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

**-------------------------------------------------------------------------------**

**CSE 681-Software Modeling and Analysis**

**Project #5**

**OPERATIONAL CONCEPT DOCUMENT (OCD)**

**VERSION: 1.0**

**Venkata Karthikeya Jangal (567413382)**

**Date: 12-08-2014**

**Instructor-Dr. Jim Fawcett**

Contents

[1.1 Importance 5](#_Toc405843421)

[1.2 Primary objective 5](#_Toc405843422)

[1.3 CBIS model 5](#_Toc405843423)

[1.4 Data Flow of CBIS 6](#_Toc405843424)

[1.4.1 Check-In 6](#_Toc405843425)

[1.4.2 Check-Out 6](#_Toc405843426)

[1.4.3 Dependency queries 7](#_Toc405843427)

[1.4.4 Exclusive tests requests to test harness server 7](#_Toc405843428)

[1.5 Users and Uses 7](#_Toc405843429)

[1.6 Critical Issues 7](#_Toc405843430)

[2. Introduction 8](#_Toc405843431)

[2.1 Architecture Diagram: 8](#_Toc405843432)

[2.2 Organizing principles: 9](#_Toc405843433)

[3. Uses 10](#_Toc405843434)

[4. Users 11](#_Toc405843435)

[4.1 Software Professionals 11](#_Toc405843436)

[4.2 Software Managers 11](#_Toc405843437)

[4.3 Software Architects 11](#_Toc405843438)

[4.4 Quality Assurance Team 12](#_Toc405843439)

[5. Client 12](#_Toc405843440)

[5.1 Objective 12](#_Toc405843441)

[5.2 Package Diagram 12](#_Toc405843442)

[5.3 Views 17](#_Toc405843443)

[5.3.1 Login 17](#_Toc405843444)

[5.3.2 Select server to connect 17](#_Toc405843445)

[5.3.3 Test Harness server view in the client 18](#_Toc405843446)

[5.3.4 Repository server view in the client 19](#_Toc405843447)

[5.3.4 File Info 20](#_Toc405843448)

[5.4 Issues 22](#_Toc405843449)

[5.5 Uses 23](#_Toc405843450)

[6. Repository 23](#_Toc405843451)

[6.1 Introduction 23](#_Toc405843452)

[6.2 Repository Server Package Diagram 24](#_Toc405843453)

[6.3 Activity Diagram 29](#_Toc405843454)

[6.3.1 Activity Diagram for Check-in 29](#_Toc405843455)

[6.3.2 Activity Diagram for Check-out 35](#_Toc405843456)

[6.3.3 Activity Diagram for Dependency analysis 38](#_Toc405843457)

[6.4 Uses 40](#_Toc405843458)

[6.5 Issues 40](#_Toc405843459)

[7. Build Server 41](#_Toc405843460)

[7.1 Client on usage of Build server functionality. 41](#_Toc405843461)

[7.2 Package Diagram 42](#_Toc405843462)

[7.3 Activity Diagram for Build server 44](#_Toc405843463)

[7.3.1 Activity Diagram for Build server based on repository request 44](#_Toc405843464)

[7.3.2 Activity Diagram for Build server based on Test harness request 46](#_Toc405843465)

[7.4 Uses 48](#_Toc405843466)

[7.5 Issues 48](#_Toc405843467)

[8. Test Harness Server 49](#_Toc405843468)

[8.1 Importance 49](#_Toc405843469)

[8.2 Description 49](#_Toc405843470)

[8.3 Functionality 52](#_Toc405843471)

[8.4 Package diagram of test harness server 52](#_Toc405843472)

[8.5 Test Harness Activity diagram 55](#_Toc405843473)

[8.6 Uses 57](#_Toc405843474)

[8.7 Issues 58](#_Toc405843475)

[9. Communication and services in the entire system 58](#_Toc405843476)

[10. Policies 62](#_Toc405843477)

[10.2 Check-Out policy 63](#_Toc405843478)

[10.3 Check-In policy 63](#_Toc405843479)

[10.4 Notification 63](#_Toc405843480)

[10.5 Versioning 63](#_Toc405843481)

[10.6 Build policy based on dependencies of modules/ packages 65](#_Toc405843482)

[10.7 Storage hierarchy in the repository server 66](#_Toc405843483)

[11. Conclusion 66](#_Toc405843484)

[12. References 66](#_Toc405843485)

[13. Appendices: 67](#_Toc405843486)

[13.1 Class diagram of Test Harness server 67](#_Toc405843487)

[13.2 Prototype 68](#_Toc405843488)

**List of figures used in this OCD:**

[Figure 1 Architecture diagram of CBIS provided by Dr.Fawcett 9](#_Toc405799135)

[Figure 2 Package Diagram of client 16](#_Toc405799136)

[Figure 3.Login view of client 17](#_Toc405799137)

[Figure 4 Select server view 18](#_Toc405799138)

[Figure 5 Test Harness server view in the client 19](#_Toc405799139)

[Figure 6 Repository view in the client 19](#_Toc405799140)

[Figure 7File Info view in the client 21](#_Toc405799141)

[Figure 8 Add dependency view 21](#_Toc405799142)

[Figure 9 Package diagram of Repository server 24](#_Toc405799143)

[Figure 10 Activity Diagram for Check-in in repository server 32](#_Toc405799144)

[Figure 11 Activity diagram of check-in while storing in the repository 34](#_Toc405799145)

[Figure 12 Check-out diagram 1 35](#_Toc405799146)

[Figure 13 Activity diagram for Check-out 37](#_Toc405799147)

[Figure 14 Activity diagram for dependency analysis 39](#_Toc405799148)

[Figure 15 Package Diagram of Build Server 42](#_Toc405799149)

[Figure 16 Activity diagram of build server based on repository request 45](#_Toc405799150)

[Figure 17Activity Diagram for Build server based on Test harness request 47](#_Toc405799151)

[Figure 18 Test Harness structure provided by Dr. Fawcett 51](#_Toc405799152)

[Figure 19 Package diagram of Test harness server 53](#_Toc405799153)

[Figure 20 Activity diagram of Test harness server 56](#_Toc405799154)

[Figure 21WCF communication between Client App and Service Host 58](#_Toc405799155)

[Figure 22Versioning Concept Diagram given by Dr.Fawcett 64](#_Toc405799156)

[Figure 23 Dependencies of packages and storage structure 65](#_Toc405799157)

Executive Summary

## 1.1 Importance

For all the software projects to be delivered on time are normally built and tested rigorously. Developing the code and building it to test and integrate with other developers is challenging. There is no common platform for all developers to check each other progress and test same test suites. The complexity of building, testing and integrating gradually grows with the number of developers and size of the project. These days it is common to see a large project being developed by many number of developers with one or more file depending on each other. Managing the quality of software becomes tremendously difficult in this scenario. The main reason for this can be miscommunication between developers and human errors. Furthermore, a developer sometimes waits for another developer’s code in order to proceed on his work because of interdependency and the problems increase with the delay. All these issues, are the reason why there is a need for a “CONTINUOUS BUILD AND INTEGRATION SYSTEM”.

## 1.2 Primary objective

As the name itself signifies, this is more like an agile type of Software development. Continuous Build and Integration System’s (CBIS), primary objective is to provide a platform for the developers to continuously develop a code, build and test it, so that there is no time lapse in integrating the developers code with the developing software baseline. This system gives rapid feedback and hence error in the code is identified and corrected effectively. It makes the job of Software manager which is working towards the common goal of producing a quality software a bit easy. Let us discuss in detail about the Continuous Build and Integration System.

## 1.3 CBIS model

This OCD describes model of CBIS system, which consists of:

1>Arbitrary number of clients 2>Repository Server

3>Build Server 4>Test Harness Server

This OCD explains clearly how the client provides user interface to developers and help to interact with the other parts of CBIS system. Implementation of each unit of repository server is mentioned. Package diagram explains the functionality of each component. Flow of execution is depicted and explained using activity diagrams. Similarly package diagram for Build server and test harness server is provided to get a complete picture of how build server is responsible for building and sending the build images to Test harness server and how Test Harness server executes to give back the result for the client to display. Since this system has client-server architecture, there is common testing and analysis which avoids lot of problems. The communication occurs through WCF service and they exchange messages in the form of XML messages.

## 1.4 Data Flow of CBIS

Let us discuss the top level data flow of this CBIS as a whole briefly.

### 1.4.1 Check-In

All the clients and servers get started from command line inputs in the executive package. Client knows about server address by querying through the xml file, which is already stored in its location. Client selects the modules or files to be checked-in. So source code along with the test suite and file info goes to repository server. Repository server does dependency analysis and invokes metadata generator. All the metadata is added to the module and sent to build server. Here, all the message communication is through XML messages. The package from repository server to build server is sent through file transfer. Build server checks the dependencies using metadata and builds them first. If the files are already built, they are present in cache and it doesn’t have to build it again. So module gets built and build image is sent to test harness server for testing. Test Harness server tests using the test suites and on success returns success message to build server and cache the build in the test harness server. Build server sends the message to Repository server and the package or module gets checked-in after authentication. It gets stored at the specified sub-system, at the specified module using the file info present in the message or using metadata. This success message is sent to client using client source address in the message object. This flow is for check-in.

### 1.4.2 Check-Out

There is a difference between check-out and download.

Check-out: File or package can be checked-out, read, edited and can be checked-in. During check-out, a lock is imposed in the server side so that other users cannot check-out same file until the user who checked-out checks it back into the repository server. So a group of team can only check-out and work towards the end of the project. This type of accessing is supported using multiple owner policy and privileges should be given to check-in for the entire development team by the Software architect.

Download: Download is available for all the clients in the particular group even after check-out. It is read only and cannot be edited or checked-in. These files are downloaded to write or develop their own code by basing on the downloaded code i.e., the developing code will be dependent on the downloaded code.

For check-out, the client has a display of all the files which are available in the repository. On selection of file for check-out, user authentication is asked to maintain records that which user checked-out which files. On successful user login, the module or file is able to download into the client system and later can be edited, read and can be checked-in as this system operates through closed check-in policy with multiple owners and only the development team, which created the module is able to check-in the package. Although the package is checked-out, it is available for the other developers as it is a part of developing baseline and there is a copy maintained in it because of interdependencies. So multiple download can be done but check-out can be done by the user in the development team and that too one at a time. Further discussion on closed check-in policy with multiple owners will be discussed in the policy section of the OCD.

### 1.4.3 Dependency queries

Similarly dependency queries can be done as check-in and check-out to the repository server and get displayed on the client side. The only change is authentication is not needed.

### 1.4.4 Exclusive tests requests to test harness server

Exclusive tests can be done QA or any client assuming the files and few test suites are already present in the repository server. Tests are done by supplying test config files to test harness server. Test harness server gets the files or modules from the repository server or if the tested .dll files are present in the cache and if it is latest, it does testing on the information given in the test config, which is sent by the client. The results are logged and the report is sent back to the client, which can be used for the display.

Furthermore the architecture is explained briefly in the introduction section of the OCD.

## 1.5 Users and Uses

The principal users of the proposed CBIS are Software professionals, Managers, Software architects and academicians along with Quality Assurance Teams.

There are lot of uses for this system like it helps in achieving to maintain a quality code, helps in reduction of network traffic, helps to synchronize work done by Software developers by continuous check-in and check-out facility and helps Software managers to lead the team towards a common goal. It also provides storage and tracking facility (which developer is developing which code).

## 1.6 Critical Issues

Every system has one or more issues. After careful analysis and listing of crucial elements and its functionality, this system also has Issues. Few of them are:

1. Security
2. Performance
3. Connection failure
4. Checking-in same files with different metadata info
5. Look up or search

All these issues are clearly discussed under each of client, Repository and test harness servers.

# 2. Introduction

This section deals with the architecture and Organizing principles of Continuous Build and Integration System.

## 2.1 Architecture Diagram:

Architecture of CBIS essentially consists of 3 servers. They are: Repository server, Build Server and Test Harness server. Clients can be of any number and it makes use of WCF service to communicate with remote servers. Development of code happens in the client’s side and client supplies code along with test suites to the Repository server. It is the duty of repository server to send the code to build server. Later it performs top level regression tests in the Test Harness server. Test suites are linked with the module and hence used from the repository server. If all of these tasks are successfully performed without any errors, then only the newly sent code from the client is added to the developing baseline code in the repository server. Repository server also provides facility to store packages that are not currently a part of module but only properly built and tested code is checked in or becomes part of developing baseline. These are called orphan packages. Build Server cache the source code modules which is previously used for builds and on requests from the Test Harness Server it checks it with the repository for any changes. If there are any changes, build function is performed again or else the same built image is loaded into the test harness for testing i.e., Re build is done if package interdependent files inside module gets modified. This information whether file is modified or not, is checked through the metadata. Test Harness performs tests received from the client XML messages, which has test config. This server also maintain cache for current tested baseline but it requests modules from the build server if there are any changes in the previous build. After loading, it executes and compiles the current tests, logs and returns success or failure notifications to the client, which is displayed on the GUI. So repository supports check-in and check-out along with dependency queries on modules. A log is also maintained for the executed tasks and if testing fails, it sends notifications to the selected users/clients which are dependent on the current modules. Test Harness server supports multiple testing using app domains.

The below diagram shows the architecture of CBIS developed by Dr.Fawcett. Based on the below diagram, a complete package diagram of individual servers and client is developed and explained clearly in the package diagram and activity diagram section.

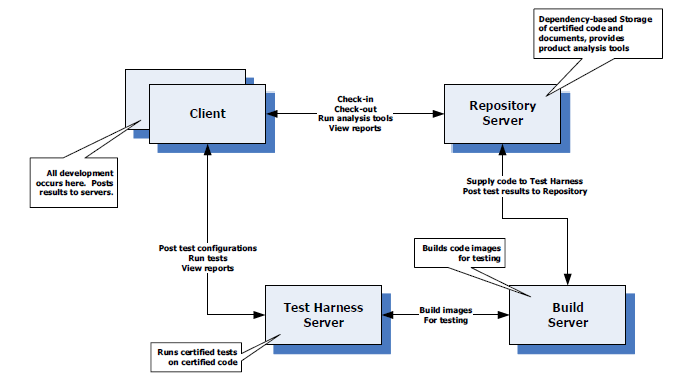


Figure 1 Architecture diagram of CBIS provided by Dr.Fawcett

## 2.2 Organizing principles:

1) Modules supports unit tests and regression tests.

2) Test Harness server supports multiple testing at a time.

3) Repository is immutable.

4) Each package provides a function that implements ITest, will normally be built as DLL.

5) Documents and builds are encapsulated.

5) Any of the server fails, then error is returned back to the client.

6) Repository server stores all the code, even if the code dependent on it is not checked in. Testing and build is usually done once all interdependent files are available. Test suites are available along with the code supplied.

7) Check in should be failed if the test drivers are failed to load.

8) Check in is initially done on the approval of the Manager only.

9) If a new package has few changes compared to old one, all the packages which are dependent on the changed package are built and tested.

9) Every time a build or test is performed, it is stored in cache of Build server and Test harness server respectively.

8) Repository server supports execution of queries for the dependency analysis.

10) Repository also maintains integrity of modules.

11) This system is closed check-in policy with multiple owners i.e., the client who is part of development team of a module will be able to check in and check-out.   
12) Test harness server has several app domains running on threads.

13) Repository has a baseline which always runs and never stops.

14) Continuous build and test via built in test functionality.

\*\*Some of the organizing principles are taken from Enterprise lecture present in the project5 folder help given by Dr.Fawcett, which are almost similar to this OCD.

# 3. Uses

3.1 Everyone can submit the code to the baseline daily

With the continuous integration baseline is updated frequently by many users and this results in transparency of what is happening. The number of changes made are few when compared to all code developed and submitted. This avoids lot of errors. Submitting the code every day is a part of continuous integration.

3.2 Conflicts are small and easy to solve

CBIS helps the developers to identify the error present in the submitted code and who has submitted that code can make few changes and build it successfully as the changes done are very few.

* 1. Availability of Repository server

CBIS has a repository server which stores all the baseline code and also supports querying of dependency information of the submitted packages.

* 1. This is a kind of agile development and hence avoids last minute chaos

CBIS has continuous build and testing which avoids last minute chaos at release dates. Even if the problems occurs, as said earlier it is easy to fix

* 1. Latest Builds

CBIS provides facility to get the latest build to work on. Latest builds are devoid of errors.

* 1. Cache

Continuous Build and Integration System has a facility to maintain cache and this helps to avoid multiple builds and thereby makes the testing and integration faster.

# 4. Users

Software professionals, Managers, Software architects and academicians along with Quality Assurance Teams are the users of CBIS. Let us discuss how they can use CBIS.

## 4.1 Software Professionals

Nowadays a large project involves lot of software developers to work as a team. Each of them is assigned with a task and there is a lot of chance that the code is dependent on one another. In this situation, if everyone works independently and try to integrate it, which results in lot of errors. Each of the developer needs to sync each other code at least once in a day and transparency of who is working on what is needed. Furthermore, if there is an error, developers needs to know whose code has error and update the latest build. For all these issues, our CBIS provides the right platform to develop the code build, test and integrate it continuously. It provides the platform for good communication and helps to achieve the common goal of building a quality software.

## 4.2 Software Managers

Managers duty is see that work is progressing in the right path and keep track of individual developers job, so that final outcome is fruitful. Managers can schedule the work of the developers using CBIS, based on the work pending. Continuous Build and Integration System makes the job of Manager easy as it takes care of major job of communication between developers and giving clarity on what needs to be done.

## 4.3 Software Architects

Software architects get a clear picture of the complexity involved while developing code and integrating with different codes. So it helps architect to analyze complexity and issues, so that a better architecture is developed the next time.

## 4.4 Quality Assurance Team

Almost all the code are free of errors while integration is done. QA can use Test harness server to do unit tests and get the errors. They can get the information of the developer, whose code has error and send notifications to developer whose code has error and registered developers for all modules that depend on the tested modules.

# 5. Client

Clients: These are the developers and managers, who are making use of CBIS. This has a user interface for communicating with Repository server and Test Harness Server. It is also responsible for displaying the received messages.

## 5.1 Objective

The primary objective of the client is to provide the users an interface, so that they can successfully communicate with the remote Repository, Build and Test Harness Server for performing continuous build and integration. Client makes use of Windows presentation Foundation for display. It provides an interface to check in, check out, make dependency query into the repository server and also to request direct testing using config files to the Test Harness Server.

## 5.2 Package Diagram

The package diagram of client mainly has following modules:

1. Client Executive

2. WPF Client GUI

3. File Handler module

4. Error Handler module

5. Process reply module

6. Communication module

7. Sender

8. Receiver

9. Local File set storage module

10. View handler

* + 1. Client Executive

This package is responsible to start the Graphical User Interface for the client. Client executive reads the port number from the command Line and initiates the client. If there is a client existing with the same port, error is handled and displayed using console to use a different port number.

* + 1. WPF Client GUI

Here graphical user interface of client is displayed to the user using Windows Presentation Foundation. ServersAndPorts.xml file supplied to the project has list of all the servers ports and whenever user logins successfully, he/she can select Repository server or Test Harness Server and get connected to the selected server by querying the selected server name and getting port number from the xml file to establish a channel for communication. Detailed analysis of how communication is established and what are the message types is explained in the communication section of OCD.

Depending on the server selection, the requests sent are changed. For instance, dependency query, check in and check out requests are sent to Repository server and Testing directly using test config files can be done by sending request to the Test Harness Server. The received messages has event handlers and on new message received, they are displayed on the results section of WPF. Result section just shows whether the request sent is successfully processed and received. Furthermore, the entire result like dependency analysis or result from test harness server is displayed with the help of new window which get popped out from the client WPF. Messages are sent from the client to the remote server as XML. Data Contract has Data members. Object for the class is created and data members are declared and this object is sent. Data members of the message can be

* Message source
* Message destination
* Message body
* Message Client source
* Message command
* Message Test suite
* Message Test config
* Message result
* Message List<mod\_sub>
* Message metadata

These are some of the messages that can be sent by creating object of Class which has all these data members.

Message source: It explains about the source address from where the message is generated

Message destination: Destination to which message is to be sent is contained in this.

Message body: All the file contents can be sent through the message body by using xml as string.

Message client source: When servers communicate with each other source and destination changes and ultimately the received message has to be sent to the client and so its address is preserved in this data member

Message command: Whenever a message is received, its command is checked and it contains what to do with message. If the message sent is for check in, command has string “check-in” and by comparing this string on reception of the message, the servers know what the client is asking to do with the message.

Message Test suite: Consist of all the test suites for testing. Test suites are collection of test cases that are intended to be used to test a code in test harness server.

Message Test Config: Sometimes the client requests the Test harness server to do unit tests on few codes directly, at that time client has to supply test config file to the Test Harness server and this data member holds the information as xml.

Message result: The result whether Check in failed or got success or whether check out is successful or whether test harness executed testing successfully is known by the client using this data member.

Message (List<mod\_sub>): As the repository server supports querying the dependency analysis, list of modules or sub systems are selected for query and are passed to repository through this message list.

These are few sample messages that can be sent between client and server as well as server and server using single object of the class consisting these data members.

A shown in the package diagram given below, Wpf Client GUI package sends requests to the communicator package of client which creates channel to send the message.

* + 1. File Handler

A files handler package helps to access the file to read the contents for display or to send the contents as string to the remote server.

* + 1. Error Handler

All the errors occurred are carefully handled by the error handler package. Wpf client GUI package sends the exceptions occurred and it is the duty of error handler package to handle the exceptions and display the error in a new window, so that the user can rectify his mistake.

* + 1. Process Reply

This package receives all the messages coming from the server and processes what to do. If for instance received message command is “checkout received”, then it stores all the files in the local storage available and displays success message in the results window of the client. The client GUI provides a refresh button to list all the checked out files and file handler module help to access the file and client can open the files present in the storage and modify the files to avoid errors in build and test.

* + 1. Communication module

Communication module is responsible for communication with other servers. This module receives from receiver and sends to appropriate address by creating a channel or receives from other remote servers and sends it to receiver package. Detailed version of communication is explained in the communication section of the OCD.

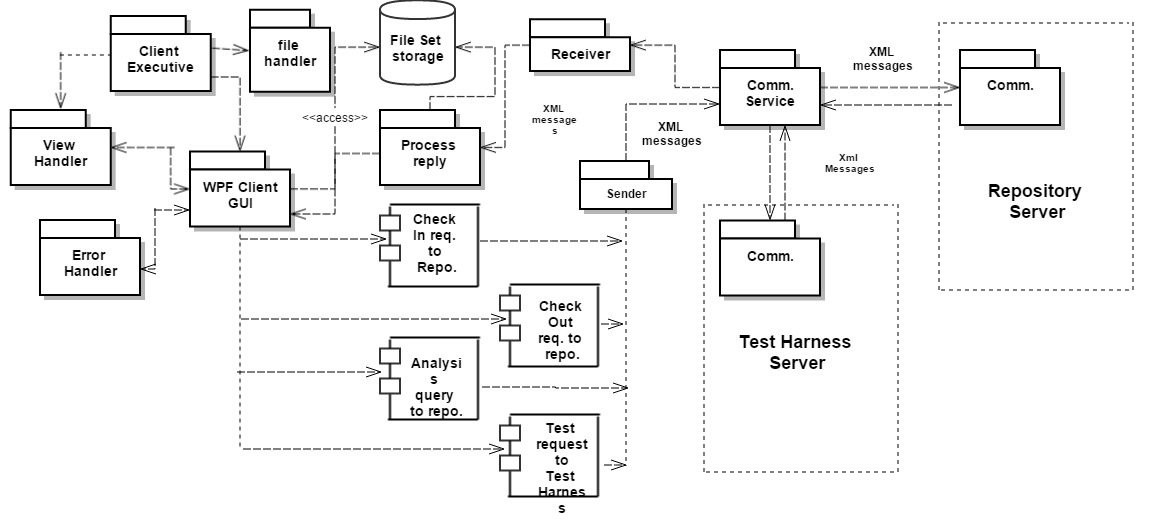


Figure 2 Package Diagram of client

Given above is the package diagram of client. This package diagram is re drawn in Repository server package diagram and Test Harness package diagram. But those diagrams of client focus on the functionality of respective servers.

5.2.7 Sender

This package encapsulates the message and sends it to the communication module.

5.2.8 Receiver

It is used to give the received reply to “process reply” package on reception of new message, to get the message processed and displayed on the client GUI.

5.2.9 View Handler

View handler takes care about the display of the WPF GUI. It updates the view on clicking update tab in the GUI.

## 5.3 Views

Client is almost about the views. Given below is the view of the WPF application which will be developed for the client to interact with the remote servers.

### 5.3.1 Login

The only way to implement security is to provide login facility. Here, clients are authorized by the managers using their ID and password. For every file checked in by the client, approval of Software managers is needed. The file metadata stores the employee id and thereby helps software managers to identify the right owners of the file and give rights to the client to perform check in of the files. If the client is not the right user of the file, client shows invalid user to check-in the file in the window. Check-out can be done by all the clients. If the client is not at all an employee, then it shows invalid user id and password.

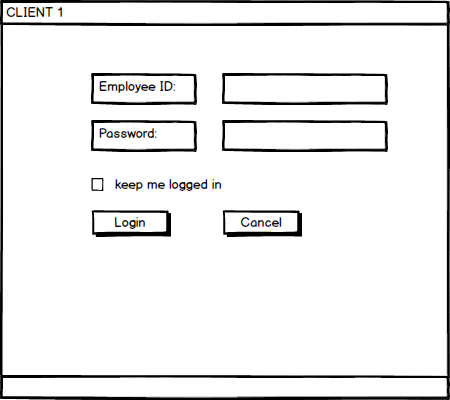


Figure 3.Login view of client

### 5.3.2 Select server to connect

Client has the option to connect to repository server and perform query on dependency relationship or do check in/ check out. It also has the option to connect to Test Harness server and supply Test config file to do testing on the Test harness server and get back the results. So after logging in the client, select the server to communicate.

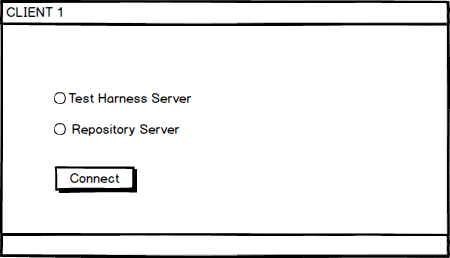


Figure 4 Select server view

### 5.3.3 Test Harness server view in the client

Test Harness Server view needs a config file to upload and a results window to display the tests. The received message are stored as xml file, however LINQ to SQl query is performed and results are displayed in the list box provided.

Search Test Config Files button helps to display all the files in the list box, which are present in the client. A file has to be selected and upload button helps to transfer that Test Config file to the test harness server. Client views reports after running certified tests in the test harness server from the reply messages received from the test harness server.

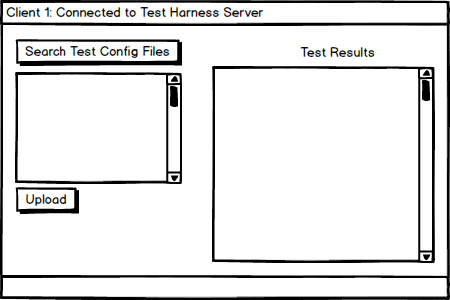


Figure 5 Test Harness server view in the client

### 5.3.4 Repository server view in the client

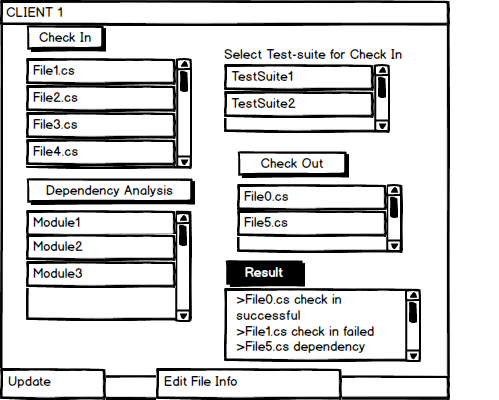


Figure 6 Repository view in the client

As discussed, repository server has list of files to do check in. The view shows the check-in,

Check-out, dependency analysis, update and file info buttons. Initially check-in button is not enabled. After selecting the files it needs to be checked-in, test suites and the entry of file info it enables the check-in button. Update button always updates the view of the GUI.

It also displays all the files available in the repository storage under check-out button. For check-out, Client selects package or file in the list box present under check-out button and clicks the button. The request goes to repository server and there is an authentication package in the repository server. This package is responsible to check whether this client is a part of development team of that module and gives access rights. So only those files which can be checked-out. All these files are displayed in the list box by sending request to database table which consists of Employee ID and file names. Update button helps to refresh the window and list the latest files available for check-in or the files available in the repository server for check-out. File checked-in needs approval from the manager. Information in File info is used to tell repository, under which sub system and which module does this file should be stored or checked in.

Messages are sent to repository server as XML messages.

When files under check-out button are selected and button is selected, all the files are downloaded into the client side and modifications can be done and re checked-in. Dependency analysis can be done by selecting modules and clicking on dependency analysis button. These exclusive results of dependencies are shown in a new popup window and the success or failure of the dependency analysis is shown in the results view.

### 5.3.4 File Info

File info is used to supply the following information:

* Employee ID
* File Name
* Package name
* Sub system Name
* Module Name
* Team ID

Client can enter filename, employee id and metadata information of the file while development of the package for check-in. Team ID can be used to help all the team members to access check-in or check-out their team’s packages.

If multiple modules gets selected for analysis, each of them is sent by Xml messages using threads. The file info is used by the repository to store in the exact sub system and module with the exact name if it gets checked in.

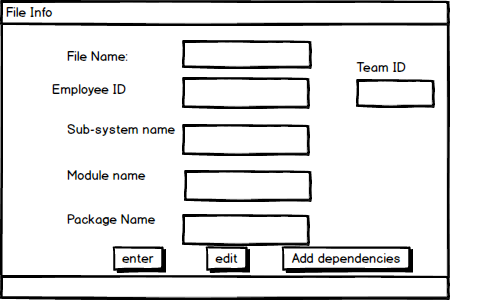


Figure 7File Info view in the client

When add dependencies button is clicked, the view is appeared as shown below:

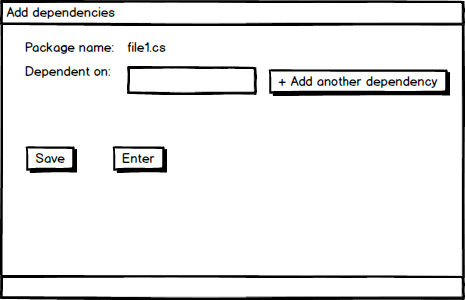


Figure 8 Add dependency view

Normally, dependencies are obtained by dependency analysis package in the repository server. But it is always better to add dependency file info, so that the repository need not do analysis every time it receives the message. While developing, the client knows that its package is dependent on so and so packages, so adding that info is not a problem.

## Issues

5.4.1 Many processes accessing same files

Problem: As we have check-in, check-out, dependency analysis features available on repository server but accessing or using same file while it is in process creates problems.

Solution: Threads should be used and locking is done on the file. Once the file is used and released then only it can be accessible to be used for other process. For example, Mutex can be used to avoid this problem.

* + 1. Performance

Problem: Large amount of data being sent and multiple clients sends multiple files to multiple server, results in more traffic

Solution: If the server functionality is same, instead of using and sending to different server, using a high configuration server which does the work of many servers and has the capability to handle many requests must be used.

* + 1. Security

Problem: Chance of attacks on the data sent across the networks can happen. Messages can be altered or deleted or diverted.

Solution: Proper network encryption is to be done before sending the data to the remote systems

* + 1. Same files with multiple names

Problem: There is a good chance that a developer after getting the check-in failure message, updates the file and change file info and submits back to the repository server to do check-in. The repository server considers it as a complete new file and treats it accordingly leaving behind the saved file which is not checked-in but stored in the repository.

Solution: Whenever client is trying to do change in file info, it should be informed to the repository server to make the similar changes and hence versioning becomes easy and it will avoid storing multiple files with same information or else a mechanism should be adopted to avoid the usage of multiple names. If a name is already used, it shouldn’t let it use again.

## Uses

* Provides GUI to make use of Continuous Build and Integration System.
* Client can remotely get dependencies of the developing baseline code.
* Provides developers to locate errors, fix and re check-in.
* Provides GUI to display error reports coming from Test Harness server.

# 6. Repository

## 6.1 Introduction

Repository server is related to storage. It supports:

* Check-In
* Check-out
* Dependency query

Repository server stores all the baseline code which gets checked-in. It maintains hierarchy of files system as:

System--->Sub-system---->Module---->Package

Group of packages from a module and group of modules form sub-system. System has several sub-system. So this hierarchy provides a team to work in a group and share the software baseline code to develop a quality software using a team effort. A team works on the development of module. So basically repository defines modules as a list of packages. Packages are accessed through interface and object factory. Repository also stores test suites and link test suites with particular modules. All the modules and test suites are supplied to repository from the client using XML messages. The hierarchy of files are defined in the file info part of the client, while supplying to the repository.

Repository maintains centralized storage which stores baseline development code and also failed check-in code. Code doesn’t become part of baseline if failure occurs in build server and test harness server. Failure occurs because the interdependent files are not available, which results in the build failure or the source code has some errors.

## 6.2 Repository Server Package Diagram

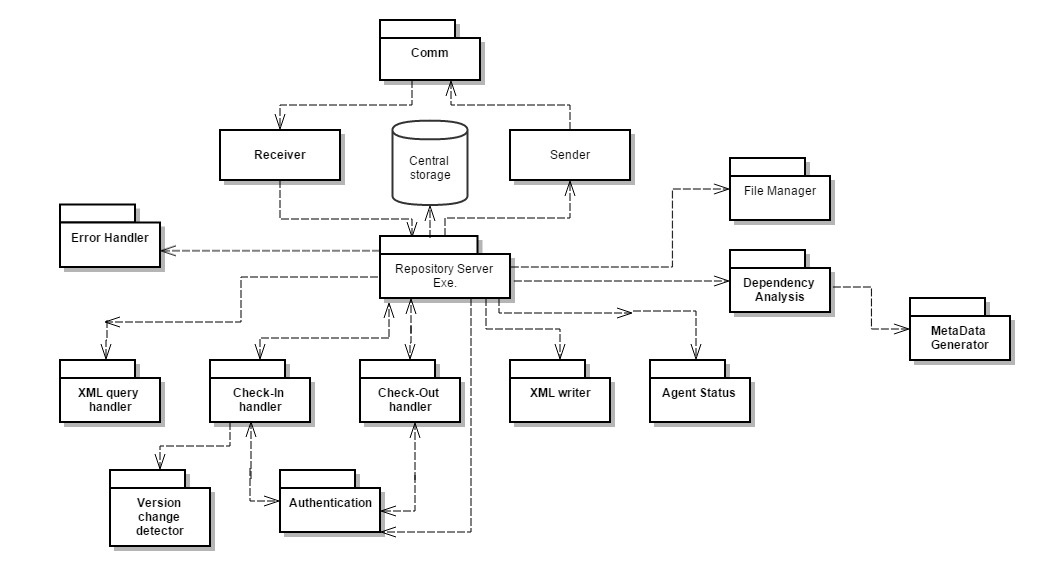


Figure 9 Package diagram of Repository server

Major parts of package diagram related to repository server are:

1. Repository server Executive
2. Communicator
3. File Manager
4. Dependency analyzer
5. Meta Data generator
6. Central storage
7. Check-In handler
8. Check-out handler
9. Authenticator
10. Version change detector
11. XML writer
12. Agent status
13. XML query handler
14. Error Handler

Let us discuss about each partition in detail:

* + 1. Repository Server Executive

Repository server takes port number from the command line input and starts the server by listening on that port. Whatever port number entered in the command line has to be updated in the ServersAndPorts.xml file in the client side, so that client will be able to connect to the server port on which it is listening. This package in the repository server has processing capability of the messages and dispatches it to appropriate package. Basically it knows what every package can do and uses object to access the package classes. In simple terms, it achieves internal communication within the server. Finally it gives the reply message to the sender present in the communicator package.

* + 1. Communicator

Communicator package is responsible for contacting the client as well as Build server. All the files for which check-in requests coming from the client are to be sent to build server through the communicator. On the event of new message reception through receiver, it is sent to Repository server executive. So it performs communication using WCF. All the messages received and sent are in the form of XML messages.

* + 1. File Manager

File manager module retrieves all the file names from the selected package or module from the given path that needs to be analyzed. It provides support for retrieving files with a pattern along with directories and sub directories. It gets them and stores them in a list<> data structure. Basically all these files are used for giving it to Dependency analyzer to do analysis.

* + 1. Dependency Analyzer

This package internally has many packages to find dependency between modules. Some of the packages that can be present in the dependency Analyzer are:

* Executive
* Analysis
* Type analysis
* Relationship analysis
* Dependency analysis

Along with these top level packages, there are lower level packages like parser, semi, toker etc. which forms the foundation for the dependency analyzer.

In order to get dependency analysis, we have to parse twice. First for building type table and second time for dependency relationships. Dependency analysis between modules or packages are essential to find the file dependency and build it if the dependent files are changed. So how do we know that the packages are changed or not? This is find out by checking metadata info for the file. So whenever dependency analysis is done, metadata is created and sent to build server, so that it checks the changes in the latest built of the source code module and decide on whether to build or not.

* + 1. Meta Data Generator

For every package or module sent from the client to repository server, the sent message has the information about file name, owner, sub system and module under which it has to be stored. It doesn’t have file last accessed or modified time and dependency information which is essential while building in the build server to avoid unnecessary built. So after dependency analysis, the metadata gets generated and gets added to the current module. After the end of testing, the metadata file which is usually xml file is linked to the current module or current package by storing in the respective folder either in check-in failed folder or in the baseline depending on the success or failure of the test.

* + 1. Central storage

Central storage has several storage places to store the baseline code and orphan packages along with check-in failed code. It sometimes have to check-out after authentication and therefore has to give the file contents to the client. It also gives the module to the test harness server when direct tests are conducted on the Test harness server by the client. It maintains a hierarchy levels in the form of sub-system, module package and thereby helps searching easy incase the client has to check-out the file. Metadata present in the module also helps for this reason.

* + 1. Check-In handler

This package is the core and crux of the check-in functionality. It handles all the modules or packages after successful testing and authenticates, if the user is right person to do check-In. If the authentication is successful, it sends the message to dispatch, which in turn stores in the central repository under check-in or baseline code with specified module name and package name in a specified sub system gave by the user. Check-In handler also informs version change detector that check-in is successful. Its functionality is explained in version change detector package. Another important job of check-in handler is integrity maintenance. All the interfaces and object factories are defined by the architect. These interfaces are to be implemented by the package which comes for check-in. Instead of implementing interfaces, if the package try to change the interfaces then check-in fails. Thus integrity is maintained while check-in

* + 1. Check-Out handler

Check-Out handler is very simple when compared to the check-in handler. The user check-ins few modules which has packages. For suppose, client wants to improve the functionality and add extra features to the developed code. So basically he wants to edit the baseline code but rule of thumb is that baseline code should be available for all the other developers. The reason it should be available for other developers, while client is making modifications in the client side is that many packages has dependencies on the file which the client is about the make changes. All the developers in a team cannot wait for the client to make changes and proceed. This results in delay and therefore we implemented CBIS, which is already discussed in the executive summary section. While the all the developers are working on a module, the client can check-out the package present in that particular module and make changes. This modified package needs to be checked-in again to become a part of baseline code. In this way, we get many versions of similar code as it is updated frequently. Check-out needs Software architect authentication. Here the client requests check-out and the repository server has Server executive which sends to Check-Out handler which handles these requests. It gives to authentication package where the authentication is done and the reply goes to Executive again. Executive process the reply and on success takes the module or package from the central storage and sends the file or module to the client. File transfer is done through xml message. On the client side, if the module is large xml, it is read as chunks and written to a file.

* + 1. Authenticator

This package basically authenticates the user/ client who is asking for check-in or check-out of a package. The authenticator package connects to Software architecture using communicator package in the repository server and sends the employee id and password. If the architect approves the request then authenticator package sends success message and executive checks in or checks out from the central storage.

* + 1. Version change detector

There is a package attached to the check-in handler package called Version change detector package. This usually checks whether the package is a new or updated one and maintains all the packages in the forms of versions. It also updates the version tag to the metadata of the package. This package is really helpful because if there is a problem with the new version of package, we need older version to get back on track and build it successfully. Later changes can be made on this older version by comparing new version and can be re-checked in. The problem is because of the version of the source code and one should be able to fetch the old version of the package and this maintains all the versions of the packages in the repository.

* + 1. XML writer

All the messages are in the form of xml and so all the files are converted into xml format and saved. Later StreamReader is used to read xml and sent them across the network. This package creates xml file that can be used for messages. It is also used for creating metadata.xml file which is important in the repository server.

* + 1. Agent status

It has two functionalities.

1. There is a lot of communication going from client to repository server and from repository server to build server and back from build to repository server etc. To understand the flow, the agent status package tells what processing is currently happening on the repository server. Is it sending the file for check-in to the build server or is it just received success message from the build server or is it received dependency query from the client? So whenever a message is received or sent, agent status package updates the status. This status updates are just to help repository and not used frequently.
2. This package is also responsible for starting the timer and defining time limit for the employee to check-in the checked-out package. If the time limit expires, a mail is sent to Software manager or architect who has complete privileges. Architect removes the lock, and informs developer about this.
   * 1. XML query handler

Basically this is a package that decodes xml file or it consists of code that performs LINQ query on the xml file and holds selected node contents for further use.

* + 1. Error handler

Error handling is very much important for any application. It is always better to show error message than to crash the application. All the exceptions in the repository server is handled by the Error handler package and suitable message is shown in the console.

This is the end of package diagram discussion.

## 6.3 Activity Diagram

Repository server activities are many. So let us discuss each of the activities separately using multiple activity diagrams.

The main activity diagrams which can be included are:

1. Activity for Check-In
2. Activity for Check-out
3. Activity for Dependency analysis

### 6.3.1 Activity Diagram for Check-in

The main activities for check-in can be populated as:

* Login
* Selecting server
* Select packages, test suites and file info
* Check in request
* Entering port number
* Get files
* Dependency analysis
* Meta data generation
* Process request
* Check-in
* Send notification
  + - 1. Login

Client enters the port number and gets started. After the client starts, it asks for login. The client needs to enter the employee ID and password and the Software architect authenticates. On failure, the GUI asks the client to re-enter correct employee ID and password.

6.3.1.2 Enter port number to start server

Server takes command line input for the port number and gets started. If the port number is already in use, error is displayed in the console to enter a new port number. Server gets started and will be waiting for the client request.

6.3.1.3 Select Server

After the client logs in, client can see repository server and test harness server to connect to. In this activity, client selects repository server. If the server is down or not started, connection fails. The view again asks to check the server.

6.3.1.4 Select packages, test suites and file info

As the check-in button gets enabled only after selecting

> Package which needs to be checked-in

> Test suites present in the client side

> And file info

All these are to be selected and entered.

6.3.1.6 Check-in request

Check-in request goes to repository server on clicking check-in button.

All these are activities on the client side for check-in in the repository server.

The request comes to repository server and the actual check-in starts now.

Basically the received messages are objects for a class which has all the data member contracts. Upon receiving message, the repository server checks msg.cmd { cmd is data contract of the communication.svc, which is accessed through the object msg of that class} and there can be four commands for the incoming message and based on the command the repository server understands what is requesting and what to do.

Request 1 > If the command of the message is “check-out”, check-out is done. Its activity is discussed in the next section and therefor is not shown in the activity diagram given above.

Request 2> If the command or msg.cmd is “dependency\_analysis”, its activity is discussed after check-out activity.

Request 3> If the command is “check-in success” or “check-in failed”, that means the message came from build server says that build and testing is done and this is the status of the message sent. If it is check-in failed, Check-in failed is sent to client and client understands that the package is failed to check-in. If it is check-in success, it gets merged on to the waiting repository activity.

* 6.3.1.7 Get files, Dependency analysis, Meta data generation, Process request Check-in, Send notification

Request 4> If the command is “check-in” it gets all the files from file manager and does dependency analysis, which in turn gives it to Metadata generator. It creates metadata.xml and adds the dependency info along with file info into the metadata.xml. As the request is check-in and for the repository to do check-in, it has to send to build.

[In Build server, the source code module gets build and given to test harness. Test harness tests and send result message to build which again sends to repository.]

So the message is sent to build and keeps waiting for the reply message. The reply message is request 3. On receiving request 3, it is sent to this waiting repository server. So if the message is success, the file is checked-in and notification is sent to client.

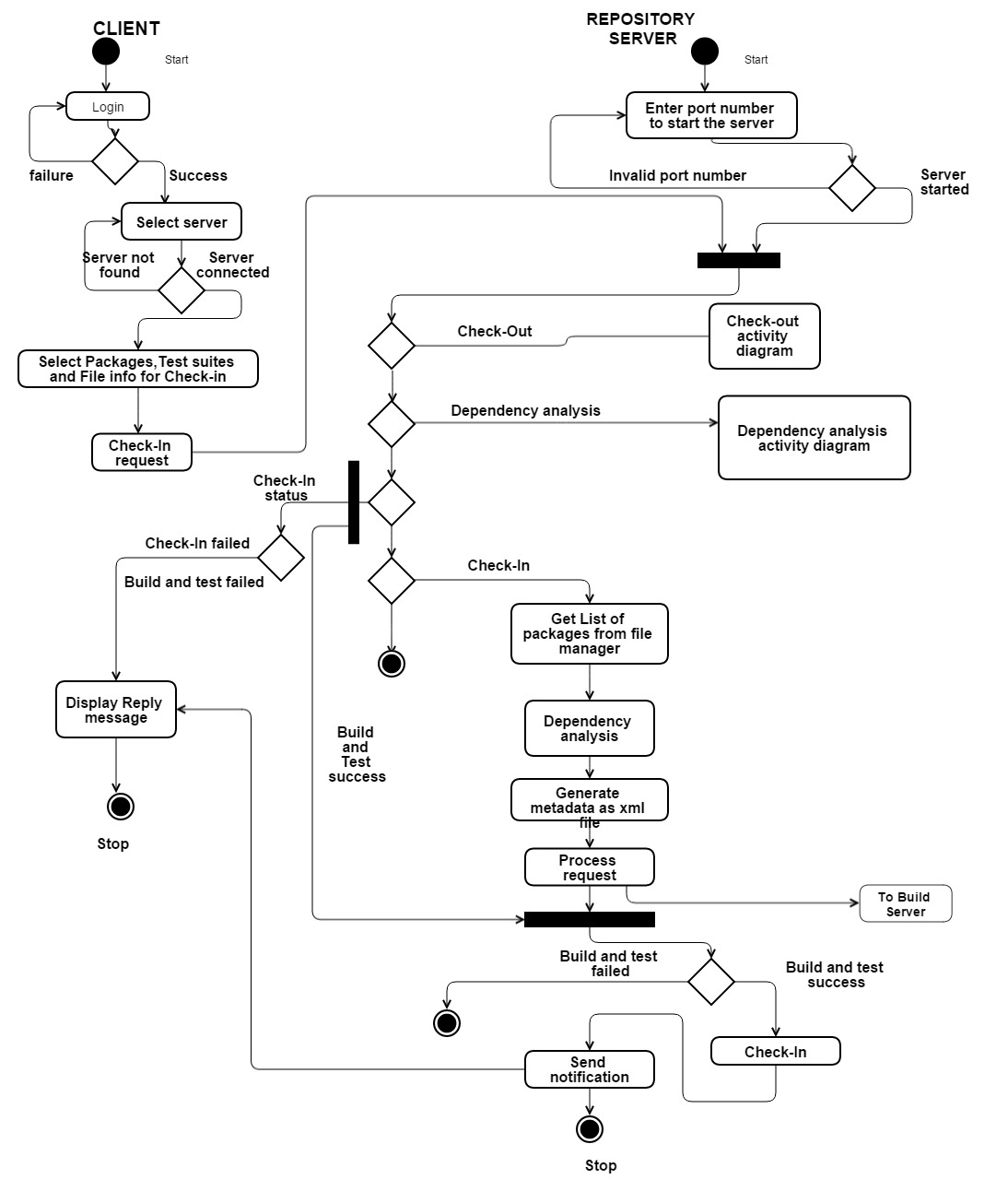


Figure 10 Activity Diagram for Check-in in repository server

Individual check-in activity has some more sub-activities to store in the repository. They are:

* Open module and get metadata
* Search for package name
* Create a new xml metadata for a new version of package.
* Versioning
* Save

Open Module and get metadata: The central storage has to store the module/package in the hierarchy level consists of sub-system, module name. So metadata is examined and first package name is checked

Search for package name: Package name is searched to find if it is a new version of existing package or completely new package.

If it is new package, a new sub system name is created with the new module name and package is stored in it. All the names and info is available in metadata.xml

Create a new xml metadata for a new version of package: If the package is already existing, a new metadata file with different file name, package name and module name is created for the versioning to happen.

Versioning: If the file is edited and checked-in, it will have same metadata other than last access and modified time. Package with same name is already existing in the repository and it cannot be deleted as we have discussed that repository server is immutable in the organizing principles. So there is a possibility that files gets checked-out and again gets checked-in, it is better to store the packages as versions. The module name is created with different version name but they are stored in the same sub-system. This kind of storing same edited packages in the forms of versions is called versioning. So every package has version 1, version2 etc. depending number of times the same package is checked-in.

Save: It is maintaining a database for all the packages in the repository. It basically stores in the correct path of repository.

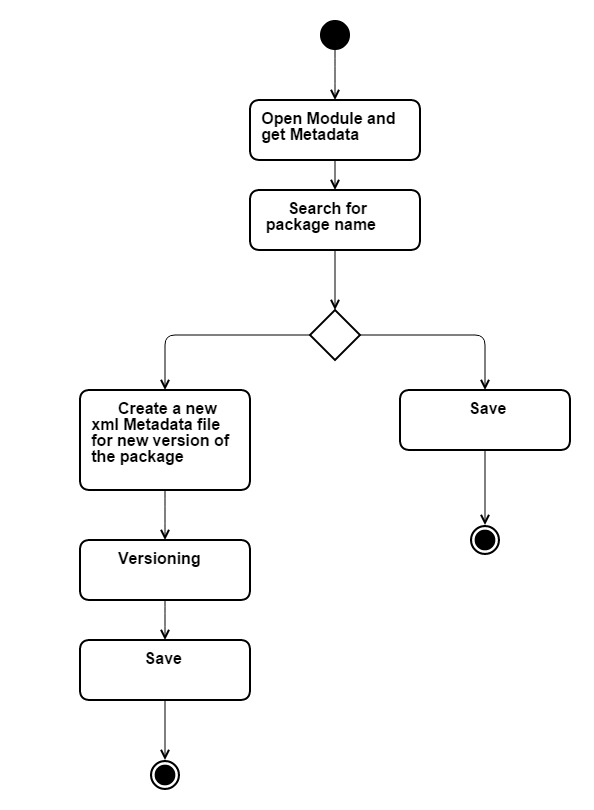


Figure 11 Activity diagram of check-in while storing in the repository

### 6.3.2 Activity Diagram for Check-out

We have already seen where Check-out activity diagram occurs in the Check-in Activity. For reference, this is the portion of diagram which the above line is talking about:

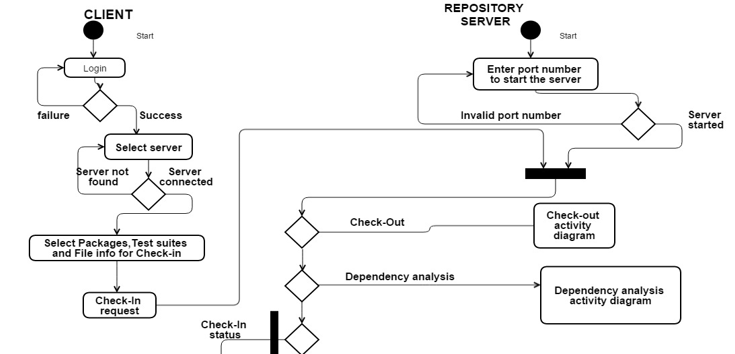


Figure 12 Check-out diagram 1

Because the request sent from the client in the above diagram is check-in, we avoided check-out diagram. Consider same diagram but the request coming is check-out, let us see what all activities check-out is having.

Some of the important activities of check-out once repository receives message are:

* Processing
* Authenticating
* Getting metadata and searching for file
* Check for lock
* Copy contents of file
* Send message back to client
* Start timer
* Send notification to manager based on timer

Description:

Initially the request is processed and sent to authenticator. This activity is done to check whether the client is the team member of the group which created the package which is asking for check-out. If authentication fails, failure notification is sent. Else it is sent to central storage, where metadata.xml is opened and package name is checked. If it is already locked, then it means someone already checked-out the code and same message is sent to client. If it is unlocked, contents of file are read and copied as xml messages and sent to the client.

Later locking is done and timer is started. If the time limit is exceeded, a notification is sent to manager and timer gets started again.

This loop repeats until manager unlocks or client who checked-out the code, checks in the code and removes the lock.

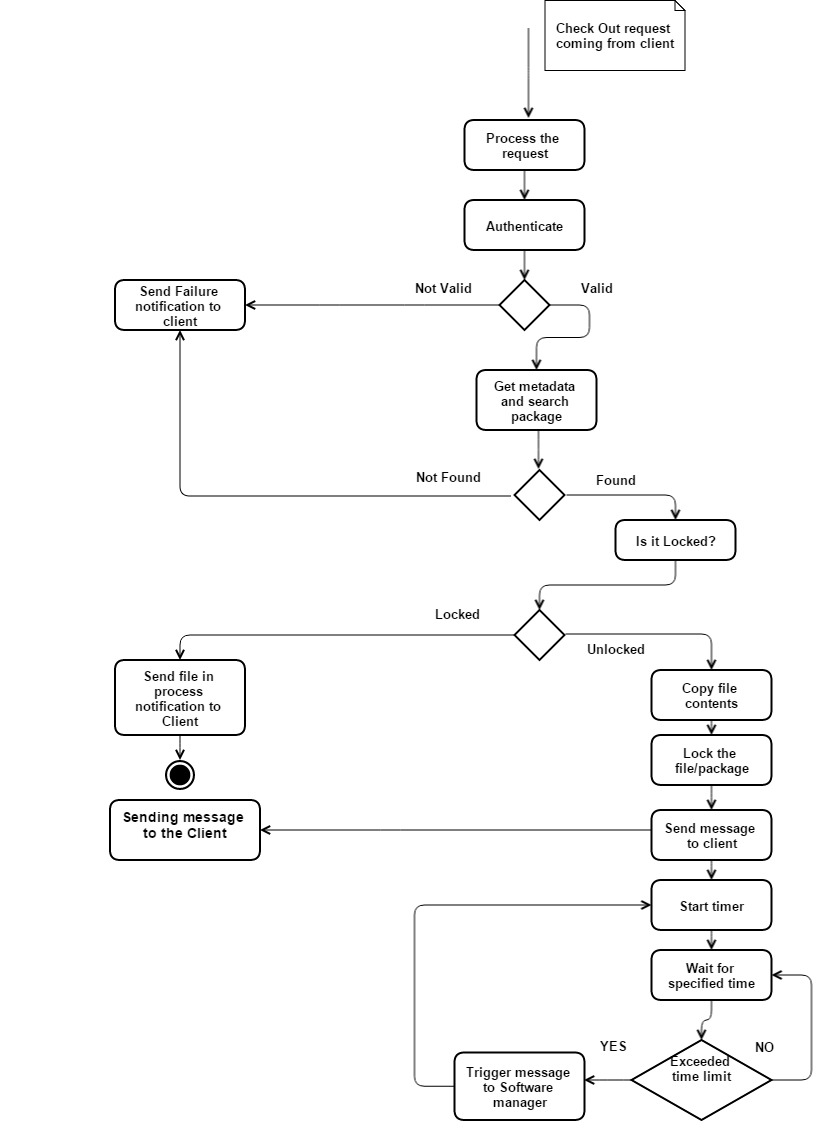


Figure 13 Activity diagram for Check-out

### 6.3.3 Activity Diagram for Dependency analysis

The main activities of dependency analysis includes:

* Start client
* Start server
* Send dependency analysis message from client to repository server
* Get files using file manager
* Parse it to parser present in dependency analyzer to build type table
* Parse again to find dependencies of packages and thereby find dependencies of modules
* Create xml file of the result
* Read the xml and send the file contents
* Receive message from the server and display it in the list box.
* Stop

Explanation on important activities:

After the request of dependency analysis is passed from the client to the server, repository server receives the message and process the request.

It understands that client is asking for dependency analysis. Authentication is not needed for doing dependency analysis. The Server get the list of files in the module and parse to get type table. It also checks for dependency files from given file info and builds complete type table. Later this does second parse to get dependency analysis and stored in xml. Stream Reader is used to read contents of xml and contents are copied to messages as xml and sent to the client.

Client receives the reply process it and displays dependency analysis by querying the xml file.

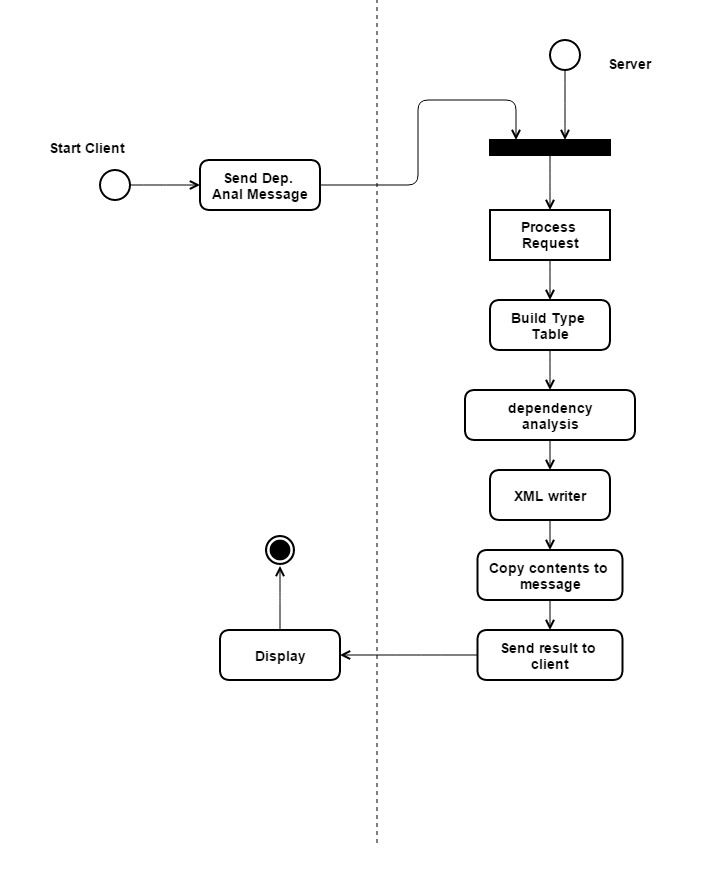


Figure 14 Activity diagram for dependency analysis

## 6.4 Uses

* + 1. Storage

All the packages and test suites can be stored in the repository server. It maintains not only the developing baseline code but also failed check-in code.

* + 1. Versioning

Repository server does versioning, which is explained clearly in the policies section of the OCD.

Versioning is maintaining records of the packages in the form of versions. If the file is same, the new file name is changed to different name with the version getting appended to file name and a new module with a version name is created and stored in this way. So several updated files can be stored without deleting old files.

* + 1. Dependency Analysis

Repository server handles query for dependency analysis. So in the client side, we can get dependencies present in between modules which help the developers to develop their code.

Testers get timely input from repository server and they need to perform the test cases execution. Results are sent back to repository server.

* + 1. Use for QA

Testers get timely input from repository server and they need to perform the test cases execution.

* + 1. Check-In and Check-out

The main use of this Continuous build and Integration System is to do check-in and check-out, which helps the developers develop a quality code as a team with proper communication channel. This check-in and check-out facility is supported by repository.

## 6.5 Issues

Since a lot of functionality is involved in the Repository server, issues are also present.

Some of the important issues are:

**Problem #1:** Large files takes lot of time in doing dependency analysis. While there are lot of requests coming to repository for check-ins and for all these requests also dependencies are to be found out, which results in delay.

Solution: Execution of dependency queries must be done in parallel using threads.

**Problem #2:** Search time may be large with versioning. Since repository is immutable, all the files are stored in it. It takes lot of time in handling check-out requests because time taken to search a file in the huge database is very long. Furthermore, time taken to search a package with different version further complicates the search.

Solution: Proper data structure is implemented while searching using metadata info. There should be a favorite folder for frequently checked-out files.

**Problem #3:** Complexity is very high since it does most of the functionality in the entire CBIS system.

Solution: Simple threading structure is to be adopted and should not make coding complex. Basically, its main priority is to not crash and send error message when something unexpected happens.

# 7. Build Server

Build server basic functionality is to take the source code module and compile it. Request to compile can come from Repository server while doing check-in because module have to be built and tested before check-in. The request can also come from Test Harness Server to get the latest build, when the client asks the Test Harness server to execute and send the test reports back.

Build server cache source code modules which is previously used for build.

When Test Harness server requests for the latest build, it checks in the repository whether the module and source code is same as previously built one or changed. Changed implies two conditions:

1. If the current module is changed
2. If the current modules depend on other modules and they are changed.

Both of the above conditions implies, there is a necessity to perform build again.

If there are no changes same build image is sent back to the Test Harness server. If there are errors in the build, error messages are sent to repository server and the Test Harness server.

## 7.1 Client on usage of Build server functionality.

Though messages are not directly sent from client to build server, doesn’t imply client is not depending on build server. Client sends check-in requests to repository and Test request to Test Harness server. For both cases, the server in order to achieve its goal, they will have to interact with the build server to get build images for the Test Harness server to test.

## 7.2 Package Diagram

Let us see the package diagram of the build server and discuss each of the partitions in detail.

Build server has following partitions or packages:

* Communication service
* Sender
* Receiver
* Error Handler
* Navigation handler
* Build Server executive
* Build tool
* Cache manager
* Cache

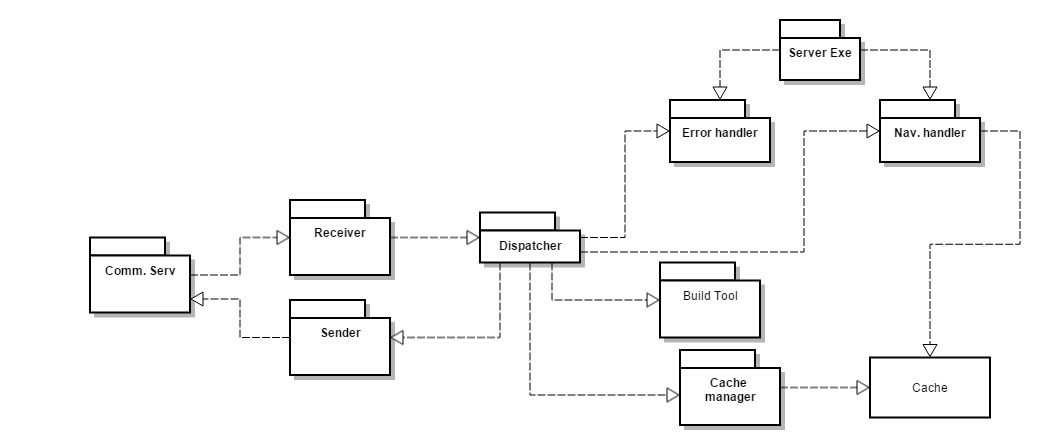


Figure 15 Package Diagram of Build Server

* + 1. Communication service

This package functionality is already discussed in the repository server. All the communication service packages are same and does create a channel for sending or receiving messages from the remote machines. Responsible for communication.

* + 1. Receiver

On reception of new message, the receiver package gives it to dispatcher.

* + 1. Sender

Sender package will posts the message to communication service.

* + 1. Dispatcher

Dispatcher routes the message to the desired package. It takes care of internal communication and it knows the functionality of each package and routes accordingly.

* + 1. Server Executive

Server executive is responsible to start the server and listen on a port number. It takes port number from the command line and starts the server. It is linked to Navigation handler and error handler.

* + 1. Error Handler

All the errors occurring in the build server are handled by the error handler.

* + 1. Build Tool

This tool provides the utility to compile and link various files. It creates the build image and gives it to dispatcher so that image can be sent to test harness server. After building, the source code module has to be submitted into the cache. It needs to be updated every time build tool builds the module.

* + 1. Cache Manager

This package manages the storage of source code which is built by the build tool recently.

It is connected to the cache.

* + 1. Cache

It is used to hold source code modules after every build. Cache manager updates the cache.

* + 1. Navigation Handler

When Test Harness server asks the build server for the latest build, navigation handler takes the request and navigates through the source code modules and extracts to give it back to the Test harness server.

## 7.3 Activity Diagram for Build server

Build server gets two requests. One from repository server and one from test harness server.

Each server is using build server in different way. Repository server is sending to build server for the case of knowing whether the package/ module sent is built and tested to do check-in.

Test harness server is sending to get the latest build to test the package mentioned in the test config file which is sent by the client to test harness.

### 7.3.1 Activity Diagram for Build server based on repository request

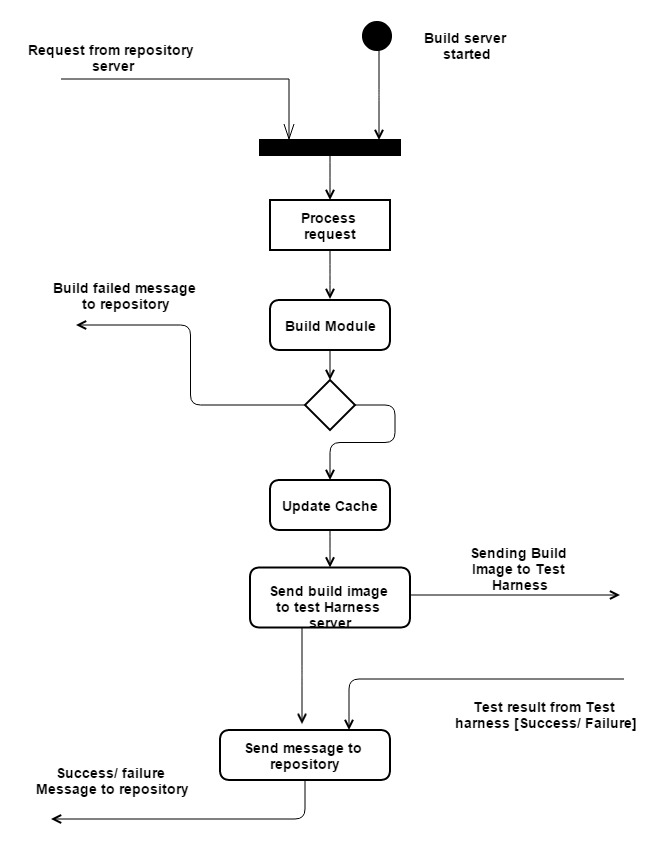
The main activities for the build server are:

* Request from repository server
* Process request
* Build module
* Update cache
* Send build module to test harness server
* Receive message from test harness server
* Send the message to client

Explanation:

The request sent from the repository is processed in the build server and builds the module. If it is failed, there is no need to send to test harness server, so failure message is sent to client.

Else if the build is passed, the source code module is sent to test harness for test and expects result message to send back to the client.

Figure 16 Activity diagram of build server based on repository request

### 7.3.2 Activity Diagram for Build server based on Test harness request

The main activities include:

* Request from the test harness server
* Process request
* Get Metadata
* Check metadata with client module
* Changed--> get new module and build again
* Unchanged -> use same module and build
* Send build image to test harness

Explanation:

This time the request comes from the test harness server, which gets processed. Later it check the metadata of the module present in Build server with the metadata of the module present in repository server. IF the metadata of the module in build server is not present or is different when compared to repository, the build server gets the complete modules and build it. It then updates the cache and gives the build image to test harness server.

Activity diagram is shown below:

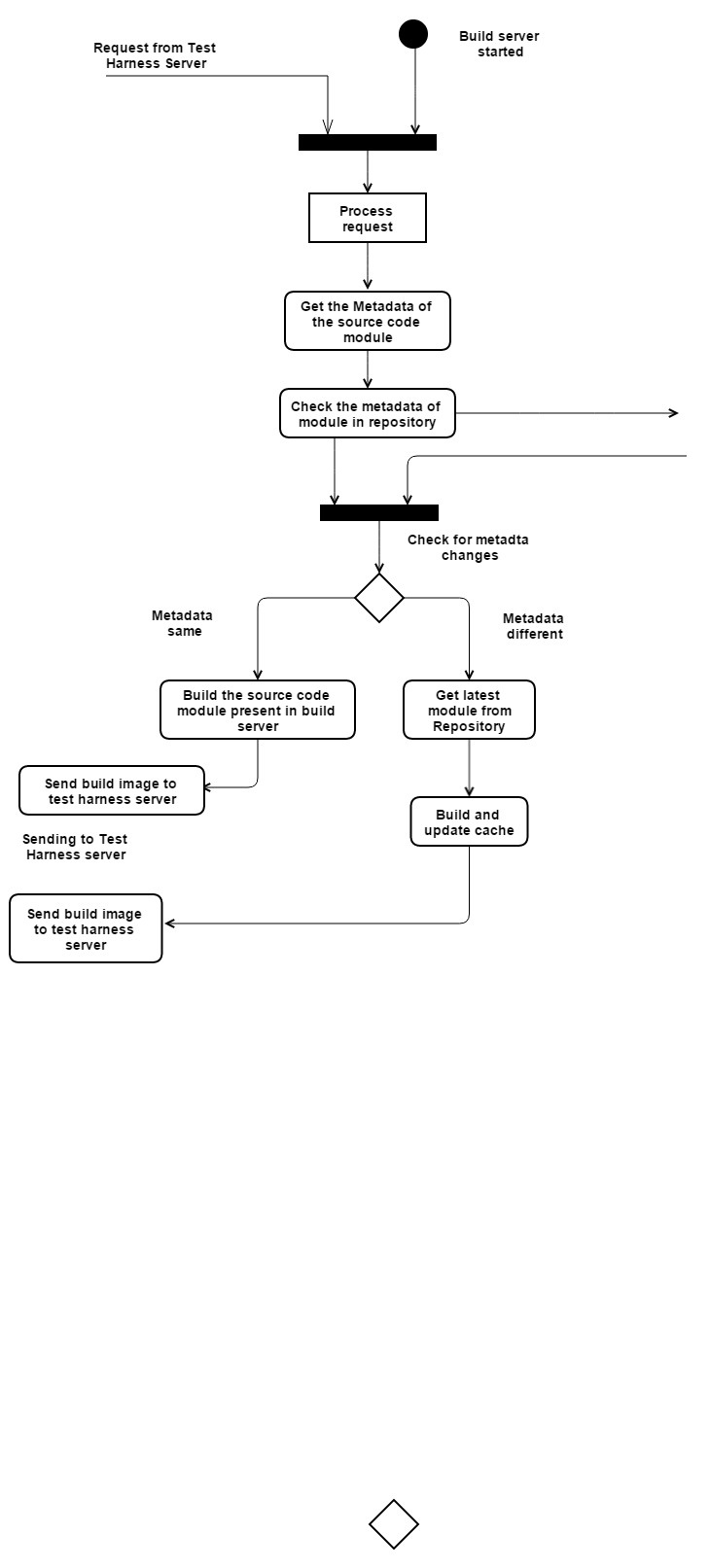


Figure 17Activity Diagram for Build server based on Test harness request

## Uses

7.4.1 Takes load off the developing system

The advantage of build server is that it takes load off the system and client can completely concentrate on developing code. This is useful when client system specification or resources are low.

* + 1. Usage of cache

Build server maintains cache the frequently build modules and can be easily retrieved when compared to repository server.

* + 1. Provides stable environment for builds

Build server job is to build using the build tool package. It mainly focuses on building and hence it is stable.

* + 1. Makes the job of QA easy

QA can go to location and get versioned builds within no time.

* + 1. Can be automated

Build server can further be automated.

* + 1. Easy way to find broken code before testing

If build server fails there is no need to test the file at all and hence it reduces complexity.

* + 1. Reduction of traffic by usage of cache

Cache helps in reducing traffic as the user can extract the source code module from the same server instead of getting it from repository server.

## Issues

7.5.1 Sometimes Build fails and gives wrong builds

Solution: Restart the build server on weird behavior. Assign a job for backup build server until this server comes up back and running perfectly

7.5.2 Build becomes Slow

Solution: Use different build server and decrease the load. Re configure and initiate the build again.

7.5.3 Loading modules from repository whose size is very large

Solution: Packages and modules needs to be sent from repository server to build server in chunks.

7.5.4 Although cache is maintained recompilation occurs frequently

Solution: Tools to support system building are usually designed to minimize the amount of compilation that is required. A unique signature identifies each source and object code version and is changed when the source code is edited. By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.

# 8. Test Harness Server

## 8.1 Importance

A software development team daily routine is to develop the code. Each time check-in fails, the bug is identified, fixed and tests are written to test new functions. So test config file along with the code needs to be supplied to some tool to test and report errors. Test Harness server is such kind of tool which helps the developing team to test the code regularly and get the reports. Based on the reports, quality code can be developed. Test Harness can also be used to test the module after successful build while performing check-in operation.

## 8.2 Description

Test config files can be sent from client to test harness server directly and results are logged and sent as reports to client.

A test config file gives information about the test drivers which needs to be loaded and tested. It also tells on what modules these test drivers are to be run.

Test Harness server holds test suites for each module and sub-system. Test suites basically contains collection of test drivers that are intended to test a software program. It also can contain input test file and expected output. Test harness server runs all tests or only tests which are specified and logs which all tests passed and which tests failed.

Test Drivers are used during Bottom-up integration testing in order to simulate the behavior of the upper level modules that are not yet integrated. Test Drivers are the modules that act as temporary replacement for a calling module and give the same output as that of the actual product. Drivers are also used when the software needs to interact with an external system and are usually complex than stubs. [Test driver description source: www.Tutorialspoint.com]

Hence upon request, it executes and send the message back to the client. Apart from individual testing, Test Harness server is also used to do tests whenever a file gets checked-in.

The Test Results can also be logged in a file stream to create a Re-use pack which may be used for subsequent tests. Each new test is created as a library and placed on a known path as a dll file. When the user runs the test, the system loads all libraries from that path and runs the tests defined by the loaded libraries. All the latest built images for modules are taken from the Build server. They are also in the form of dll.

Test Harness server supports multiple testing by running several app domains in threads.

Let us discuss what app domain is and how multiple testing is supported.

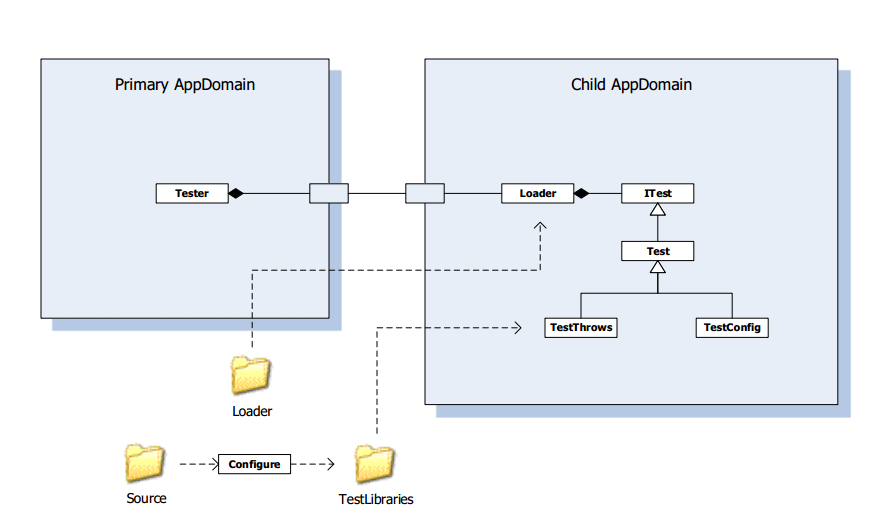


Figure 18 Test Harness structure provided by Dr. Fawcett

App domain provides an execution environment in which managed code runs. Basically it does memory access isolation i.e. all the applications running are isolated from each other at run time.

Diagram shows that the primary domain starts a child domain to load a loader into child domain. Similarly it can start multiple child domains and load reference is given for all child domains. We have already discussed the loader package in the activity diagram. It loads tests for processing. Loader implements ITest interface. Primary app domain knows loader type and not testing type. Use of testing multiple child app domains is if there is a failure of test, primary app domain is not affected.

\*\*\*Reference of description Dr.Fawcett notes on App domain.

## 8.3 Functionality

Its main functionality are:

* Load test suited from the xml messages
* Maintain cache of compiled modules
* Request latest modules from the build server that have changed since last cached
* Compile the tests and execute
* Return success notification, error notifications, and results log.
* Send selected notifications to registered developers for all modules that depend on the tested modules.

## 8.4 Package diagram of test harness server

The core contents of test harness package diagram are:

* Communication service
* Sender
* Receiver
* Test harness executive
* Logger
* Test driver
* Test suite repository
* Cache
* Core tester
* Test vector generator
* Loader
* Notification handler

Let us draw package diagram and discuss each of them in detail.

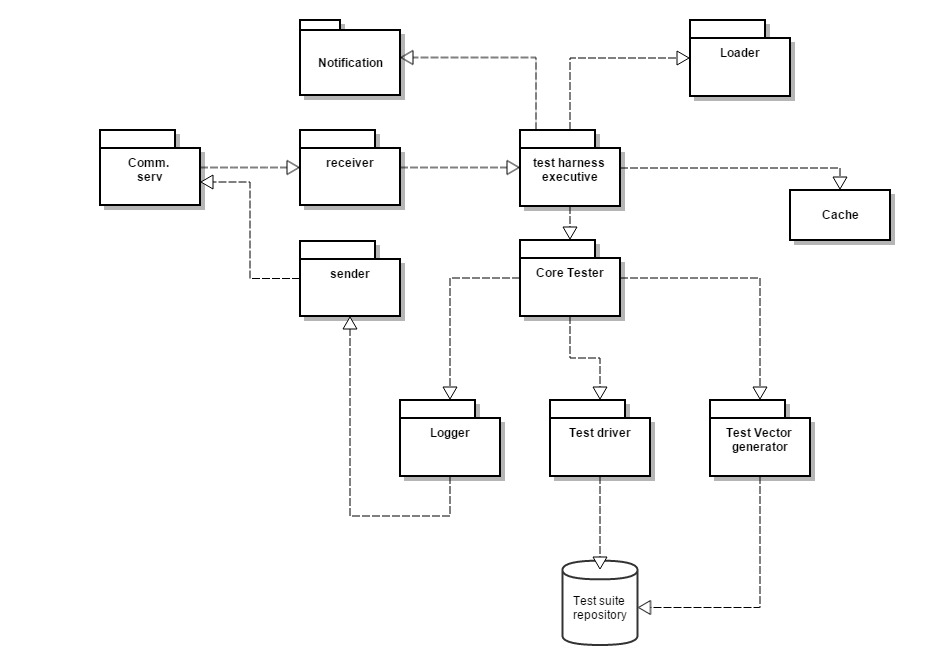


Figure 19 Package diagram of Test harness server

* + 1. Communication service

It is used to create channel for communication with remote machines

* + 1. Sender

This package sends the message to communication.

* + 1. Receiver

This package receives from communication service and gives to test harness executive.

* + 1. Test Harness executive

Test harness executive is the central part of the server. It starts the server by taking port number from the command line. It is responsible for internal communication between the packages.

* + 1. Loader

Each new test is created as a library and placed on a known path as a dll file. When the user runs the test, the system loads all libraries from that path and runs the tests defined by the loaded libraries. For this loading, we require loader package.

* + 1. Logger

A logger is provided to log the results. Test Harness server class diagram is provided in the appendix and if it is observed a logger basically has

* Console logger: It is used to output the buffered stream which consists of logged results to the console.
* File logger: It is used to log the results to a file and save.
* Memory logger: keeps tests results in memory for future use.

Logger result basically has

>description

>result

>other information related to the text.

* + 1. Core tester

Core tester is place where execution is done. It receives the test config file and asks the build server. It makes use of test vector generator, loader and test driver for executing. It gets build modules as dll from cache.

* + 1. Test suite repository

Test harness server holds the test suites for each module and for this we need this package.

* + 1. Test driver

Test driver maintains the data required to test the source code.

* + 1. Test vector generator

Test vector generator is generating test data for the test cases. Normally these data can be given in file say test suite.xml. It can also generate random test data for some test cases.

* + 1. Notification handler

Send selected notifications to registered developers for all modules that depend on the tested modules.

## 8.5 Test Harness Activity diagram

Activity of test harness server includes:

* Start client
* Start server
* Send test request message to test harness server
* Process message
* Read test config file
* Get test suite and build image of module
* Store test suite and dlls
* Load test case, input data and test module
* Execute tests
* Log results
* Send notification to selected users
* Send reply message to client
* Display

Activity diagram is shown below:

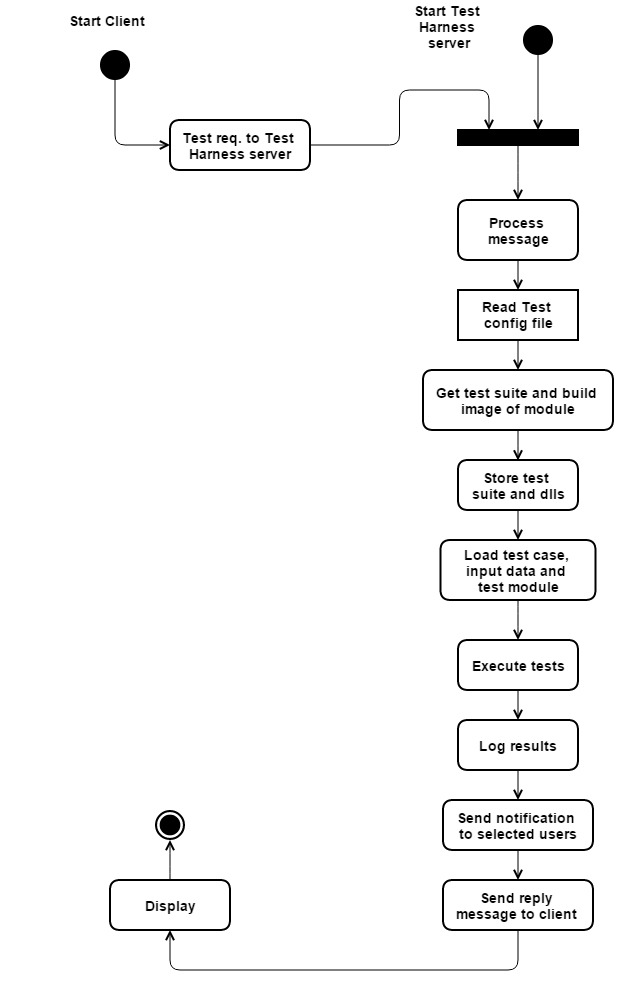


Figure 20 Activity diagram of Test harness server

Explanation:

Initially client and server is started and client sends test request and supplies test config file to Test Harness server. Test Harness server receives message and process it. It opens the config file and gets the module on which tests should be conducted. The dll of the module is already present in the cache of Test harness server. But it checks the metadata with the build server module and if the dll is latest, same is used for tests. Otherwise it gets new dll from build server and updates cache, which is further used for tests. The corresponding test suite for the module is loaded from the repository server and saved in the Test suite repository which is present in Test Harness server. The test vector generator generates data from the xml file of test suite and loads all the dll s of tests, These tests are conducted on the .dll files of the module. Execution is done and results are logged using logger. Notification package sends notification to all the users who are dependent on the tested module and server prepares reply message using logged results to the client. This is received on the client side and store as results.xml file. Later LINQ to XML queries are performed to display test results in the list box.

If the Test harness server is used for check-in process, then repository sends the package to build. Build server sends the build image to test harness server. Test harness server gets build image along with test suite. Tests are performed and its result is sent to build which in turn sends back to repository server. Basing on the status of the message received, it decides whether to check-In or not and sends the message to the client.

This is the activity diagram of Test Harness Server.

## Uses

8.6.1 Provides platform to develop quality code by continuous testing facility

New features can be added by the developer along with different test config files. Errors can be checked, fixed and re tested.

* + 1. Provides cache for fast accessing of file

Test harness server provides cache to load the module and do testing. Hence the performance is good.

* + 1. Provides notifications to all selected users

If a module is tested and failed, all the users who are dependent on the module gets notifications because of Test harness server.

* + 1. Logger facility

Logger facility can be used for future use to check back the results and analyze the error patterns, so that same errors are not repeated again.

## Issues

* Issue with order of selecting XML config files if there are more than one config file

Solution: A simple solution is to select xml config files randomly and execute.

* Handling test result logs. Issue can be with file name or storage or transfer.

Solution: Proper naming should be done so that rewrite of log files doesn’t take place. A random number should be generated and attached to the directory name which was tested.

# 9. Communication and services in the entire system

This section of OCD explains about how communication takes places between machines which are located remotely. Here, we use Windows Communication Foundation for interaction between clients and servers.



Figure 21WCF communication between Client App and Service Host

WCF is a framework for building service oriented applications. Using WCF, you can send data as asynchronous messages from one service endpoint to another. A service endpoint can be part of a continuously available service hosted by IIS, or it can be a service hosted in an application. An endpoint can be a client of a service that requests data from a service endpoint. Figure tells that each endpoint consists of four properties:

* An address that indicates where the endpoint can be found.
* A binding that specifies how a client can communicate with the endpoint.
* A set of behaviors that specify local implementation details of the endpoint.
* A contract that identifies the operations available.

The messages can be as simple as a single character or word sent as XML, or as complex as a stream of binary data. Figure 3 shows that a Client App is communicating with the Service Host using WCF. Hosts may have multiple end point and can accept multiple requests. We use Message passing as communication channel.

**Message Passing:**

* Sends message with encoded request and/or data
* Message contains endpoint information for routing
* Directly supports asynchronous processing
* Examples: Internet

Client and Server has type safe message blocking queue at their receiving ends. So each message goes into the queue until it is full. Service is hosted and client sends requests to the end point of the service, whose address is known and gets the reply back.

Sample code for WCF is given in the appendix. All the messages are object for data member contracts. Every message is in the form of xml as string. On receiver side, the string is converted back to file and can be used.

Communication sample:

=======================code===============================

namespace WCF\_Peer\_Comm

{

[ServiceContract]

public interface ICommunicator

{

[OperationContract(IsOneWay = true)]

void PostMessage(SvcMsg msg);

SvcMsg GetMessage();

}

[DataContract(Namespace = "WCF\_Peer\_Comm")]

public class SvcMsg

{

[DataMember]

public string src;

[DataMember]

public string client\_src;

[DataMember]

public string dest;

[DataMember]

public string body;

[DataMember]

public List<string> direc;

[DataMember]

public string cmd;

[DataMember]

public string[] directories;

[DataMember]

public string dir\_anal;

}

}

The above code represents Service contract implemented for my project 4.

Contracts defines content of messages. A service will publish a contract that specifies what methods client can call. OnewayTrue represents whether the operation returns a reply.

[ServiceContract] defines the service’s contract. It defines name of the service. It is an interface with methods which are recognized with [OperationContract] tag.

A data contract is an agreement between a service and a client that abstractly describes the data to be exchanged. That is, to communicate, the client and the service do not have to share the same types, only the same data contracts. WCF uses a serialization engine called the Data Contract serializer by default to serialize and de-serialize data.

Endpoints:

All communication with a Windows Communication Foundation (WCF) service occurs through the endpoint of the service. Endpoints provide clients access to the functionality offered by a WCF service.

Each endpoint mainly consists of:

* An address that indicates where the endpoint can be found.

Declaration: string address = "http://localhost:8000/HelloService/MyService";

* A binding that specifies how a client can communicate with the endpoint.

Example: BasicHttpBinding binding = new BasicHttpBinding();

* A contract that identifies the operations available.

Channels are the vehicles that transport messages. Transport protocols via binding.

Proxy: After you create a service, the next step is to create a WCF client proxy to communicate with the service.

Building proxy programmatically:

* WSHttpBinding binding = new WSHttpBinding();
* Uri address = http://Odysseus:4040/ICommService;
* ICommService proxy = ChannelFactory<IContract>.CreateChannel(binding, address);
* SvcMsg msg = new SvcMsg();
* msg.body = “a message”;
* proxy.PostMessage(msg);

Here body is data member.

Similarly all the xml content is read as string and stored by msg.body=xml.tostring();

Where xml is a document which has contents in xml format and the msg is sent through communicator. Other side remote machine receives same message and access body by msg.body and saves the contents as xml file. In this way messages are communicated.

\*\*Reference of explanation: Microsoft website and StackoverFlow.com and Dr.Fawcett ppt

# 10. Policies

In this section, we will be discussing about

* Ownership,
* Notification,
* Versioning
* Check-In
* Check-out
* Build policy based on dependencies

10.1 Ownership

* Here we are implementing Closed check-in policy with multiple owners

Explanation: Check-in is not open i.e., everyone cannot modify the file and check-in. Furthermore it is not single owner policy either. This CBIS OCD discussed so far has multiple owners who are working as a team in developing a module. Only owners with team ID have rights to edit and check-In.

* Every file has a user

Explanation: If a team consists of two people and if a person leaves the group. Ownership is transferred to other person. If both of them leaves the group, the complete rights are with Software architect until he gives privileges to a different team.

* Meta data edit

Explanation: Editing the metadata of the file requires Software manager/ architect authentication.

## 10.2 Check-Out policy

* If a file is already checked-out, a lock is implemented and other clients have to wait for the lock to open. The lock opens only after recheck-in. A copy of the file/package is always present and the users in that group can always download even after logging out.
* Check-out always needs user authentication to know whether user is correct person to access and edit the file.
* If check-out is successful, it is user duty to check-in again.
* If check-in is not done in a particular time, notification is sent.
* Only one user can check-out at a time

## 10.3 Check-In policy

* For a package to get checked-in, it has to build and tested successfully.
* If build or tests are failed, then check-in is failed
* Even though check-in is failed, the source code gets stored in the repository but is not a part of developing baseline code.
* Authentication is needed for check-in.

## 10.4 Notification

Notifications are mails sent to the user or software manager. Notification occurs in three situations.

* When check-in or check-out fails
* When user checks-out and doesn’t check-in within stipulated time.
* In the test Harness server, when the tests are executed and dll files are produced, notifications are sent to all the users whose modules are dependent on the tested source code module.

## 10.5 Versioning

Explanation: If the file is edited and checked-in, it will have same metadata other than last access and modified time. Package with same name is already existing in the repository and it cannot be deleted as we have discussed that repository server is immutable in the organizing principles. So there is a possibility that files gets checked-out and again gets checked-in, it is better to store the packages as versions. The module name is created with different version name but they are stored in the same sub-system.

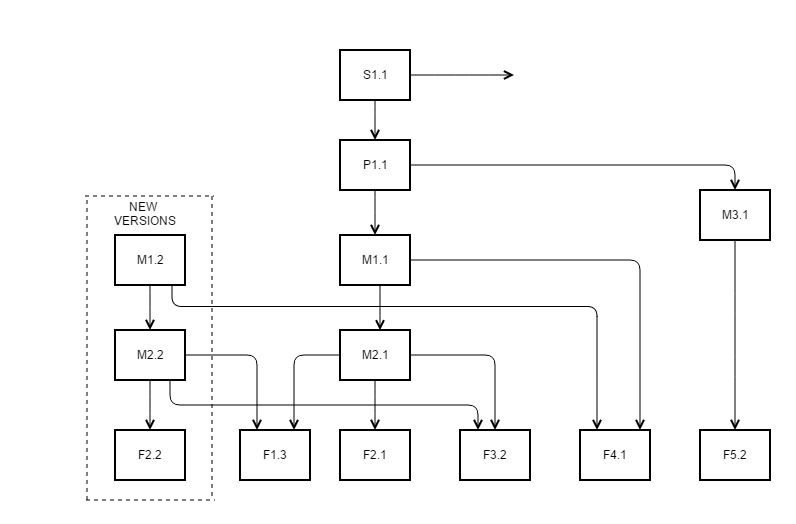


Figure 22Versioning Concept Diagram given by Dr.Fawcett

This kind of storing same edited packages in the forms of versions is called versioning. So every package has version 1, version2 etc. depending number of times the same package is checked-in.

**Explanation of versioning diagram:**

F in the versioning diagram implies file and M implies module, S implies System.

We can see a new version of F2.1 is created or checked-in, which is named as F2.2.

This is the edited or modified new F2.1 file and therefore we can see that older versions are retained in the repository. By this the developers can have data of the older version in case there is a problem with the new version.

When F2.2 file is created, it cannot be saved to same module and its version also gets changed to M2.2. Moreover M1.1 is dependent on M2.1, so we therefore have M1.2 version. As we can see that files F1.3 and F3.2 are also copied into new module M2.2 but instead of F2.1 we have F2.2 and this is the only difference with older version module and new version module

In this way system maintains multiple files in the form of new versions for every new update.

## 10.6 Build policy based on dependencies of modules/ packages

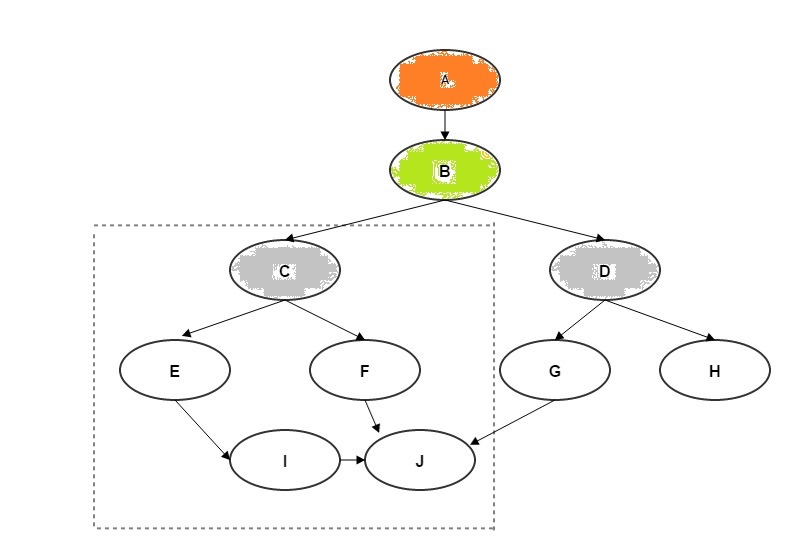


Figure 23 Dependencies of packages and storage structure

Description:

Here we have A as system and B as sub-system.

Sub-system has one or more modules and given modules are C and D.

These modules has packages for instance C has E, F, I, J packages and D has G, H packages.

The figure represents package dependencies within a module and between two modules.

Package F is dependent on J and both are in same modules but package G is dependent on J which is present in completely different module. Whenever there is a change in module for suppose J, all the users who created packages F, I, G should be notified by the test harness server that there is change in the package which you are dependent on, so that users can build and test their code using Test harness server without having to check-in the code again. Similarly Build Server also checks the dependencies and builds from bottom up approach i.e., if Repository server asks to build package E in module C, build server checks the cache for source code packages for bottom level dependent files. If any of the bottom level packages are changed, then build is performed again. After that the actual build of package E is done. Modifications in the bottom level packages are known by checking the metadata of the packages.

## 10.7 Storage hierarchy in the repository server

Repository server define modules as lists of packages contained in the Repository store. Also defines subsystems as collections of modules contained in the Repository store.

So every system can have one or more sub-systems.

Every sub-system can have one or more modules

Every module has one or more packages.

It is perfectly fine to have a system which is empty.

Whenever a check-in or a check-out is handled, the packages gets checked-in this hierarchy or checked-out by searching in this hierarchy respectively.

# 11. Conclusion

Continuous Build and Integration System can be built based on this OCD. Developers needs to keep track of the issues given in the OCD and develop the code for this system. Uses and architecture of overall system is explained at the start of OCD. This CBIS processing is divided in to partitions and explained at more detailed level. User Interfaces are provided for client to help the developers. This CBIS can further be extended by taking multiple repository servers, multiple build servers, and multiple Test harness servers. Communication becomes complex in this scenario. Build server can be automated. Prototype code for cache in the build server is implemented and results are shown in appendix and thus this architecture can be implemented if carefully developed.

# 12. References

* Wikipedia
* Dr.Fawcett presentations in class
* [www.msdn.microsoft.com](http://www.msdn.microsoft.com)
* [www.stackoverflow.com](http://www.stackoverflow.com)
* Some part of the Test harness description is written by understanding the Test harness OCD of Vijayanand Appadurai, which is provided in the OCD survival guide.

# 13. Appendices:

## 13.1 Class diagram of Test Harness server

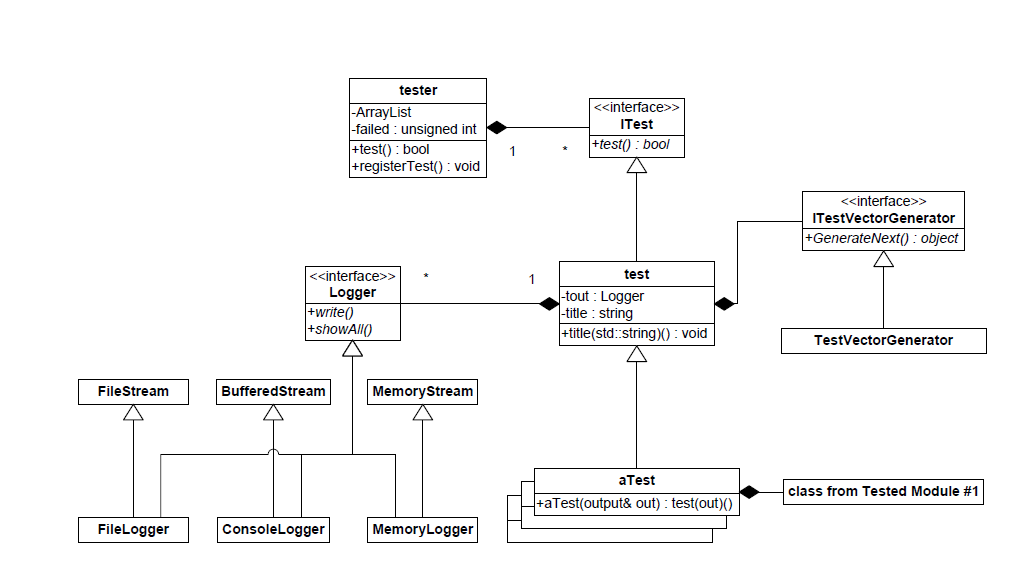


Figure 24 Test Harness Class Diagram provided by Dr.Fawcett

The core contents of class diagram are:

* Tester: it is tester class
* Itest: This interface needs to be implemented.
* Test: It contains Itest and test classes

ItestVectorGenerator is an interface to implement. This interface if implemented can generate data for the test cases.

A logger is provided to log the results. A logger basically has

* Console logger: It is used to output the buffered stream which consists of logged results to the console.
* File logger: It is used to log the results to a file and save.
* Memory logger: keeps tests results in memory for future use.

## 13.2 Prototype

From the preliminary architecture concept, it is given that Build Server Cache source code modules previously used for builds. So Build server needs to check the files coming from repository if they are present or not. If they are not present, get the package and build it. It should also build all dependency packages and modules but image obtained by building the package sent by repository is sent to Test harness server. My prototype concentrates on whether the package is present in cache or not. This prototype assumes cache as “Savedfiles” folder. If the file is present in cache, we need to check metadata and based on metadata downloading file from repository is decided but as of now we don’t have metadata generator ready and because of this, my prototype avoids checking metadata and doesn’t download from repository if the file is already present in cache.

/////////////////////////////////////////////////////////////////////////////

// Client.cs - WCF Timed, SelfHosted, File StreamService client //

// Implementing above client.cs as build server functionality //

// Author: Venkata Karthikeya Jangal, //

// MS in CE, //

// email; vjangal@syr.edu //

//Code developed by modifying the code of //

//"Client.cs - WCF Timed, SelfHosted, File StreamService client" //

//developed by Dr.Jim Fawcett, CSE681 - SMA Instructor //

////////////////////////////////////////////////////////////////////

/\*

\* Note:

\* - Uses Programmatic configuration, no app.config file used.

\* - Uses ChannelFactory to create proxy programmatically.

\* - Expects to find ToSend directory under application with files

\* to send.

\* - Will create SavedFiles directory if it does not already exist.

\*/

// file is last modified on 12/8/2014

// Description: Here client acts as a BUild server and Service acts as a Repository server

// Whenever build server{client in this situation}, receives a package to build it checks its cache which is the

// savedFiles folder in the current context. If the file is not present then it downloads from the repository server.

// If the file is present, it checks the metadata with metadata of same package present in repository and downloads

// if metadata of both packages are different. Here metadata cannot be checked as this is prototype and hence if

//the file is present it avoids downloading. Concept is if the file is present in cache, avoid download from repository and

// uses same file to build and send image to the test harness server.

using System;

using System.IO;

using System.ServiceModel;

using System.ServiceModel.Channels;

namespace CSE681

{

class Client

{

IStreamService channel;

string ToSendPath = "..\\..\\ToSend";

string SavePath = "..\\..\\SavedFiles";

int BlockSize = 1024;

byte[] block;

HRTimer.HiResTimer hrt = null;

Client()

{

block = new byte[BlockSize];

hrt = new HRTimer.HiResTimer();

}

static IStreamService CreateServiceChannel(string url)

{

BasicHttpSecurityMode securityMode = BasicHttpSecurityMode.None;

BasicHttpBinding binding = new BasicHttpBinding(securityMode);

binding.TransferMode = TransferMode.Streamed;

binding.MaxReceivedMessageSize = 500000000;

EndpointAddress address = new EndpointAddress(url);

ChannelFactory<IStreamService> factory

= new ChannelFactory<IStreamService>(binding, address);

return factory.CreateChannel();

}

void uploadFile(string filename)

{

hrt.Start();

string fqname = Path.Combine(ToSendPath, filename);

using (var inputStream = new FileStream(fqname, FileMode.Open))

{

FileTransferMessage msg = new FileTransferMessage();

msg.filename = filename;

msg.transferStream = inputStream;

channel.upLoadFile(msg);

}

hrt.Stop();

Console.Write("\n Uploaded file \"{0}\" in {1} microsec.", filename, hrt.ElapsedMicroseconds);

}

void download(string filename)

{

int totalBytes = 0;

hrt.Start();

try

{

bool flag = true;

string[] files = Directory.GetFiles(SavePath);

foreach(string f in files)

{

string k=Path.GetFileName(f);

if (k == filename)

{

flag = false;

break;

}

}

if (flag)

{

Stream strm = channel.downLoadFile(filename);

string rfilename = Path.Combine(SavePath, filename);

if (!Directory.Exists(SavePath))

Directory.CreateDirectory(SavePath);

using (var outputStream = new FileStream(rfilename, FileMode.Create))

{

while (true)

{

int bytesRead = strm.Read(block, 0, BlockSize);

totalBytes += bytesRead;

if (bytesRead > 0)

outputStream.Write(block, 0, bytesRead);

else

break;

}

}

hrt.Stop();

ulong time = hrt.ElapsedMicroseconds;

Console.Write("\n Received file \"{0}\" of {1} bytes in {2} microsec", filename, totalBytes, time);

return;

}

Console.Write("\n File is not received because {0} file is already present in the cache",filename);

}

catch (Exception ex)

{

Console.Write("\n {0}\n",ex.Message);

}

}

static void Main()

{

Console.Write("\n Build server running ");

Console.Write("\n ==========================================\n");

Client clnt = new Client();

clnt.channel = CreateServiceChannel("http://localhost:8000/StreamService");

HRTimer.HiResTimer hrt = new HRTimer.HiResTimer();

hrt.Start();

clnt.download("ICommunicator.cs");

clnt.download("Semi.cs");

clnt.download("Parser.cs");

clnt.download("Toker.cs");

clnt.download("Communication.svc.cs");

clnt.download("Semi.cs");

hrt.Stop();

Console.Write(

"\n Total elapsed time for downloading = {0}",

hrt.ElapsedMicroseconds

);

Console.Write("\n\n Press key to terminate client");

Console.ReadKey();

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

((IChannel)clnt.channel).Close();

}

}

}

Output:

Fig1 and fig2 shows the start of Repository server and build server respectively.

Figure 3 shows that file is already present in cache and hence it is not downloaded from repository server.

Figure shows file is not present and hence it is downloaded from repository server.

It also shows repository server sending file.

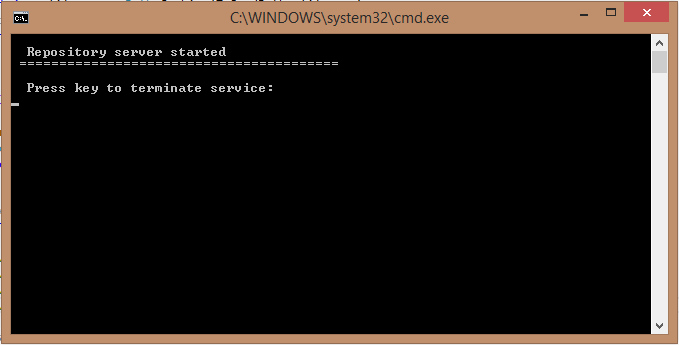


Figure 1

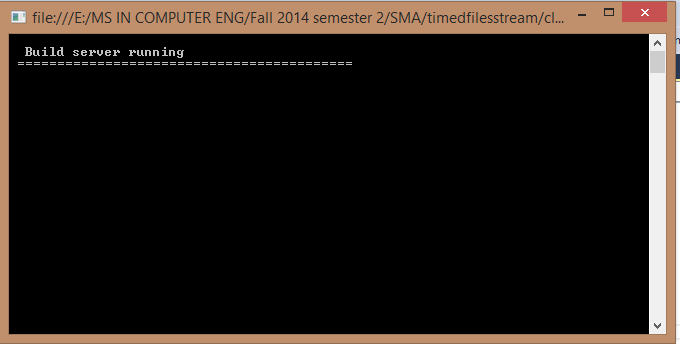


Figure 2

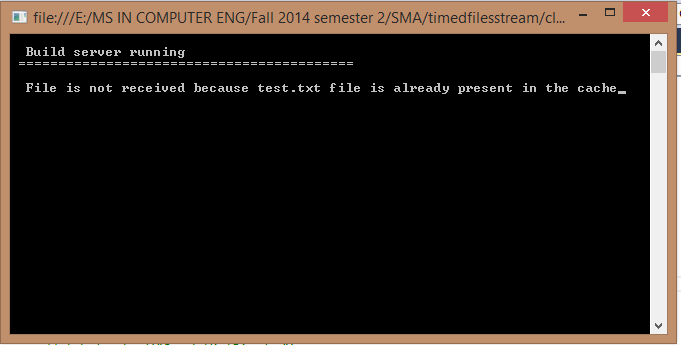


Figure 3

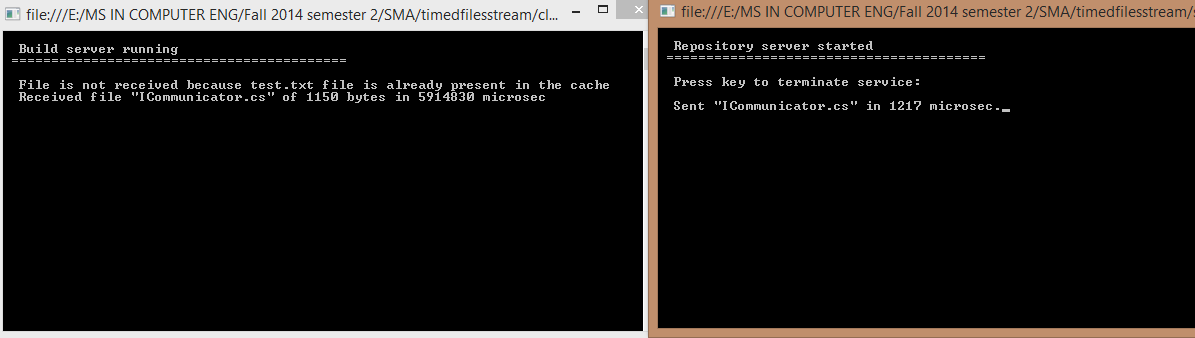


Figure 4

These are the output views.