make sure that the overall project is working in the required manner. The test suites contains various test cases used to test the developed codes and identify if they are not behaving as required. These test cases have a list which contains the expected output when that test case is used. The testers can use different test suites based on the codes that they want to test. Usually the test suites start from the code that is being tested and expands to all the codes that are dependent on this code and covers the module or the entire project.

The test suites are created and managed by the test team leader by collaborating with the develop team leader. They have overall idea about how the project should behave and which part of the module and the project is more prone to failure. They can add new test suites into the test harness or remove any test suites if they find it be not useful. Test suites should be efficiently handled as they alone can identify loop holes and error prone cases in the codes.

* **Testing and notification:**

Whenever a new code is checked-in to the repository the test harness gets hold of that code through the build server. The test harness sends a request to the repository to provide the recently checked-in code along with all the dependent files of that code through the build server. When the repository sends the code the build server compiles those codes to identify if there are any build errors. Once the build is successful the code is sent to the test harness to be tested. The test harness performs various tests on these codes using the test suites. These test suites are either provided by the developer or obtained from the Meta data file of the code which contains all the test cases or test suites that were used in previous tests. The test results are also compared with previous test results (if any) and expected output. A notification is sent in a XML file listing all the test cases used and the result. If there are any errors then an error notification is sent to the person responsible for that code and other developers whose codes were dependent on this code and to developers whose codes that this was dependent on. The notifications are logged along with the codes for future reference.

## **Package diagram:**

Based on the tasks designed for the test harness server the following packages can be identified in the application.



Figure 4.4.1 package diagram for Test Harness Server

* **Receiver:**

The receiver gets the recently checked-in codes from the repository server through the build server. The receiver receives both the codes to be tested and the build image of that code which is generated by the build server. The receiver has its own blocking queue to receive and pass it to the dispatcher.

* **Dispatcher:**

The dispatcher handles the conversion of messages from the receiver into requests in the format in which the test harness server can interpret. The dispatcher sends the successfully compiled code to the tester and the build images to the build module.

* **Tester:**

The tester module is primary part of the test harness server. Whenever a new code/file is checked-in to the repository server they have to be tested so that they can be integrated into the project’s baseline. The repository sends the recently checked-in files using the recently checked-in files list maintained by it. The tester also receives the build images through the build module. If the build image has error/bugs then the corresponding file is not tested and an error report is sent to the owner of that code. If the build image does not have any error then the tester starts a new application domain in the loader module by creating a new child thread and sends the image and the code to that application domain where the testing takes place. The tester also sends the test suites mentioned by the client and the default test suites allotted to that file to application domain in loader. Once the result is generated the tester sends the result log to the client. If there is any error then an error report is sent. The testing must only be performed on the loader module so that if the application domain crashes the tester can unload and load a new application domain using a new thread. This prevents the test harness server from crashing due to faulty codes.

* **Build module:**

The role of build module is to interpret the build image sent by the build serevr and send it to the tester for testing.

* **Test cases module:**

Test case module contains the collection of all test suites along with various levels of testing mechanisms. These test suites are placed by the test team and develop team leaders which is discussed in test suites under tasks section.

* **Loader module:**

The loader module is the place where the actual testing takes place. Only those codes which were successfully built can be tested. The loader work is to run the code in an application using the build images and the codes. The loader also gets the DLL files from the tester. Once the child application domain is built inside the loader various test cases sent by the tester are used to test the codes. A final report is generated based on the analysis generated by the loader and sent to the tester.

* **Sender module:**

The sender module sends the messages passed by the tester to the clients and to repository server. The communication is done using communication channel.

## **Activity Diagram:**

This section discusses the main activities associated with the test harness server.



Figure 4.5.1 activity Diagram for Test Harness Server

As discussed in the package diagram the test harness server performs the following activity.

* It requests the repository for recently checked-in files.
* The repository has a list of all recently checked-in files and it sends those file one by one through the build server.
* The build server generates the build image for the requested files and sends them to the test harness server.
* The test harness tool verifies whether the build image has any errors if so then the error message is sent to the owner of the code.
* If the build is successful then the codes and build image is loaded for testing.
* The testing takes places in the monitored environment so that any effect will not affect the integrity of the overall system.
* Various test suites appropriate to the file is loaded by the test harness tool and test suites mentioned by the clients are also added.
* Once the test is complete and if are no error then a notification is sent to the repository informing that the checked-in code has no errors and it can be successfully integrated in the project’s baseline.
* If there are any errors then an error message is sent to the owner of the code and also to the owners whose codes are dependent on the tested code. The error message is also sent to the repository along with the list of errors and test suites used for logging.
* If there are multiple errors on the same code files then the project team leader is notified on this issue.

## **Critical issues:**

1. **How will the test harness test itself? Can the same test harness tool be used to test itself?**

Description:

The quality of the overall project depends on efficiency of the test harness server. It is the essential part to test the quality and integrity of the project. If the test harness itself has bugs and other issues then the outcome of the project will not be as expected.

Solution:

The test harness should be able to test itself by creating the test class and giving this class’s path to the test harness tool. By doing this, the test harness will be considered as a normal project on which various test cases are applied to test its efficiency. However, there should be a stable version of test harness which has no issues. This stable version can used to test the new version of test harness tool.

1. **What if the test cases become too large?**

Description:

As the project development progresses more and complex test cases are need to test the codes. As the number test per day keeps increasing the performance of the test harness will get affected.

Solution:

The test team leader should discuss with the developers to identify and eliminate the redundant and undesired test cases from the test configuration. The test harness can be used to compare which of the two redundant test cases is more effective and that test cases can used further for testing.

1. **What if the DLLs requested by the client does not satisfy the test harness interface?**

Description:

Not all DLLs tested to be used in the test harness server. This may create issues while testing different version of DLLs.

Solution:

The DLLs can be divided into tested DLLs and untested DLLs. The untested DLLs can be used when testing normal cases of codes. There are certain codes that form the critical part of the project like security, integration, etc.., in such codes only tested DLLs should be used.

1. **Can test be run on a single thread or multi-thread?**

Description:

As the test cases and codes becomes larger it is easy if the tests are distributed across multiple system. However this may lead to few overheads.

Solution:

Testing codes in threaded application will cause any damage if they are managed properly. Multi-threads can be used for complex codes and single-thread can be used for simple codes.

# **Clients**

## **Concept:**

Client can be a developer or a tester who uses the Graphical User Interface (GUI) which is implemented by Windows Presentation Foundation (WPF). The GUI is divided into several parts to provide different functions like check-in, check-out, test harness and project report.

## **Users:**

The following users interact with client GUI in various ways.

* **Developers:**

The developers are one of the primary clients who use the client GUI to communicate with the other servers. Whenever a new code is developed or a code is modified they have to integrate it with the rest of the project’s baseline. Multiple developers will be working of the same part of the project so each one of them should make sure that their code will not conflict with the others. The client GUI provides an easy way to make sure that their code is proper. The developers can check-out their code from the repository and check-in their modified code back to the repository. They can also use the client GUI to test their code using the test harness server. They view the result log of their code through the client GUI. Each developer is given a user ID so that only those are part of the specific project can access and modify the code. Developers who are not part of a code /file can extract the code/file but cannot check-in back.

* **Testers:**

Tester also frequently use the client GUI to perform various test on the codes in the repository. They can select various test cases and DLLs to be used to test the codes. The test result logs and previous bug reports stored in the repository can be viewed by the testers to understand the issued in their testing code. Similar to the developers the testers are also given unique user IDs so that only testers who a part of that code/file can check-out and perform various test and check-in.

* **Project team leader:**

The project team leader is responsible for making sure that their part of the part of the project is at a steady state. They use the client GUI to access their codes, analyse codes, logs and dependencies.

* **Manager:**

The manager cannot often go through the entire project code. They can use the client GUI to request any files and view their logs. The manager can keep a constant watch on the progress of the project by interacting with the client GUI.

## **Tasks:**

This topic describes the various functions handled by the client GUI.

* **Check-in:**



Figure 5.3.1 GUI for client to Check-in

* The user must authenticate himself/herself before checking-in the file to the repository. If the user tries to check-in without proper user ID then the entire check-in process would file after the check-in button is pressed.
* User can browse the file or enter the file name manually to check-in the file into the repository server.
* User should provide description about the file to be checked-in. This is important when the files are viewed by other users.
* User should provide list of dependent files of the check-in file or else the check-in file will be in the open-check-in state.
* Keywords are used to search the required files quickly later.
* User has to generate a metadata file for the checked-in file. This is done using the generate Metadata button.
* The status message is received when the file is successfully checked-in.
* If there is a failure in check-in process an error message is displayed in the error list box. The client GUI also notifies the repository about the dependent files of the new version and they update the checked-in file as a new version.
* **Check-out:**



Figure 5.3.2 GUI for client to Check-out

* The user must authenticate himself/herself before checking-out the file from the repository.
* The user can browse the file or enter the file name manually to check-out the file from the repository.
* Based on the file selected by the user, the check-out button is enabled or disabled. If the user is not the owner or not in a group which own the file then only the download button will be enabled.
* Once the file is successfully checked-out from the repository, a lock is placed on that file in the repository so that others can only download the file but cannot check-out. The lock means that the repository expects that file will be checked-in back later.
* **Client’s view of test Harness:**



Figure 5.3.3 GUI for client to access test harness tool

* The user can specify various DLLs to be used when testing the file by browsing the DLL location.
* Apart from the default test suites allocated for each code/file the user can also mention additional test suites from the available list of all test suites.
* Once the DLLs and test suites are mentioned the user can perform testing on his/her file by clicking the Run Test Harness button.
* The list of all successful and failed tests are listed in the successful test and failed tests list box.
* If there is any error or bugs during the test it listed in the error reports list box.
* The user can also log the test results for future reference.

## **Package diagram:**

Based on the tasks designed for the client GUI the following packages can be identified in the application.



Figure 5.4.1 package diagram for client

* **Sender module:**

The sender module is responsible for posting message received from the dispatcher to other servers. It usually sends messages to repository server for check-in and check-out of files and to test harness server to test developed code made by the user. It has its own blocking queue to manage multiple sends.

* **Dispatcher:**

The dispatcher module is responsible of conversion of messages received from the GUI into request format which the other servers can interpret. These request are sent to the sender module to be sent to servers. They also converts messages received from other severs for the GUI to interpret.

* **Receiver:**

The receiver module receives messages from server and sends them to the dispatcher to be interpreted.

* **Communication-service:**

The comm-service module handles communication between client and server using message passing. It is handles creating and terminating connection between servers and client.

* Graphical User Interface:

The GUI module is responsible for the design part of the client. The GUI is divided into several parts to handle check-in, check-out and view files in repository and test codes in the test harness server using WPF. The GUI gets the required input from the user and passes it to the dispatcher to be converted into messages and to be sent to the concerned servers. It displays the result from the servers in the GUI.

* **Local cache:**

The local cache module handles the frequently accessed files in the client side. Every time when the client checks-out or downloads a file from the repository it is stored in the local cache. Usually the checked-out file in the local cache has the same version number as the file in the repository server. The Least Recently Used (LRU) replacement algorithm is used when there is a cache overflow in client’s local cache.

* **File manager:**

The file manager module is used to read and display files received from the servers. This module is also used to create and manage files in the local cache. It uses the .Net framework for handling files.

## **Activity diagram:**



Figure 5.5.1 Activity Diagram for client GUI

As discussed in the package diagram the client GUI performs the following activity.

* The client uses the communication-service package to create connection between the client and the other servers. If the connection is not successful the client has to try again after certain duration. The duration is because the server communication-service is assumed to busy (queue is full) or service is down for some reason.
* After the connection is established the client should authenticate himself/herself by providing his/her user ID. The client is given three attempts to authenticate. Once the authentication is successful the client can access the repository server and test harness server.
* The client can check-in files into the repository. During check-in the client can browse for the file from the client’s local cache, provide description and keywords to better understand the file being checked-in. The client can also generate a metadata for that file which will get updated on the repository server once the check-in is successful. After check-in the lock on the previous version of that file is released and the version number is updated.
* The client can check-out/download the files from the repository server. The client can browse the repository for the required file. Once the required file is selected the repository checks if the client has ownership on that file, if so then the repository checks if there is a lock on that file and then I performs the following activity.
* If the file is not locked and client has ownership on that file then the check-out can happen and once the file is checked-out, a lock is placed on that file.
* If the client has no ownership on that file or if the file is locked then the client can only extract the file by downloading it.
* The client can perform tests on the files. Along with the default DLLs assigned for that file the client can also specify additional DLLs. The client should also specify files which are dependent on the file to be tested. Mostly the test happens on the file that was recently checked-in to the repository by the client. Once the test is completed the client can view the reports and log them.

## **Critical issues:**

1. **What happens if the client’s system has a different version of DLLs that is not present or supported by the CBIS?**

Description:

The clients develop their codes in their own system which has its own configurations. Sometimes these configurations are supported in the CBIS so when clients check-in their code back to repository and run them using test harness the code would fail or break down the system.

Solution:

The clients can avoid this by adding DDLs and other configurations into the repository. These DLLs and configuration are marked as untested (orphan packages) and they can be used when testing codes in test harness by the selecting these untested DLLs. However it is recommended that the clients use the same DLLs that are marked as tested and running the repository.

# **Prototype**

## **Metadata generator:**

A metadata file is an XML file format present for each and every files/ code that is present in the repository server. The metadata contains various information such as ownership of the file, creation date, last modified date, file size, version number, dependent files, supported DLLs and so on. The list of dependent files that are mentioned in the metadata are also part of the project’s baseline and are stored in the repository. The dependency is used to identify the child relationships which is necessary when the file is checked-in to the repository. When the file is checked-in the dependency analyser tool goes through the repository to find if the dependent files mentioned in the metadata are part of the baseline and these information is stored in the dictionary. The dictionary can be used for future reference until the file is modified or the dependency is changed.

## **Caching in build server:**

The build server could implement caching by storing the files received form the repository server earlier. When the test harness server request repository server to get the dependent files of the file to be tested the repository sends the list of those files based on dependency relationship. This list is received by the build server and it downloads all those files from repository into its local cache as well build images.

The repository gets these dependent files either by going through the dictionary or running the dependency analyser tool. The dependency analyzer goes through the metadata file of the file to be tested to get the dependent files.

This would reduce the load on the repository server since the cache in build server also these files it can send them to the test harness.

However the build server should manage the cache incoherence. It is not always possible for the client to have the same version that is present in the build server’s cache. So whenever the client requests for files the build server should make sure that the file’s version in build server’s cache is same as the file’s version in repository. If not then the build server should download all those files. Another issues is to manage the cache contents of build server. One possible way out is to use the Least Recently Used replacement algorithm to replace files if the cache get filled up.

# **References:**

<http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/HiResTimerInterop/>

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Project5HelpF14/>

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/TestHarnessPrototype/>