CONTINUOUS BUILD AND INTEGRATION SYSTEM

ARCHITECTURAL CONCEPT DOCUMENT

CSE 681 – SOFTWARE MODELLING AND ANALYSIS

PROJECT #5

DOCUMENT VERSION 1.0

INSTRUCTOR: DR. JIM FAWCETT

DHIVYA NARAYANAN

SUID: 877721012

FALL 2014

DATE: 12-08-2014

Table of Contents

1 Executive Summary

The main purpose of the continuous build and integration system is to integrate the code into the shared repository as soon as the developers developed the code. This system is very important in software industry, to develop and execute large projects which involve hundreds of developers work. Developing such a large project is quite challenging and has lots of interdependencies between the codes developed by several developers. This system helps developer to reduce the risk of making wrong assumption about interface and functionality of the package developed by other, when large number of developers working on the common set of goals and developing packages which have lots of complex interdependencies.

This document explains the architecture of the continuous build and integration system which can continuously integrate the developer's code with the rest of the software baseline for the software and hardware environment. This system consists of the following components

1. Clients
2. Repository server
3. Build server
4. Test harness server

The primary users for all the above components in this system are Project manager/team leaders, software architects, developers, test team and maintenance team. This system is mainly used to integrate the all the codes developed. The Projects managers could use this system to monitor the quality and functionality of the project by testing. A software architect could use while working on subsystem of an existing system. The testing team uses this system to test the builds.

A brief description about each component, the uses, the context where it is used, the package diagram and the activities of each component and the critical issues associated with each tool are described clearly with the necessary diagrams in the further sections.

The overall architecture of the system which has above components is explained in detail in next section. The repository holds as a storage place for all the developing units to check in and check out the code. Also queries can be made on repository and source code files or their metadata files can be extracted by user. The code files that are checked in to the repository need to be tested both individually and as a system. The build server is to compile the codes which are needed to execute tests based on request from the test harness server. The test harness server holds test suites for each module and subsystem and test the builds on demand. Each client has GUI that send request message to the servers to update the code in the repository or to build or test

the developed code. It acts as a user interface for this system for developers and managers. The client checks-in the source modules to the repository and check-out source codes from the repository server and the gets the test messages and result logs from the test harness server.

Some of the critical issues associated with this continuous build and integration system that need to be considered while implementing are as follows:

- large number of stored documents
- What if two projects have package/file with same name?
- Query result size may be large
- Search time may be large for some queries
- What happen when occasionally two concurrent clients may attempt to query same file at same time?
- When a file is extracted there is a chance of file size being large so that the network cannot get the whole file string at a time.

2 Architecture of the system

The architecture of this continuous build and integration system deals with client and server components. The server components consists of three stand-alone tools into a single environment, each perform different operations independent of each other and also interacting with each other by sending and receiving request messages. The main three server components are

- Repository server
- Build server
- Test harness server

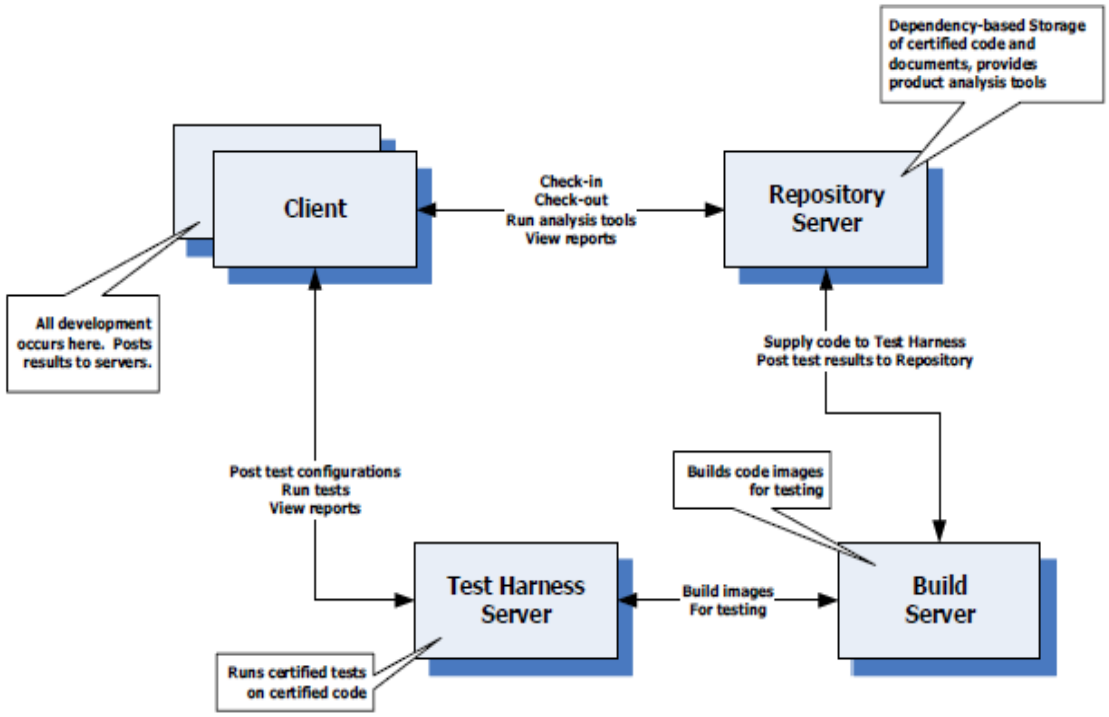


Fig 2.1: Architecture of the CBIS system – Referred from Project#5 requirement

Client:

The client provides graphical user interface into the continuous build and integration system for developers and managers. Graphical User Interface is a type of user interface which has some graphical icons or visual indicators or tabs. The client will be designed in such a way to handle user requests and communicate with the server to send source codes to the server. Each server components interact with the client, receives inputs like source files, queries, xml messages from the client and return the results to the client using windows communication foundation (WCF). The main functionalities of the client is to create and send messages to the repository and test harness servers, get the results from the servers and display it in GUI, check-in the source code files to the repository server and to download the source code from the repository server for check-out and get the test messages and result logs from the test harness server. All the communication between the client and the server is implemented using WCF.

Repository server:

The repository server maintains all source code files, their metadata files. It stores large amount of source code are kept, either publicly or privately. It gets the source code from the client. This repository plays important role in multi-developer projects to handle various versions and developers stores various patches of code. Here, modules are referred to list of packages in the repository and the subsystems as collection of modules in the repository. The Repository would hold all of the software, test results, and documents that have been checked in and added to the project's current baseline. The repository supports check-in, check-out, extraction, and versioning of all products. It also provides analysis tools such as dependency analyzer to analyze the modules and subsystems present in the repository and find the relationships between them. To maintain the integrity of modules, they are accessed through interface and object factory. Some of the important components in the repository server are executive, parser, file manager, versioning manager, check-in, check-out. It also has to execute queries to get dependency relationships. One of the important functionality in the repository server is to manage the check-in and check-out of packages source code to and from the modules.

Build Server:

The build server receives the dynamic link libraries(dll) from the repository server which needs to be compiled. It compiles any packages or modules to execute tests when it receives request messages from the test harness server. It accepts the build request messages from the test harness server and build the images and send back to the test harness server if the build is successful. In case of error, it sends the error

notifications/messages to the repository and test harness servers. The output of the build is stored in cache as well, which will avoid recompile the same code again. This server accepts the dynamic link library (dll) files, compile it and gets the executable and build it. The build server uses the dependency based builds. The build server requests for code file to the repository server. Then the repository server runs the dependency analyzer and get the information about the dependent file. The build server builds all the dependent files along with the file which has to be build and tested. Then all the build will be stored in the cache.

Test- Harness server:

The Test-Harness server holds test suites for each module and subsystem in the repository server. As each new module of a project is developed, one or more test classes are developed simultaneously and incorporated into the test suite. The compiled modules are placed in the cache. Based on XML messages receive, it loads the test suites which are the sequence of test cases for the particular module. It executes the test suites on demand. It will compile and execute the current tests and returns the success notification, results log the client, if the build is successful and in case of build errors, it returns the error notifications.

The continuous build and integration system is very useful in software industry where hundreds of developers working together on the large projects. Its users include developers, testing team, project manager, quality assurance team, etc.

The flow of this continuous build and integration system is shown in the above figure and the detailed explanation about each component is given in further sections.

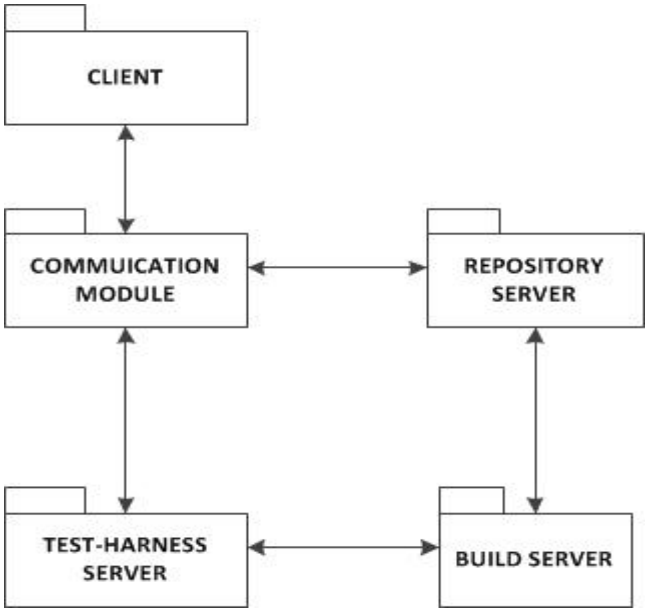


Fig 2.2: CBIS system - Package diagram

3 Users and Uses Cases

3.1 Primary Users

Continuous Build and Integration system plays very important role in software industry, where hundreds of developers working on the common set of goals for large project and developing code that have complex interdependencies with the code develop by others. CBIS system can be used by various users. In this section, we discuss about the primary users and the features provided by this system for its users. The possible users of this system are as follows:

1. Developers

   Developers work with large projects which has hundreds of codes developed by many developers and also the code they develop may have complex interdependencies with other packages. They use this system to check in code and they can extract or view the code files for reference. They check out code files to update the code with their changes. They get the test results as soon as tests are performed on code they checked-in. They are allowed to view the code developed by others if they authenticated.

2. Quality assurance team:

   Primary work of the quality assurance team is to ensure that the system is properly designed to meet all the requirements. They will run the tests on the files which are stored on the repository. Quality analysts check the errors in the system using the build server and make sure that the system should not have any bugs or inconsistency while it reaches the customers.

3. Project/Team manager:

   Project Manager can use this CBIS system to get all the developed packages or modules with currently working software baseline. It also can be used to get the file information and dependencies to keep track of the progress. They can analyze the some specified projects in specified server. They can manage the whole system by modularizing it.

4. Administrators:

   Administrators can use this repository server to manage all the developed code in single place. They can find the dependencies between types and/or packages and can help them to manage the overall system design.

3.2 Use cases

1. To integrate the develop code to the currently working project baseline

When the developers developed the new code file, they can immediately integrate into the current project baseline. This system stores all the code files which are related to the project baseline into the repository.

2. To view the code files which are developed by others

This system allows the developers to view the source code files developed by others with which their code is dependent. It helps them to develop the code according and to meet their functionality requirements.

3. Build and tests the code file whenever it is checked-in

When the new code file is added to the repository, the repository server send request to the build server which in turn compile the source code files and send the build images to the test harness server for testing. Then the test harness server tests the code file with the test cases developed for that file and stores the results in the logger and returns the result reports

4. To find the dependency information

The source code files in the repository are stored on the basis dependencies between them. Their metadata files also contain the information about dependency. So that the user can send request to the repository server and get the dependency information of the file.

5. Get the metadata files

By sending the request to the repository server, the user can get the metadata files which contain all the information about the files. All the metadata files are stored in the repository.

6. Stores all source code files and metadata files in single place named repository

All the source code files, metadata files, test cases and the test results related to the currently working project baseline are stored in the repository. So that one can get all the information about the project easily

7. Stores all the versions of the files, so that roll back can be done

When the code file is updated, the repository stores the old version and new version of the file in different folder. So that if one wants the older version to be added to the baseline in future, they can roll back it.

4 Critical Issues

Some of the critical issues related to CBIS system are mentioned below. The detailed explanation is given in the respective module sections

Issues with **Repository server**:

1. Ownership configuration
2. Same file names to different files
3. Security Issues
4. File transfer with chunking
5. Performance due to large number of files in the repository

Issues with **Test-Harness server**:

1. Performance due to large number of test cases
2. Testing of test harness itself
3. Supply of invalid data/inputs by test vector generator
4. Request from several clients to test the code files at the same time

5 Client

5.1 Introduction

The Client is the Graphical User Interface layer which is used to communicating with the users. It provides some tabs/panels for the user to perform some tasks. There are some tasks associated with the client module and those tasks are performed by different modules in the client side. The main functionality of this client module is to create and send source codes that are newly created by developers to the repository server where all the source code files are stored, for check-in. It also allows the developers to retrieve the codes from the repository server to update or to view the codes, for check-out. In order to test the code or execute some queries about dependency relationships, the client GUI creates and sends XML messages to the repository server and in turn it will do the corresponding action and the client receives and displays the messages from the repository server. It also sends message to do testing on the source code that the client checked-in. So, the repository server send the dll of the source code to build server, which in turn send the build images to test harness server and the testing part is done there and the client gets the test messages and result logs from the test-harness server. In order to implement the Graphical user interface, the client uses Windows presentation foundation (WPF). All the functions like sending and receiving messages, uploading and downloading the source code will be done using the tabs/panels in the GUI. Some of the tabs/panels at the client GUI are:

- Upload/check-in
- Download/check-out
- Send message
    - To repository server
    - To test-harness server
- Download result logs and test messages

5.2 Users

The primary users of the client module are:

1. Developer
   A developer is the primary users of this client module. They develop the code for the current working project baseline, checks-in it into the repository and checks out the code from the repository server.

2. Tester
   A tester uses the system to make and deploy test cases for the projects. They develop the test cases and pass it to the test harness server, which perform the tests on the code and gives back the result of the tests.

3. Project Manager/Team Leader
   A project manager can use this module for managing the project. For the manager, the client module mainly interacts with the repository server to get the dependency relationships between the packages.

5.3 Partitions

The Client can have the following modules:

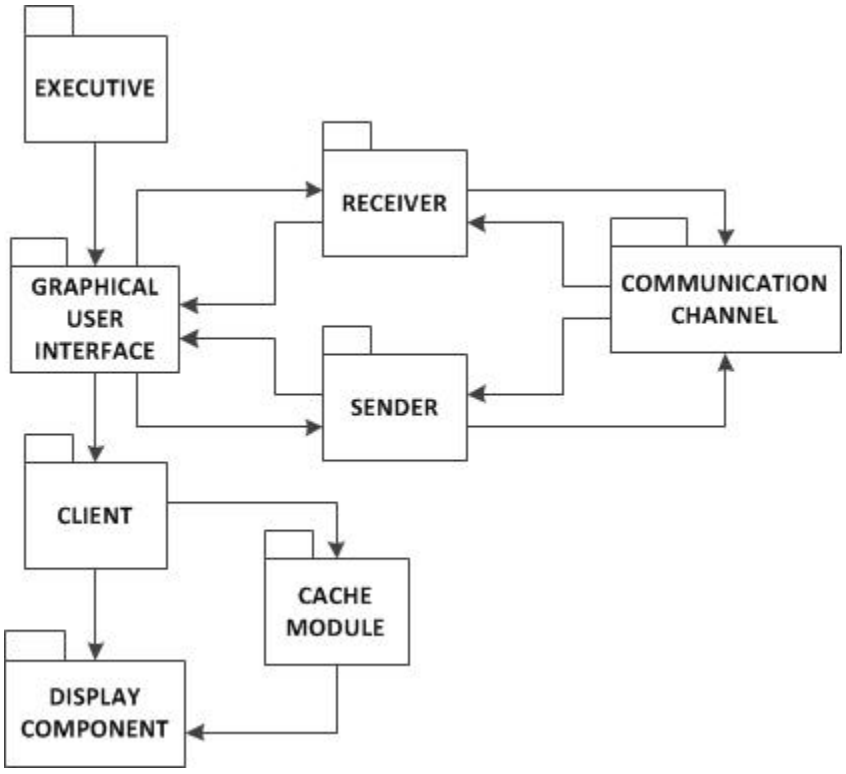


Fig 5.1: Package Diagram for client module

Executive module:

It is the main entry point of the system and plays an important role in the client. It is a small package which contains the main function for the execution of the client module. The main task of the executive module is to load the graphical user interface which helps the user to interact.

Graphical User Interface:

This module provides the user interface through user can communicate. The graphical user interface will be implemented using Windows Presentation Foundation (WPF). It consists of several interface tabs/panels which is shown in the main window of the GUI module. It helps the user to perform any things using the data displayed on the user interface. Some of the important tabs/panels are upload button which helps to upload the file from the local machine to the repository server, download button which gets the specified source codes from the repository server. When we click the download button, it will list all the projects available in the repository server and we have to specify the source code which we want to check-out for update or to view. Generally the check-out is followed by check-in. Once we updated the source code which we downloaded, then have to check-in the newly modified one into the repository server. The GUI also has tabs to send or receive messages to/from the repository server and test-harness server.

Cache module:

Cache mechanism for the client will be done at cache module. It stores the files to improve performance of the system. If the output to user requests is in the cache, then doing the process again in the server will be reduced.

Sender module:

This module is used when the clients send messages to the server. It contains the sender address and receiver address, commands, etc. It has two main components namely queue which hold the client requests and Threads which sends the requests to the server using communication module

Receiver Module:

This module is mainly to receive the response from the server. It also have queue and thread which hold the receiving messages and process those messages respectively.

Communication Module:

The communication between client and server will be handled by the communication module. This communication can be implemented using Windows Communication Foundation (WCF). Here, the communication channel will be created through which the request from client, sends to the server and receives the response from the server.

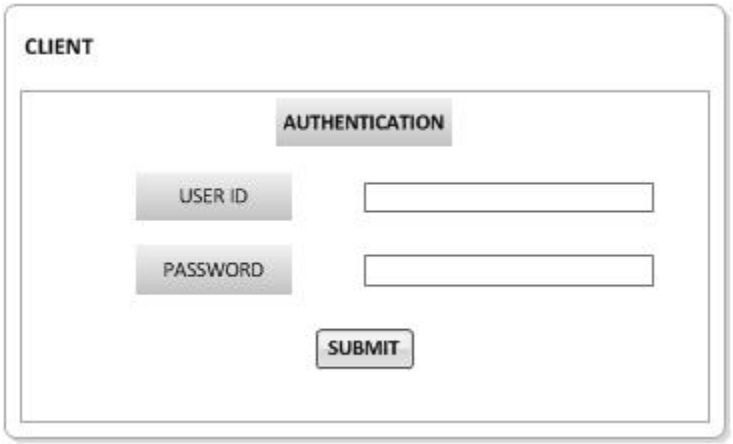
Display Component:

Display component is one of the important parts of the client, which displays the output of the user requests. It displays the type dependencies among the file set or package dependencies according to the user requests. It displays the test messages and result logs from the test harness server. In case of errors in testing, the test-harness server sends the error notification to the client and it is displayed in this component.

5.4 Views

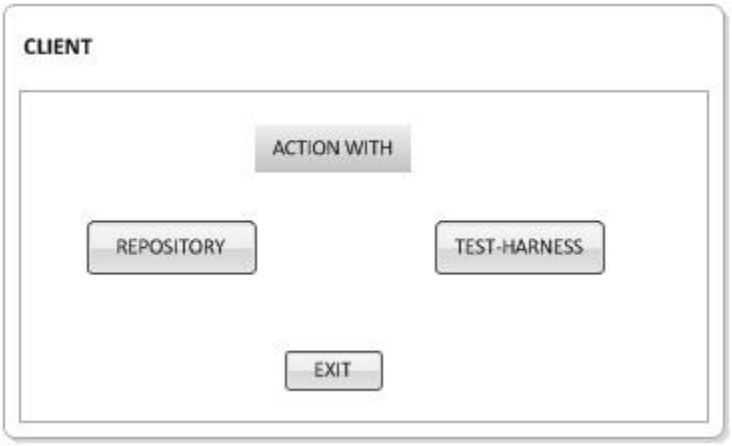
Continuous build and integration system has client GUI. This section deals with some of the major views associated with the GUI.

1. Login:
   Every user who accesses the repository should be authenticated. This is done in two steps:
   First the user id and password of the developers are verified with company data base and if valid then employee id is verified with members in project server. If authentication fails in first step authentication failed message is displayed to user. If verification fails in second step then it displays about the access to wrong project. The user then can get access by contacting his team lead who contacts system administrator.

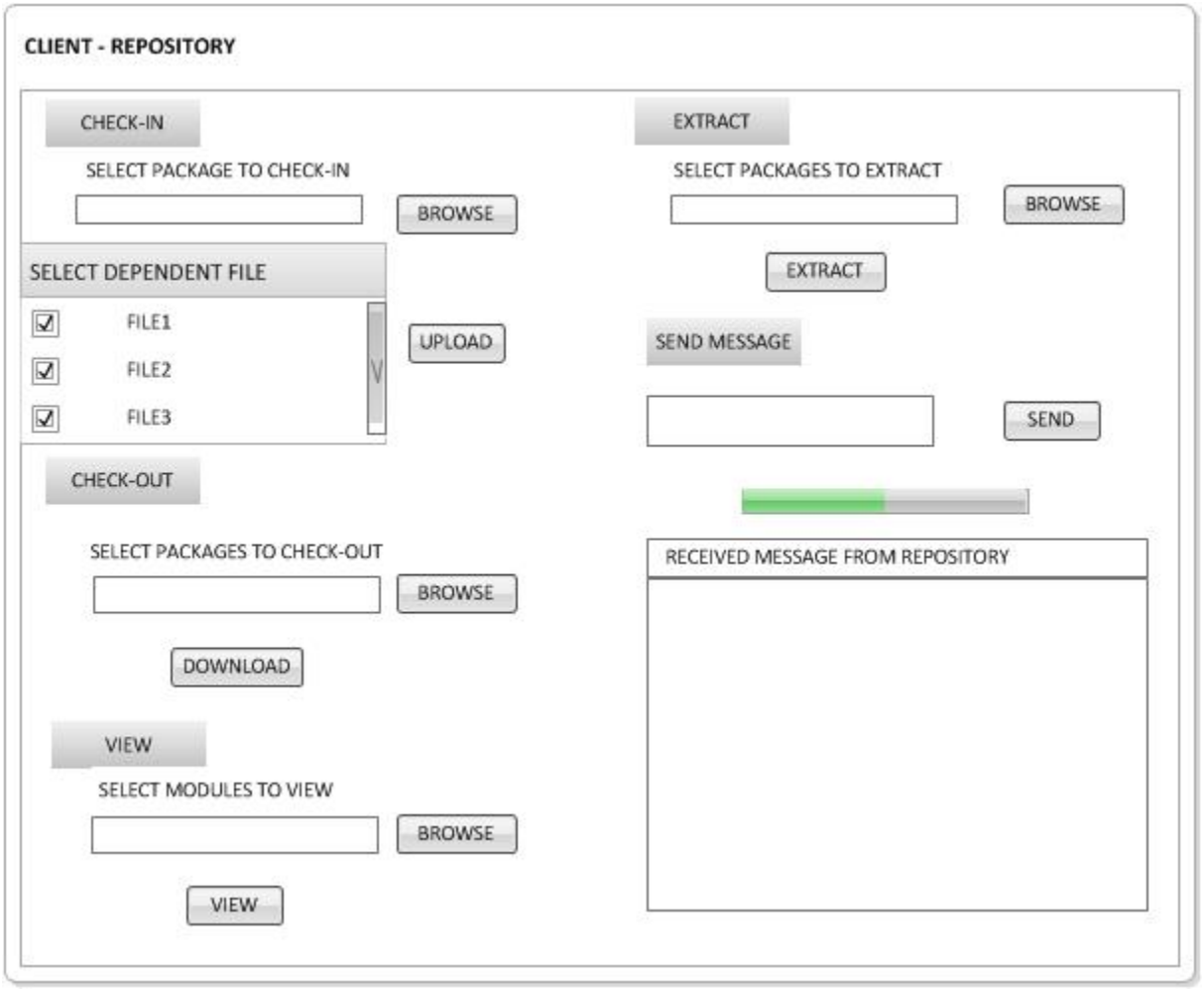


1. Home page view:
   After the user successfully logged in, he/she gets the home page view as shown below. The home page has repository button and test-harness button. By clicking the repository button, the user will be provided with all the actions/tasks that can be performed and communicates with the repository. After this button, the communication channel between the client and repository is created and all the

messages or requests given in the user interface are directed to the repository server. By clicking the test-harness button, the user will be provided with all the actions that can be performed with test-harness server. This will direct the requests given by the user to the test-harness server.
The exit button exits from the user interface.



2. Repository client view
   The client interface has the check in option where a user would browse the modules to which they would want to check in or can choose to check in to default repository which would not be tagged to any project. Once the module is selected the user can select the packages which would be needed to check in. The user can click on view option by selecting the project name to view the list of packages available in the project and their versions. User can check out based on versions of the document by using the check-out option. The user can choose which packages they needs to check out by clicking on browse button. The send message option provides an option to send XML message to repository server. The received message from the repository server is displayed in the GUI.



3. Test-Harness Client view:
   When the user clicks the tests, it will display all the testable dlls as shown in the diagram. Each test dll has a checkbox. If it's checked, then the Test Harness would run and test and the output the test results in the right side. Otherwise, Test Harness just ignores this test dll. The user can send messages to Test-harness server by using this send button. They can also download the test messages and result logs by clicking the corresponding buttons and the output will be displayed in the right side.

5.5 Functionalities

The important functionalities of the client are described below

1. **Check-in the source codes**

   The client will upload the newly created code to the repository server to add to the currently working software baseline. This process is called Check-In. The check-in will also be done after making any changes or updating the existing code.

2. **Check-out the source code form the repository server**

   The client can download the specific packages or modules from the repository server either to view or to update the source code.

3. **Send XML messages to the repository server**

   It creates and sends the XML messages to the repository server as query. The repository server execute the received messages from the client and return the corresponding output to the client.

4. **Run dependency analysis**

   The client runs the dependency analysis into the repository server, to get the dependencies between the packages and/or modules. The repository server stores the code based on dependency.

5. **Run test-harness**

   It runs the test-harness to perform testing on the newly added code or any existing code. The test-harness returns the error message in case of any errors, otherwise, it returns the output of the testing to the client

6. **Send message to the test harness server**

   The client will be able to send messages to the test harness to do testing, or to get any test messages or message log.

7. **Receive and displays the output messages from the repository server**

   As a result of sending requests to the repository server to perform any tasks, the repository server returns the output of the corresponding request to the client. In turn the client receives it and displays it in the graphical user interface.

8. **Download the test messages from the test-harness server**

   Client will be able to download the test messages of the corresponding tests specified by the client from the test-harness server.

9. **Download the result logs from the test-harness server**

   It also able to download the result logs from the test-harness server as on demand from the client.

10. **Receive and displays the error notifications from the test harness server, in case of errors in testing**

    In case of any errors during testing, the test-harness server will send the error notifications to the client, to notify about the errors.

11. **Receive error notification from the build server, in case of any build errors**

    If there are any build errors like compile errors or any linking errors, the build server sends the error notification to the client.

5.6 Critical issues

Issue 1: What if there are lots of files matching to the given query, even after providing categories?

Solution: One of the possible solutions is to display the categories and the number of matches from those categories. This will reduce the output to a large extent. If the client needs to view the matches from a category, they can obtain the files by clicking on the specified category.

6 Repository Server

6.1 Introduction

The main purpose of the repository server is to hold all source code files, their metadata files. It acts as the central server for storing all the source code files and can be accessed by all other server and by all clients depends on the protection policies. It gets the source code from the client. All the developed code, test logs, and metadata files related to the currently working software baseline are resides on this server. The architecture of the repository server follows the client-server model, where server is responsible for processing requests from the client such as check-in, checkout, download, searching of text or some metadata information in repository files and versioning of the files. These tools are run when a request is received from the client. Server communicates with the client and route the requests from the client to the appropriate tool, if needed. It also provides analysis tools such as dependency analyzer to analyze the modules and subsystems present in the repository and find the relationships between them. To maintain the integrity of modules, they are accessed through interface and object factory. Some of the important components in the repository server are executive, parser, file manager, versioning manager, check-in, check-out. It also has to execute queries to get dependency relationships. One of the important functionality in the repository server is to manage the check-in and check-out of packages source code to and from the modules. To perform check-in and check-out functions, appropriate check-in/check-out policies are to be framed. For metadata search or update, the metadata tag values should be provided. In repository server several tools are present which performs various functions. Some of them are listed as follows:

- Dependency analyzer
- Versioning Handler
- Metadata generator/updater
- XML writer
- Query handler

6.2 Users and Use cases

6.2.1 Primary Users

Dependency analyzer can be used by various users. In this section, we discuss about the primary users and the features provided by this system for its users. The possible users of this system are as follows:

- Developers
- Quality assurance team
- Project/team manager
- Administrators

1. Developers:

   Developers work with large projects which has hundreds of codes developed by others and also the code they develop may have complex interdependencies with other packages. They uses repository server to check in code and they can extract or view the code files for reference. They check out code files to update the code with their changes. They get the test results as soon as tests are performed on code they checked-in.

2. Quality assurance team:

   Primary work of the quality assurance team is to ensure that the system is properly designed to meet all the requirements. They will run the tests on the files which are stored on the repository. Quality analysts check the errors in the system and make sure that the system should not have any bugs or inconsistency while it reaches the customers.

3. Project/Team manager:

   Project Manager can use this repository server to get all the developed packages or modules with currently working software baseline. It also can be used to get the file information and dependencies to keep track of the progress. They can analyze the some specified projects in specified server. They can manage the whole system by modularizing it.

4. Administrators:

   Administrators can use this repository server to manage all the developed code in single place. They can find the dependencies between types and/or packages and can help them to manage the overall system design.

6.2.2 Use cases

The general use cases of the repository server are as follows:

1. Stores the all the source code for the currently working project baseline.
   The main use of the repository server is to store the all the packages and modules that are developed by the developers working on the same project
2. Allow the developers to check-in and check-out the source code
   The repository server helps the developers to view the code files for reference and to check-out the required code files to update the code files. It allows the user to check-in the updated code and newly developed codes, so that the code will be added to the currently working software baseline.
3. Used to find the dependency relationship between the source code
   This repository server performs the dependency analysis and stored based on dependency. So it is helps the users to easily understand the dependencies between the packages and modules.

6.3 Partitions

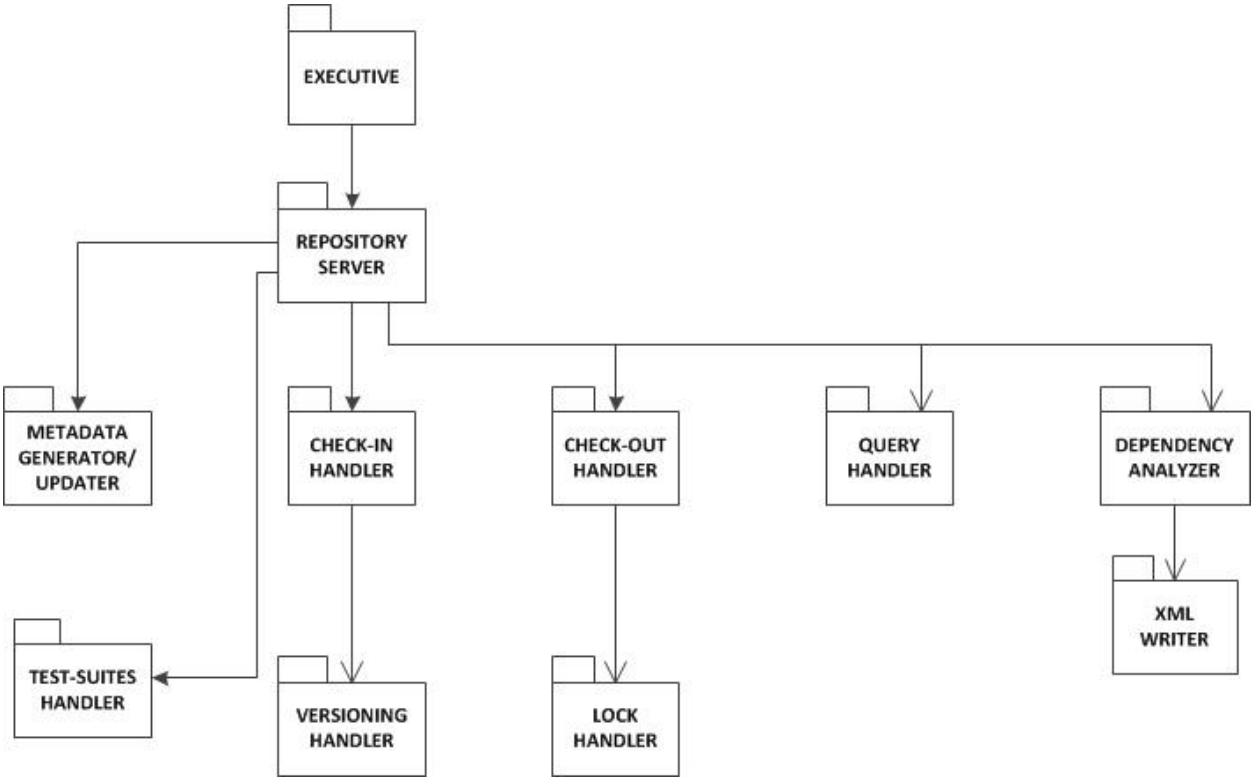


Fig 6.1: Package Diagram – Repository server

Executive:

It is the entry point of execution of the repository server. It consists of main method of the server. This executive package takes control of the whole server side module. It handles all the requests from the client to the repository server and directs them to the corresponding handler.

Repository server:

This module acts as a central processor which receives the input from the server executive and processes it to the respective handler. It defines the modules as lists of packages and subsystems as list of modules. The repository server maintains the integrity of modules accesses through the interface and object factory. It creates dll file for the source code (.cs file). So that we don't need to recompile the code again. The results or output from the handlers are returned to the executive.

Check-in handler:

When the client sends the check-in request and uploaded the source code file, the check-in handler creates the dll file and the uploaded file is stored into the repository according to ownership policy. The metadata file for the uploaded file is built using metadata generator/updater. The owner of the original version needs to update the present file to currently using file, in the case of the file is checked-in after the check-out. Whenever a user tries to check-in a modified file, the version number of the file is increased and processed to the versioning handler and a new metadata associated with this file is created. Older version of the same file is retained in the repository.

Versioning handler:

When the source code file which is already present in the repository is updated by the owner, the version number of the file is increased and check-in the modified file. When the dependency of that file is changed, then the version change detector checks the both old and new version and copies the filenames to avoid duplicates. A notification will be send to all other team members indicating the update of the old file. The older version of the file is retained in the repository mainly because that the developers can access the older configurations for products that are still in service and the configuration could be easily rolled back should an earlier change proved to be incorrect or lead some problems in the developing system. The metadata file associated with this source code is updated by metadata generator/updater.

Check-out handler:

When the user sends the request to check-out the particular file, the check-out handler gets the fully qualified path of the file and checks for the locks on that file. The Checkout handler send the requested file to the client. The server maintains information of the

users who have checked out files from the repository and this will be useful in the case when the client checks-in the modified version of this file. There are some lock policies in the repository which places the lock on the file which is checking out, so that other clients will not be able to check-out that file until the lock is released. This lock policy prevents the concurrent access of the files. When transferring large documents, file chunking can be used. The lock policies and lock of the files are handled by the lock handler.

Lock Handler:

The lock handler place the lock on the file which is checked-out file based on the lock policies. When the lock is placed on the file, no other clients can access the file which helps to avoid concurrent access of the file. The lock is placed in the file until the file is checked-in back.

Query Handler:

When the client sends the XML messages, the query handler executes the queries which are about the dependency relationships. The query handler executes the query on to the XML writer which contains the dependency relationships in XML format and get the result and return backs to the client. If the query handler gets the input consisting of strings to be searched for, project name, search type, text query searches are executed concurrently across file. It prepares the output consisting of files matching the query string and search type. If it gets input consisting of metadata tags and projects to look at, search is performed concurrently on all the files in project. The processor uses XML reader to read the xml files. The metadata element and their values are fetched. The file references and project name of metadata file are also stored in output.

Dependency Analyzer:

The purpose of Dependency analyzer is to analyze the packages that reside on repository and display the dependencies between the types and/or packages in the file set. It is responsible for finding all types in the file set, finding dependencies between types in the file set, finding dependencies between the packages. The dependency relationships between the modules and packages are stored in the XML writer, in XML format. When the client send request to get the dependencies of the specified project, the dependency analyzer analyze the projects and returns the dependency relationships of the specified package with others, to the client. The dependency analyzer tool consists of several packages like analyzer, parser, tokenizer, semi, File manager.

XML Writer:

XML writer stores the dependency relationships among the modules or packages present in the repository. XML generator module writes the output information of the dependency analyzer into the XML format. When the client send XML messages or send request to query the XML, query handler executes the query on the files reside in the XML writer.

Metadata Generator/Updater:

This module is used to generate the metadata file for the packages present in the repository and update the meta-data file when the user wants to update the meta-data file. Meta-data generator/updater receives the tag names from the clients and checks the files for the specified tags. If it is already there, then the new value will replace the old value. Otherwise, new tag will be created and stored into the repository. Metadata generator generates the metadata for the newly checked-in file and stores the metadata of that file in it. After successful build by the build server, the check in handler calls metadata generator/updater by providing with tag names and values provided by user. The tags include dependency, version value.

Test-suite Handler:

A Test-suite is the collection of test case for the particular module. These test cases are the XML files for the .cs files which are present in the repository. The test-suites based on the XML messages received from the client. The main purpose of the test-suite is to test a developed code to ensure whether it is working as per the specified behaviors of the code. The test-suite handler holds all the test cases to the test the packages reside in the repository. When it receives the request from the client or from the test-harness to test the specified code, it will send the sequence of test cases of the specified source code to the test-harness server to run the tests.

6.4 Activities

The activity diagram shown below represents the sequential order of activities performed. This diagram depicts the overall flow of control.

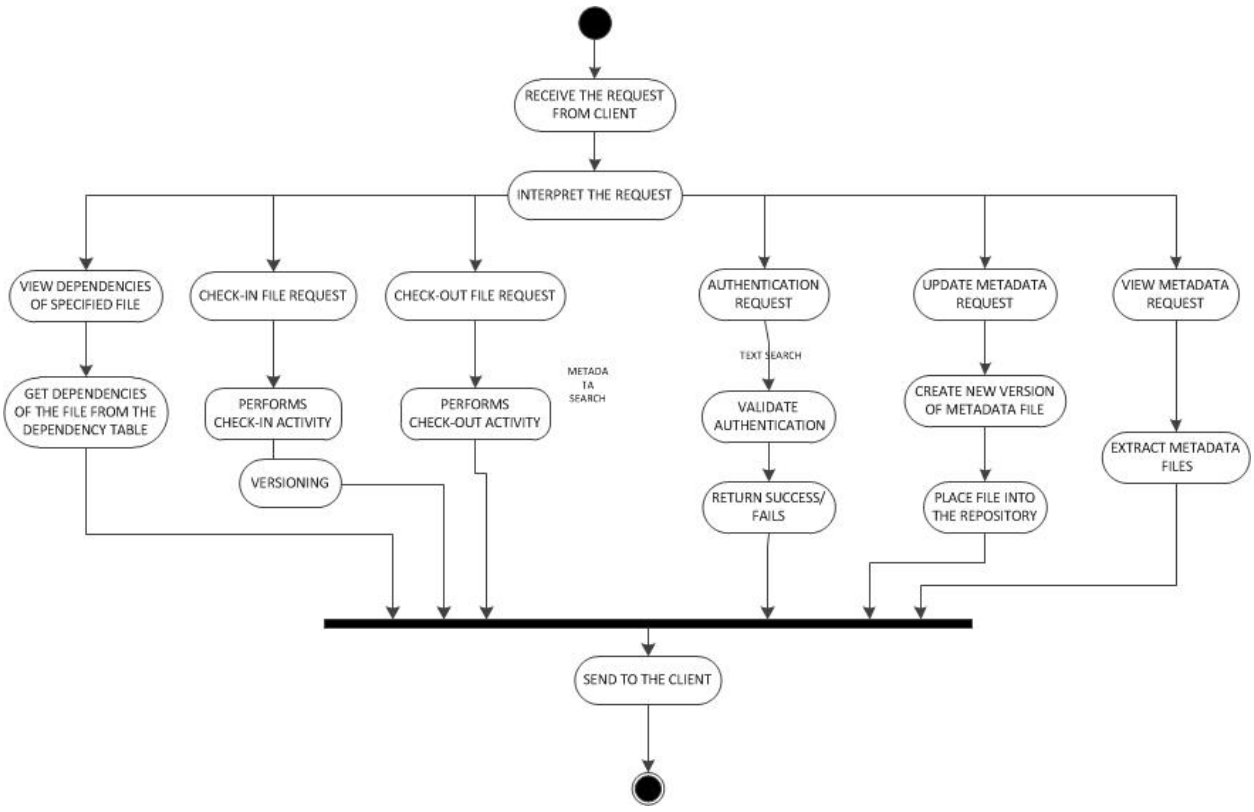


Fig 6.2: Activity Diagram – Repository server

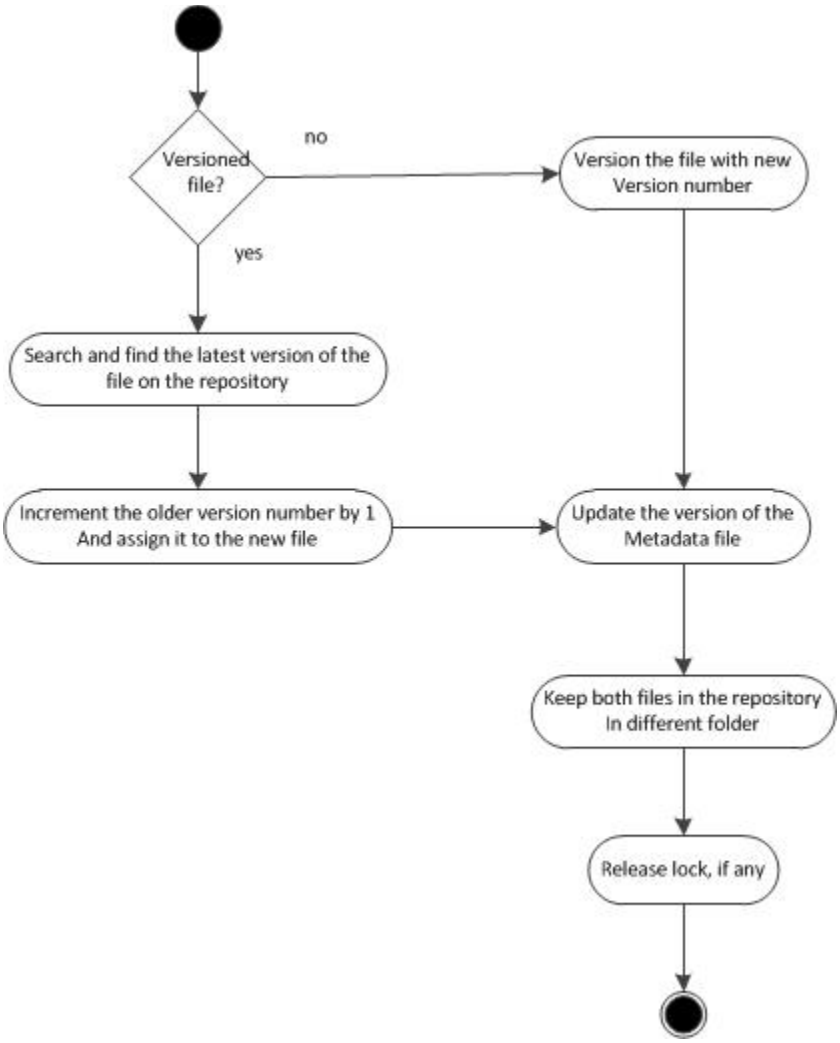


Fig 6.3: Check-In Activity

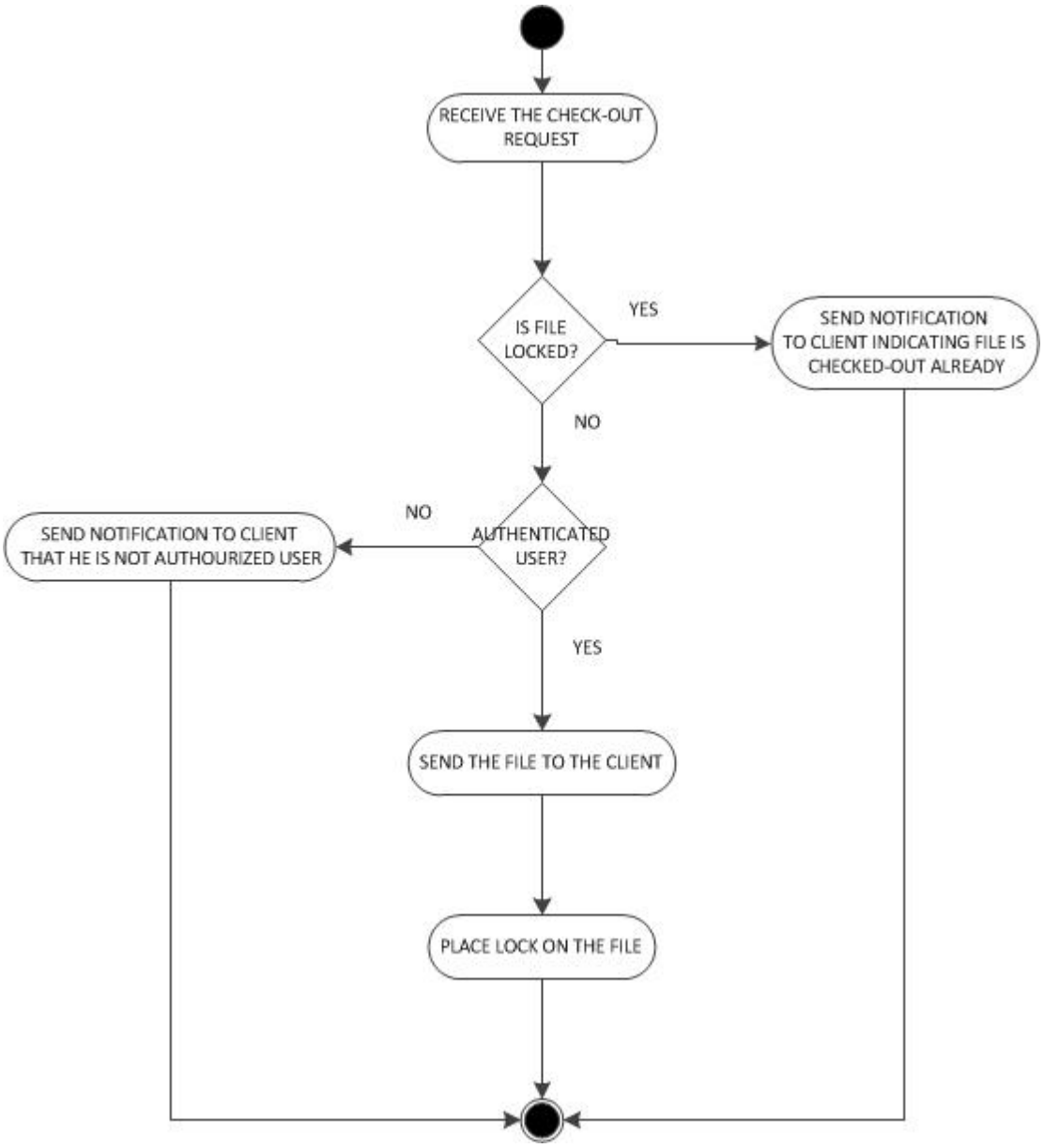


Fig 6.4: Check-out Activity

Description:

The Top level tasks that are carried out on the repository server as follows

1. Listen to any connection requests from the clients.
2. Receive the requests from the client.
3. After receiving the request from the client, interpret the request to call the appropriate tool based on the request.
4. If the user sends an authentication request, validate the authentication by matching the user id and password given by the user with those present in the hash table maintained for the user ids and passwords in the repository server. If the entries are matched a success status is sent to the client otherwise a failure message.
5. If it's a view dependency request, the server fetches the desired file from the repository, and also it's dependencies which are stored in the dependency table in the dependency analyzer module. Then it transfers the dependencies of the specified file to the client using message passing mechanism.
6. If the request is to check- in the file to the repository, versioning needs to be handled and also its associated metadata needs to be created with appropriate version of the file. It first checks if the file is the newly developed one or the updated file of the older version. If the file is un-versioned then a new version is assigned to the file, whereas if it is versioned then the latest version of the file is tracked in the repository. Then this version number is incremented by 1 and assigned to the file. Similarly the version of the metadata file is also updated. After the files are copied to the repository the lock is released on the file which was acquired during the checkout process.
7. If the request is to check-out the file, the first thing that is checked is the ownership. A client is allowed to checkout a file if and only if he or his group is\are owner of the file. If he is not the owner of the file, he is provided with the download option. After the checkout is successful a lock is assigned on the file which blocks other user to check-out the same file. Thus at a given instance of time, no two users have access to the same file which eliminates the need for merging and also the concurrent access.
8. If the request is to update the metadata, the metadata file is not actually edited but rather a new metadata file is created using the old version and inputs acquired from the client and appropriate version is assigned to the file.

6.5 Critical issues

Issue 1: Ownership configuration

This issue is related to the storage of ownership of files. A file may have single owner policy or a group policy, how this can be stored and retrieved efficiently?

Solution: The information about the owners of the file can be stored in the metadata file. The XML of both the policies looks different.

For Single owner policy, the XML looks like-

```
<owners>
    <owner>Developer1</owner>
</owners>
```

For Group policy, the would look like-

```
<owners>
    <owner>Developer1</owner>
    <owner>Developer2</owner>
    <owner>Developer3</owner>
    <owner>Developer4</owner>
</owners>
```

Issue 2: Same file names to different files

This is an issue of how we can handle the case, if two clients attempt to check-in different files with same name

Solution: The solution to this issue would be using the owner name as prefix for all documents, e.g., Dd_QueryProcessor.cs.

Issue 3: Security Issues

Since there is a lot of data is transferred to and from the client and the server, it makes really important to make sure the data and the communication is properly transmitted.

Solution: By considering the security issues, the data is being sent to the communication channel is encrypted and sent to the other package. Then the receiver can decrypt the message and can use it. We could use any mechanism for encryption and decryption.

Issue 4: File Transfer with chunking

Since Source code repository is a Client server application there are chances of connection interruption between them during file uploading/downloading from/to the server which might result in loss of data.

Solution: This can be handled by maintaining a checksum at the client and server and verify the checksum for any loss of data after each transaction.

Issue 5: Performance when large number of files in the repository

This is an issue in the case when the large number of files will be searched from the Repository Server due to the selection of multiple categories. This may also take more time to search.

Solution: One of the solutions to this issue, is to show the matches for the selected categories as soon as they are found. A progress bar can also be shown saying that the query is in process. The query can also be run in parallel. Here, the search files can be searched in parallel as each file search is independent of the other file searches.

7 Repository Structure, Policies and Other Functionalities

7.1 Repository Structure

The repository stores all projects of the developing baseline. It defines the modules as lists of packages and subsystems as collection of modules. It stores every project in separate directories. The directories stores all the files related to the project such as source code files, metadata files, test cases and the test case results. The storage in the repository server would looks like:

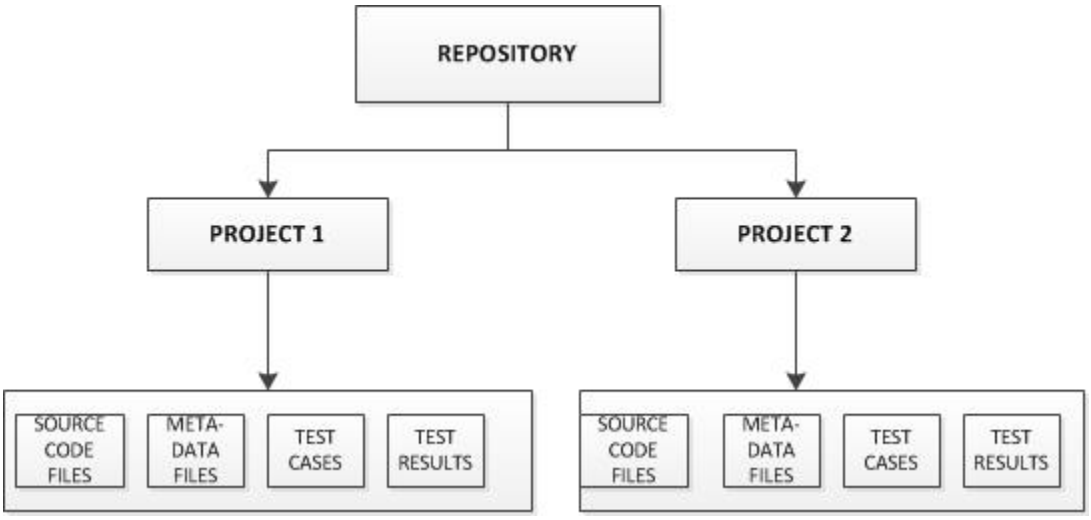


Fig 7.1: Repository Structure

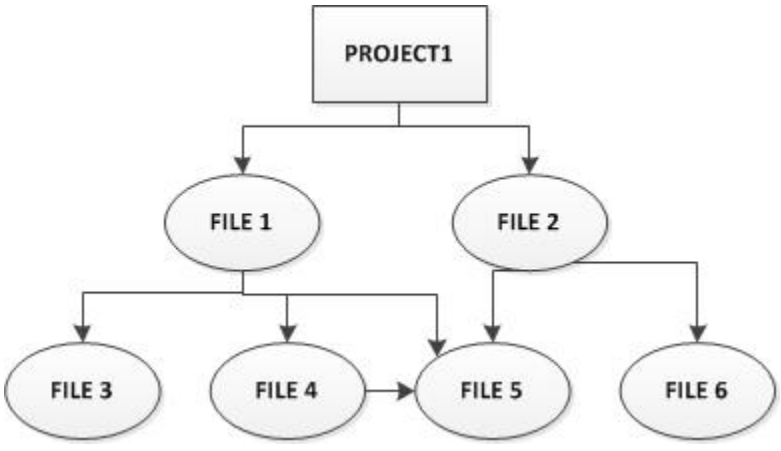


Fig 7.2: Dependency based storage of code files

A new directory is created for every new project, in the repository. The main project directory consists of 4 more directories. They are

1. Source code files - This directory contains all the source code files of the project
2. Metadata files – This directory contains all the metadata files
3. Test cases – This directory consists of the test cases saved by the tester.
4. Test Results – This directory stores all the test results which are sent by the test harness server and can be accessed by the clients.

The source code files under the project directory are stored based on the dependency as shown in the above fig 5.2

7.2 Repository Policies

The architecture of the repository server is based on some of the policies defined below:

1. Check-In policy
2. Check-Out policy
3. Ownership policy

7.2.1 Check-in Policy

The check in policy used in the architecture of the repository server is as follows

- When the source code file is check-in after checked-out that old file and update it or making changes to the code file, then the new version of the code file which is currently checking-in is stored in different folder. Both the old version and new version are resides in the repository. Similarly, for metadata files, the updated version of the file is placed in different folder.
- Every source code files the owner tag. The Owner's id is placed in the owner tag of their respective source code files.

7.2.2 Check-Out policy

The check-out policy used in the architecture of the repository server is as follows

- While checking-out the source code file, no more than one user can check-out that at the same time.
- When the file is checked-out for making changes or updating, a lock is placed in the checked-out file, so that no one can access that code file. The lock is applied on all write operations of the checked-out file but it can read.
- The check-out file should be followed by check-in the code file after updating. Otherwise, notification will be send.

7.2.3 Ownership policy

An ownership policy is mainly to define who is allowed to access the code files into the repository. The repository can have various ownership policies for the files in the repository.

1. Single owner policy – This is the most basic ownership policy. In this policy, only one user is allowed to check-in and check-out the file into the repository. This ownership policy is implemented by adding the owner tag in the metadata of the file.
   For Single owner policy, the XML looks like-
   <owners>
   <owner>Developer1</owner>
   </owners>
2. Group owner policy – In this policy, more than one users are owners of this file. So all the users who have authorization can access the files in the repository. All the owners are added into the metadata file. Once the file is checked-out, a lock is placed on the file, so that no other user can check-out that file. The lock will not be released until the file checked-in back.
   For Group policy, the would look like-
   <owners>
   <owner>Developer1</owner>
   <owner>Developer2</owner>
   </owners>

The architecture of the repository server in the CBIS system follows the group ownership policy with lock sub policy. This policy allows group of owners for the particular file. Only the owners of the file have access to that file like check-out, view, modify, etc. If anyone needs to check-out any file that resides in the repository server, then they must be one of the members of the group which owns this file. The administrator and project managers are responsible for this membership of the group and they provide grant access to the file to the group members. If the user is granted access to the file then his name is entered in a value field of the hash-table which contains the group-ids as the key field and the users in the value field. Now as soon as the checkout request is received by the server the user id is matched in this hash-table. If the match is found then the user is granted to check-out the file. After the access to the file is granted to the user, the lock is placed on the file in the repository server and the lock will not be released until the file is checks-in back to the repository server. As long as the lock is on the file, no other group members access or check-out that code file. This lock prevents concurrent access of the file by different users. This policy helps to eliminate the overhead by merging of files because of the concurrent access to the same file by different members of the group.

7.3 Functionalities

7.3.1 Versioning

A policy needs to be maintained regarding the versioning of the files in the repository server. The versioning policy used in this repository server is as follows:

Whenever the repository server receives the check-in request from the client, the versioning should be handled. The repository server first needs to identify whether the check-in file is newly developed code file (un-versioned) or updated code file of the checked-out file (versioned). If the check-in code file is adding newly to the current project baseline, then the new version number is assigned to the file. Otherwise, the old version number is incremented by 1 and assigned to the version number of the newly updated code file. Similarly for metadata file, the same policy is used for versioning. Both the old and new versions will be stored in the repository in different folders. The old versions can be used for roll back or backward compatibility purposes. If the old versions have dependencies, the files which are dependent on the old versions are still have dependency with the older version.
So we have to notification indicating the availability of the newer version of the dependent files in the repository. Then they change their dependency to the newer version of the code files.

7.3.2 Notification

The notification part in the repository server ensures that the owners of the project get the notification if the new version of the file in the project is available. By sending the notification about the change, one can update the file. When a file is updated in the repository, the repository server sends a message to the users who are responsible for the project and notifies them about the updated file.

7.3.3 User Authentication

This policy refers to the login that the user performs when the project starts. The user is provided with unique user id and password which must be authenticated to use the files in the repository.

7.3.4 User Authorization

This policy allows only the authorized users to access the files in the repository. For instance, when the user wants to check-out a file, the repository server checks whether the user is an owner of the file. Only if it is yes, then the file is checked-out to the user, otherwise, they intended with error message.

7.3.5 Caching

A Code Repository client could implement caching by storing locally files it had received from the Repository at some earlier time. When it wants to download a component that depends on other files it requests the Repository to send a list of all the files on the component's dependency graph. It then downloads all the files on the list that it does not currently have in its cache, and adds them to the cache as well as uses them for its current purposes.
The repository builds the file list by traversing the virtual dependency graph formed by the component's metadata file and all its descendant metadata.
If the client modifies a file and checks-in a new version the client also places the modified file in its cache.
Note that this process could be part of a lazy loading mechanism to allow the client to navigate through local metadata, making a file request whenever it is unable to satisfy a reference found in one of its metadata files. This would save a lot of network traffic and server load and make client's navigation view more responsive.
One issue is how to recognize and resolve cache incoherencies. Note that it may not be terribly important that a client's cache doesn't have versions that were added to the Repository today. The client could, at midnight, request the server to send all metadata in the categories of interest to the clients that have been added since the last update 24 hours ago. You probably want the client to have the ability to request metadata added to the repository since the client's last request. That is expensive enough that a client would only use when coherency is important.

Another issue is how to manage the cache contents. We probably do not want the cache to grow to become most of the Repository contents. The good way of doing it is identifying client project and discarding details of other projects. Even with in a project if file count is large then cache can be allocated using "Most frequently used" algorithms. This file present in local repository can also be extracted but check out cannot be performed.

Reference: Midterm solution Fall'13 by Dr. Jim Fawcett

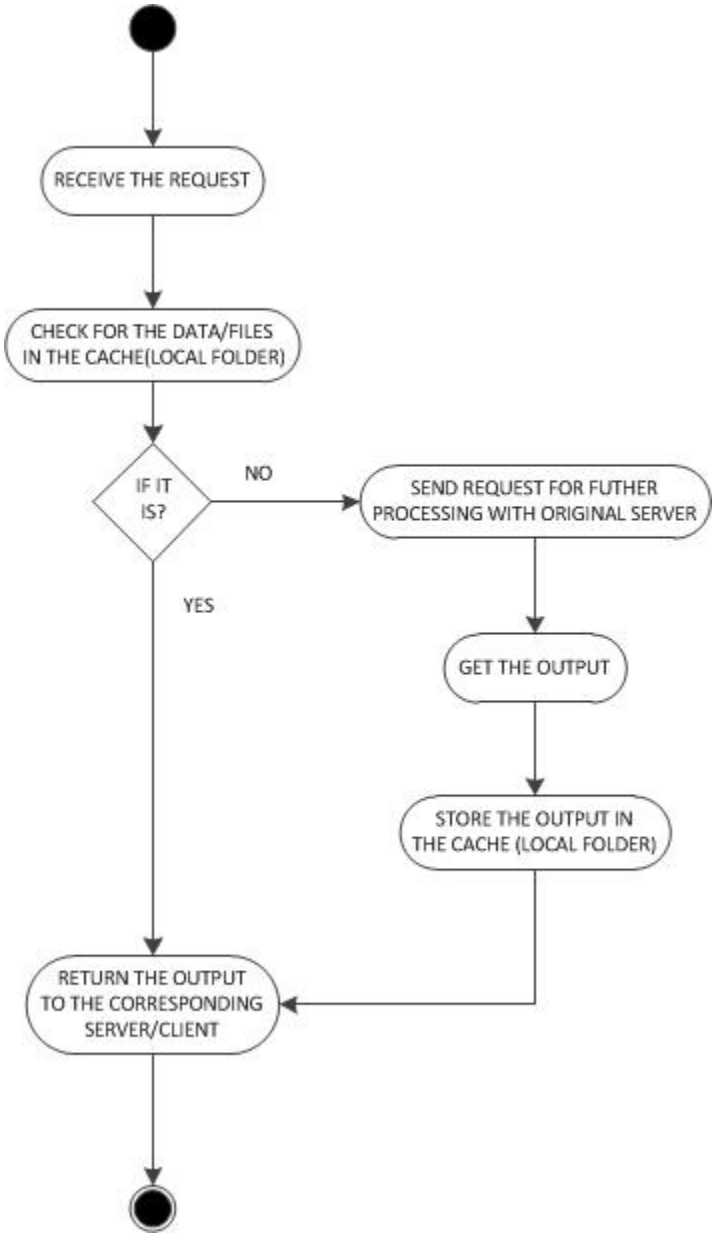


Fig 5.3: Activity Diagram – Caching

8 Dependency Analyzer Tool

8.1 Introduction

The purpose of Dependency analyzer is to analyze the packages that reside on the repository and display the dependencies between the types and/or packages in the file set. This system will be very useful to find the relationships/dependencies among the files in remote machines. It provides a clear view of the dependencies and also helps the user to get a clear picture of the source code.

The main users of the dependency analyzer are

1. software developers
2. quality analysts
3. project manager
4. Administrators
5. customers

8.2 Users and Use Cases

8.2.1 Primary Users

Dependency analyzer can be used by various users. In this section, we discuss about the primary users and the features provided by this system for its users. The possible users of this system are as follows:

1. Software developers
2. Software architects
3. Quality analysts
4. Project manager
5. Administrators
6. Customers

Software Developers:

Software Developers work with large set of files. This dependency analyzer tool will be very useful for them to generate metadata for the complete file set. This tool will provide the dependencies among packages/files which will help the developers to identify the dependent files and they separate and modularize those files and work on it which makes them easy to work on each functional unit separately. This tool will also help them to get a clear view of the software design.

Quality Analysts:

Primary work of the quality assurance team is to ensure that the system is properly designed to meet all the requirements. Quality analysts check the errors in the system and make sure that the system should not have any bugs or inconsistency while it reaches the customers.

Project Manager:

Project Manager can use this tool to get all available projects. It also can be used to get the file information and dependencies to keep track of the progress. They can analyze the some specified projects in specified server. They can manage the whole system by modularizing it.

Administrators:

Administrators can use this dependency analyzer to find the dependencies between types and/or packages and can help them to manage the overall system design.

Customers:

Customer will use this tool to find all available projects in particular server (remote machine) and find the dependencies of the complete file set in the specified project. They can view the required dependency alone either type dependency or package dependency or all the dependencies in the file set.

8.2.2 Use Cases

The general use cases of the dependency analyzer system are as follows:

1. Performs type dependency analysis

By sending the request to analyze the project reside in the specified server, the server analyze and finds out dependencies of all the types and sends back the output to the client.

2. Performs package dependency analysis

By sending the request to analyze the project reside in the specified server, the server analyze and finds out dependencies between all the packages and sends back the output to the client

8.3 Partitions

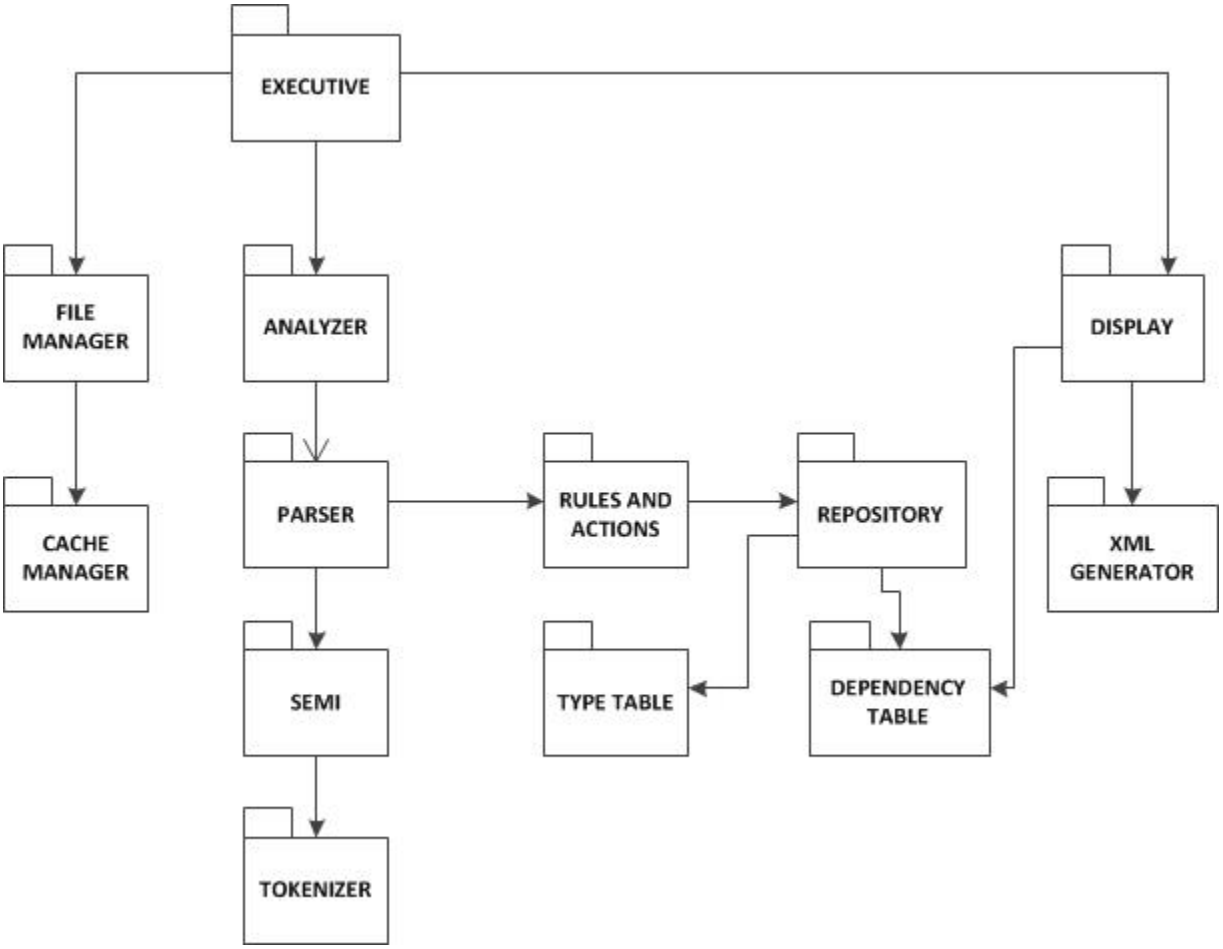


Fig 8.1: Package diagram – Dependency analyzer tool

Executive:

It is the entry point of execution of the dependency analyzer. It consists of main method of this tool. This executive package takes control of the whole module.

File Manager:

The main responsibility of the file manager module is to traverse the repository to get all the file references and stores the full path in the path list which will be needed to be analyzed by the analyzer. It will interact with Executive package, cache manager package and also with repository.

Analyzer:

Analyzer module handles all the analysis which needs to be done. It simply plays a role of controller for the analysis logic. It is responsible for coordinating all the required processing. It gets the list of file names as input and processes each file by parsing them with the parser and other modules like semi, tokenizer. As result of processing it finds all types and its members and the results are stored for future. In order to find the dependencies between the types and/or packages, all the files are need to be parsed for second time by using the result of the first parse. It interacts with parser, file manager, display modules.

Parser:

Parser module is responsible for parsing and to get all the types and several other information and store it in the repository which helps to display them easily. This package finds all the types, their members, and relationship/dependencies with other types. It is very important package to meet the requirements of the project dependency analyzer. This package interacts with semi package and receives semi expression and parses them. The parser sends the output to repository and to display.

Semi:

Semi package is responsible to receive tokens in the file and make semi expressions from the tokens. This package doesn't deal directly with the file. It receives the full path of the file from the parser and sends it to the tokenizer and sends the semi expression back to the parser for further parse.

Tokenizer:

Tokenizer package is responsible for splitting the characters into tokens from the given stream of file. It receives the full path from the semi package and send back the tokens into the semi package.

Display:

It displays the output information by setting the write output stream. This package interacts with XML generator to convert the Linq queries into the XML format. It also interacts with type table to display only all the types in the given file set, and dependency display to show all the dependencies.

Type table:

Type table module is responsible for storing all the information about all the types like class, struct, enum, interface in that server. It displays whenever needed.

Dependency Table:

This module stores all the dependencies between the types and all the dependencies between the packages and display as per the user requests.

XML generator:

XML generator module writes the output information into the XML format. XML generator will provide the interfaces to be wrapped into a output stream so that Display package can wrap it.

8.4 Activities

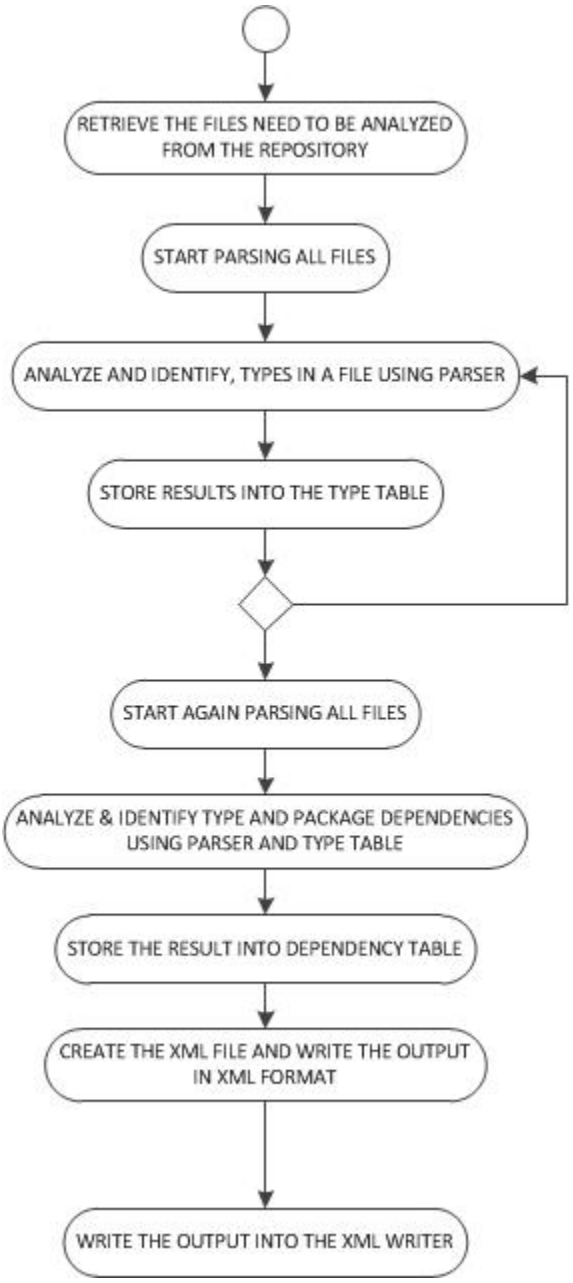


Fig 8.2: Activity diagram – Dependency Analyzer tool

The activity diagram show above represents the sequential order of activities performed. This diagram depicts the overall flow of control.

Dependency analyzer deals with identifying the dependencies between the types and dependencies between the packages in the distributed system. For this process, each and every file needs to be parsed line by line and analyzed. The major tasks involved in this application are as follows:

- **Retrieve the files**
  Once it receives the request, it starts retrieving the files to analyze from the repository. The tool parses the input parameters to be passed. This helps to set context of the application in the executive package in the server side.

- **Analyze each files for types**
  For analyzing the file, the parser package parse each input file in the list line by line. Identify all the types defined in a file and stores them in the repository. Types and its members in file are identified by using some predefined rules in the rules and actions package. All types are stored in type table as well.

- **Analyze each file again for finding dependencies**
  Each input file is parsed again to identify the type dependencies and package dependencies. The type information is stored in the type table during the first parse is used here. The dependency table stores the dependencies information found.

- **Create file and store the output in XML format**
  The final step after analysis processing is to send the results to the client. It writes the output to a file in XML format. Then the XML file is send to XML writer

9 Test-Harness server

9.1 Introduction

The main purpose of the test-harness server is to support the testing throughout the development process of the large projects. It helps the developer to ensure that the functionality of the code they developed meet the requirements, by testing. It allows them to test the code before and after integrating into the project baseline. As each new source code or module is developed, the test classes for that code is also developed simultaneously and incorporated into the test suite. A test-suite is collection of test classes for a particular module. Each test class should generate and read all the data necessary for testing and provides result logs. When run the tests, it executes all the tests in the test-suites and return the test messages which contains information like the number of tests run, errors, number tests passed, etc.

Test-harness server holds the test-suites for each modules and subsystem that reside on the repository server. The tests are executed, only when it receives request from the client or repository server. When it receives the XML messages, it loads the test-suites defined by the XML message. Once it gets the all the needed data to run the tests, it starts executing and logged the test results. In case of errors, it sends the error notifications to the client. Otherwise, it will send the success notification as test message and results log to the client. The testing should be done after check-in the newly developed code as well as check-in the code which is updated. So that it ensures that functionality still meets requirements.

9.2 Users and Use cases

9.2.1 Primary users

Test-harness server can be used by various users. In this section, we discuss about the primary users and the features provided by this module for its users. The possible users of this system are as follows:

1. Developers
2. Quality assurance team
3. Project manager
4. Testing team

Developers:

Developers use this test-harness server to test the code developed by them in order to ensure that source code functionality is working properly. According to the test results, they can modify or update the code. After check-in the code, they has to compile and

execute the corresponding test cases for that module for testing the particular module. This also has to be done after updating or modifying the checked-out code files.

Quality assurance team:

Primary work of the quality assurance team is to ensure that the system is properly designed to meet all the requirements. Quality analysts check the errors in the system and make sure that the system should not have any bugs or inconsistency while it reaches the customers. The Quality assurance team execute the test cases of the module or sub system to check errors and ensure that the quality of the product is maintained. They also formally witnesses that the Tester correctly completes all the Test Cases.

Project manager / Team leader:

The project manager is responsible for monitoring the progress of the project as on agreed plans. They use this test-harness server to keep track of all the tests run by his team and make sure that each source code files in the current project meets its requirements by looking at the outcome of the tests run by the Test Harness server.

Testing team/tester:

The primary user of this test-harness server is testers/testing team. They run the test cases for each module or package. They keep track of various test classes, test run and they also maintain the error messages, results log, number of tests run and number of tests passed or failed.

9.2.2 Use cases

The general use cases of the test harness server are as follows:

1. Run the tests
   Test harness server runs all the selected tests by the client or repository server, by the loading the test cases for the particular module. After loading the test cases, it runs the test dlls and returns the result to the client
2. Updating the test cases
   The Developer can update the test cases which were written earlier for the application under test. Modifying it requires that all the test cases should run again so that they can confirm that the behavior of the application is the same even after updating the test cases.
3. View the report of test results and results log
   After the test harness server runs the test cases, the output of the corresponding tests are stored and the test reports and results log are returned to the client.
4. Checking whether the error or bug is fixed or not in the old code after updating

When the test-harness server runs the test cases, it reports the bug, in case of any errors. Then when the same code is tested again after making some modifications into it, the same bug may come or it may fixed. The test harness server catch the old bugs by viewing the reports and checks whether it is fixed or not.

9.3 Partitions

The test-harness server consists of the following modules. These modules and their functionalities are explained below

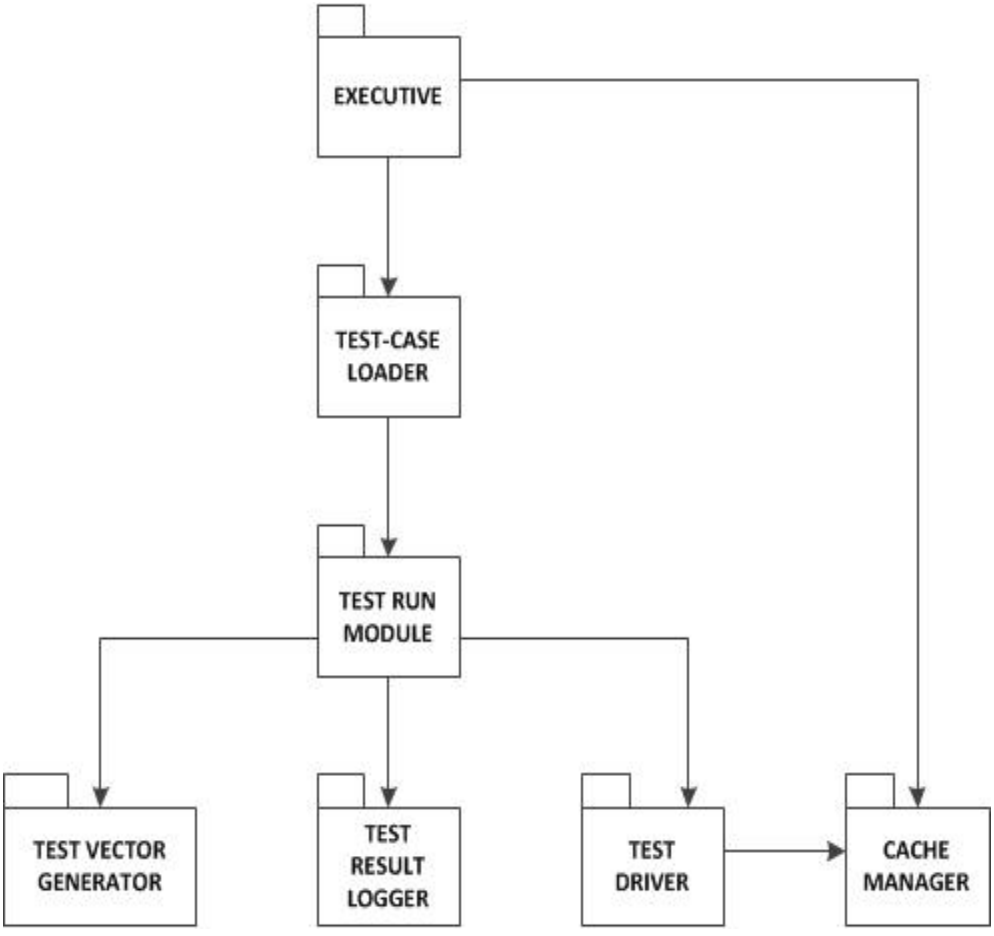


Fig 9.1: Package Diagram – Test Harness server

Executive:

The executive is the main entry point of the test harness server. It is a small package which contains the main function for the execution of the test harness server. It acts as a controller for the whole test harness module. It manages all the activities of the test harness server. The main task of the executive is to fetch the request from the client as well as repository server and forward them to the test case loader. It may be consists of test suites. It has test() method which load all the test suites and invoke this method in each of the code files. If the request code file is already compiled and stored in cache, then it will directly fetch the output from the cache manager and return to it.

Test case loader:

The test case loader module reads the request from the executive and helps them to load all the test cases based on the request, which are stored as dll files. This module sends the request to the build server for the modules that have been changed or newly added to the cache. In turn, the build server sends the recently build images to the test harness server and the test case loader receives it and loads all the test cases related to the build and can be used when the code files are testing.

Test Run module:

The test run module gets all the required data to test the source code files which need to be tested. It consists of test cases for that particular code file. It performs three functions on each test case such as initialize, test and cleanup. The test() consists of three methods: initialize(), run() and cleanup(). The main function of the initialize method is to provide an interface to the test vector generator. The actual testing is done in the run() with the specific test cases. The cleanup() part is implemented to report the test results to the test result logger module.

Test Result Logger:

The main purpose of the test result logger module is to get the test results from the test run module and store the results in some log files and return the test report to the client GUI. If the repository sends the request to the test harness server, the results are send to the repository server and stored. The test result logger module report the errors and send error notifications to the client, in case of any errors in testing. It returns the test message such as number of tests run, number of test passes, success notification and results log to the client.

Test Vector generator:

The test vector generator generates the required data/inputs for the test cases which has to be run in the test run module. The testing will be done using the inputs provided by the test vector generator. The data can be generated by using the test driver. The input from this module can also be an XML file containing the necessary data for testing.

Test Driver:

The test driver module holds all the required data or inputs for testing the source codes. It also keeps track of all the test cases for the source codes reside on the repository. It is the place where the test-suites for the project are held. It can also be in the XML format.

Cache manager:

The main purpose of the cache manager is to store all the compiled modules that are related to the current working project baseline. It maintains the compiled modules in the cache and updated when new test cases are run.

9.4 Activities

The activity diagram shown below represents the sequential order of activities performed. This diagram depicts the overall flow of control.

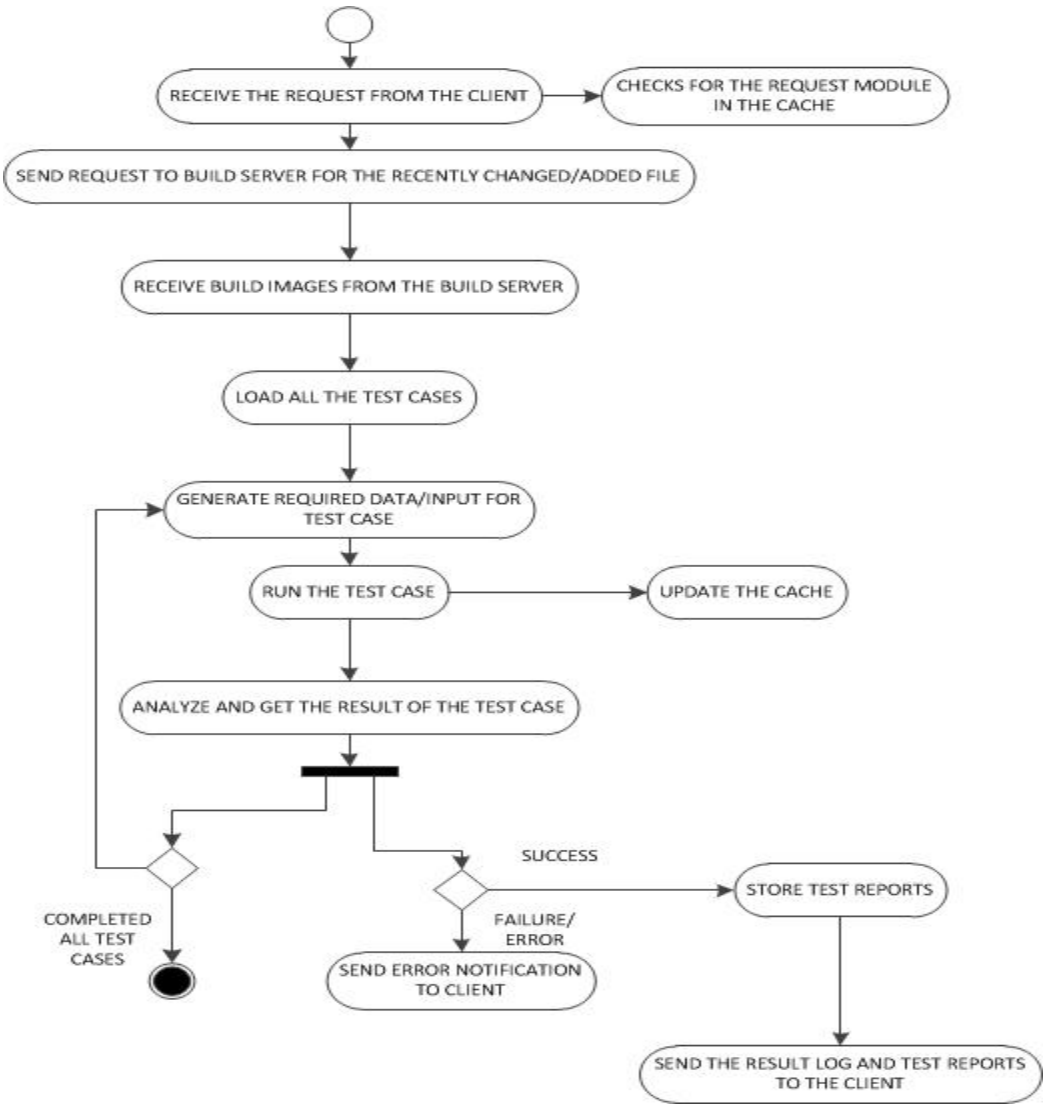


Fig 9.2: Activity Diagram – Test Harness server

Description:

1. Receive the request from the client/repository server
   The Test harness server receives the request from the client to test the specified module. When new file is checked-in or newly updated in the repository server, the repository server sends the request to the test harness to run test on that code file
2. Check for the request module in the cache
   If it receives the request from the client, it first cheks the cache whether the compiled module of the specified file is there or not. If it is there, it directly runs, no need to send request to build server.
3. Send request to the build server and receives the build images
   If the cache doesn't have the compiled module of the specified file or receives the request from the repository server, to run the recently added file, the test harness server, send request for build images of the recently added file, to the build server and receives the build images from build server.
4. Load all the test cases
   It loads all the test cases required to run the request module.
5. Generate required data/input for the test case
   The data or input required for the loaded test cases are generated by test vector generator.
6. Run the test case
   It executes the loaded test cases with the given inputs
7. Update cache
   Once it completes execution, the cache is updated by the cache manager
8. Analyze result
   It analyze the result of the like whether the test case is passed or failed, the errors it contains, number of test cases run, number of test cases passed, number of test cases failed, etc.
9. Store the result and send it to client
   The test results are stored for future reference and it sends the result report to the client.

9.5 Critical issues

Issue 1: What happened to the test case performance, if the number of test cases too large?

Solution: The test harness performance gets slow down, if there are large number of test cases over a period of time. One of the solutions to this is, to eliminate the unnecessary or undesired test cases from the test configuration. And also could eliminate the duplicate test cases.

Issue 2: Testing of test harness

Solution: It could be possible to test the test harness by using a test class for the test harness. When the test harness receives that test class, it will perform testing like the other code files.

Issue 3: What happen if the test vector generator supplies invalid inputs?

Solution: To solve this issue, we need to design the test run module such that it handles any exceptions because of this invalid data from test vector generator while running the test case.

Issue 4: What happen if the test harness server receives request from the several clients at the same time, to test the code files?

Solution: If the multiple users are using the test harness for testing the application, it is impossible to do manual tests and can't be run consistently. So, we have to design in such a way to make the test harness automated to accurately provide the set of tests.

View:

View of client GUI with test harness server:

10 Build Server

10.1 Introduction

The main purpose of the build server is to compile the source code which needed to be tested. It builds the code only when it receives the request from the test harness server or from the repository server. Based on the request, the build server gets the dynamic link library (dll) files for the corresponding source code. Then the build server builds the dll file gets the executable for the source code. The output of the build is stored in cache. If it receives the request from test harness server, the build server checks its cache for the corresponding build. If the build is in cache, then it will take the build images from the cache and return it to the test harness server. Otherwise, it build the dll file and send the build images to the test harness server. When the source code file is newly checked-in or updated in the repository, the repository server send the request to the build server to build the source code that have updated or been newly added to baseline. Then the build server builds the dll of the source code file and all the dependent files and cache it the output of the build and send the build images to the test harness for testing the newly checked-in code file. When the client sends the request to test harness to do testing for the file which is already added to baseline, then the test harness checks its cache for compiled module. If the build is not present, it sends request to build server for the compiled code for the corresponding code file. Then the build server checks its cache and returns the build images to the test harness server. The build server build the code file and sends the build images only on demand. While compiling the code, if there are any errors, then the build server send error messages to the repository server and test harness server notifying that there are build errors. The build errors include compile errors as well as linking errors.

10.2 Users and Use cases

10.2.1 Users

1. Developer
   The build server is mainly used by the developer. When they newly check-in any code, they use the build server to build the code they developed for any build errors. If there are any errors, they receive the notification, so that they can make changes to the code and update it. This ensures them that the code which is newly added to the baseline or updated is working properly to meet its function requirements.

10.2.2 Use Cases

The primary use case of this build server is when there is newly checked-in code in the repository.

1. When a code file is newly added to the repository
   Whenever a file is newly developed and checked-in by the developer to the baseline, the repository server will send request to the build server to compile the newly added source code file for checking the errors.
2. When a code file in the repository is updated
   This case is similar to the newly added file to the repository. The updated code is compiled by the build server to check whether the update code does not contain any build errors and meet the requirements
3. When client sends request for testing to the test harness server
   If the client sends the request to the test harness for testing the code file which is already added to the baseline, then the test harness send the request to the build server for the build images of the source code file which has to be tested

10.3 Activities

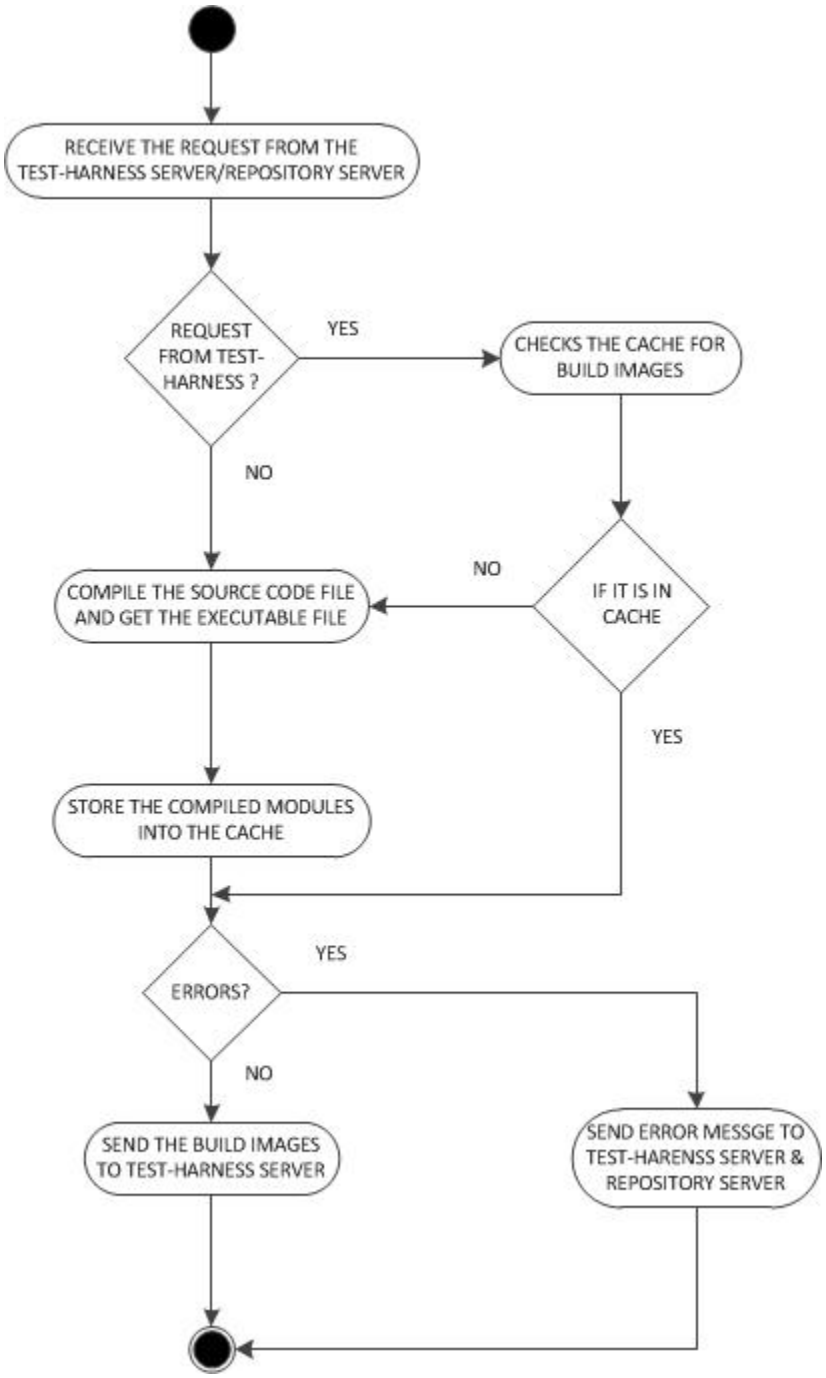


Fig 10.1: Activity diagram – Build server

Description:

1. Receives the request from the Test-Harness server or Repository server
   The build server receives the request from the test harness server to send the build images of the specified source code file. It receives the request from the repository server to build the newly added or updated source code files.

2. Check the cache for compiled modules
   If it receives the request from the test-harness server to build the already present code file, then the build server checks its cache for the compiled modules. If it is present in the cache, then it will send the build images for the specified code file to the test-harness server for testing.

3. Build dll file of the source code
   When it receives the request to compile the newly added or updated code file, the build server starts compiling the code file.

4. Store the compiled modules in the cache
   After build the dll file, the compiled module is stored in the cache for future reference.

5. Send the build images to the test-harness server
   Once the compilation is done and the build is successful, then the build server sends the build images to the test-harness server for testing the compiled module

6. Send Error notifications to the test-harness and repository server
   If there are any build errors, then the build server sends the error message to test-harness server and repository server notifying that there is compile errors or linking errors with the code files.

11 Communication Module

The communication between client and server will be handled by the communication module which is in client and server. This communication can be implemented using Windows Communication Foundation (WCF). Here, the communication channel will be created through which the request from client, sends to the server and receives the response from the server. It will implement message queue for handling the requests from the multiple clients and process them with a single thread. It opens a socket on server side with the help of communication module. Using sockets, client can communicate with the server.

The communication between the different servers is done using the message passing mechanism. Message passing mechanism simply means the object sending each other messages. The message contains the information such as destination address, source address, and message contents. Each and every client is associated with threads. A thread-safe blocking queue is used to handle the multiple requests from different users simultaneously. The blocking queue allows enqueue and dequeue of certain types and if the dequeuing operation performs on the empty queue, it waits in the queue until the item is enqueued. Servers can communicate either synchronously or asynchronously. It depends on the requirement. But in the systems, mostly the communication will be synchronous.

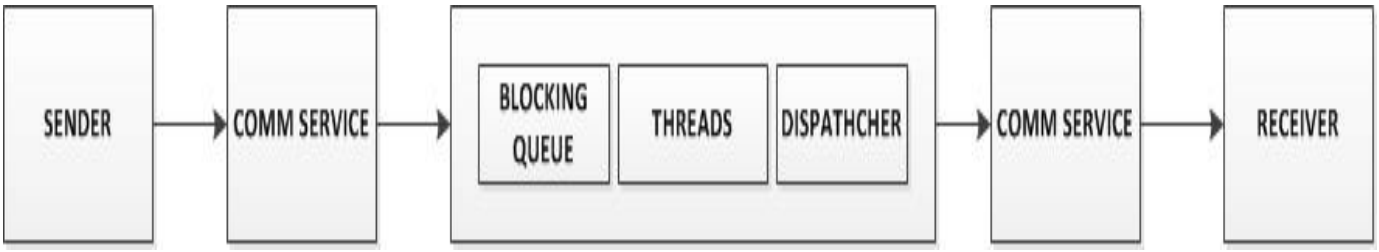


Fig 11.1: Communication channel

12 Conclusion

This architectural concept document described the overall architecture of the Continuous build and integration system. It also described the partitions of modules, activities of the system and also some critical issues and also possible views of the user interface. In conclusion, the overall features of the continuous build and integration (CBIS) system are as follows:

- Integrates all the developed code files to the project baseline
- Use of Graphical user interface for easy interaction for the users.
- Allows the developers to check-in and check-out the code files from the repository server
- Finds the dependency relationship between the code files in the repository server related to the current working project baseline.
- Executes query about the dependencies
- Ability to view and/or update metadata
- Build the code files as it is checked-in to the repository server
- Test the code files as it is checked-in
- Send error notifications in case of errors during build or test

As overall, this CBIS system provides the ability to continuously integrate the developer's code with the rest of the developing software baseline.

13 Reference

[1] CSE681 – Software Modelling and Analysis – Course Website
http://ecs.syr.edu/faculty/fawcett/handouts/webpages/CSE681.htm

[2] Midterm Solutions Fall2014 and Fall2013 by Dr. Jim Fawcett
http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/cse681codeL24.htm

[3] Wikipedia Software Repository
http://en.wikipedia.org/wiki/Software_repository

[4] Continuous Build and Integration System – Preliminary Architectural concept by Dr. Jim Fawcett
http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Projects/Pr5F14.pdf

14 Appendix

14.1 Prototype of cache for dependency-based builds in the build server

The cache manager in the build server is responsible for caching the source code modules which are previously build. This minimizes the network traffic due to transferring source code modules from repository server to build server. This also saves the time it takes to download the file and the dependent files from the repository server. The activities of caching the source code modules for dependency-based builds in the build server is shown in the figure (14.1) below.

When the build server receives the request from the repository server or from test harness by specifying code file to build, the build server receives the file name and it checks whether the source code of the given file is present or not. If it is present then the source code will be taken from the cache and build that file. It also checks the cache for the source code of its dependent file. The build server gets the dependency information from the dependency table in the repository server. If the user sends request tp test the specified file, they should also send the dependent file along with the file to be tested. So, when the test harness sends request, the build server receives both the file need to be tested and all dependent file. The build server checks its cache for the source code file of its dependent files. If the source code files are not present , it will get from the repository server and store it in the cache. If there is no space in the cache to store the files, then least recently used file is removed from the cache. This can be find using Least Recently used algorithm (LRU). Once it receives the file, the build server starts compiling the source code. Thus the cache of the build server contains the source code modules of the previous builds.

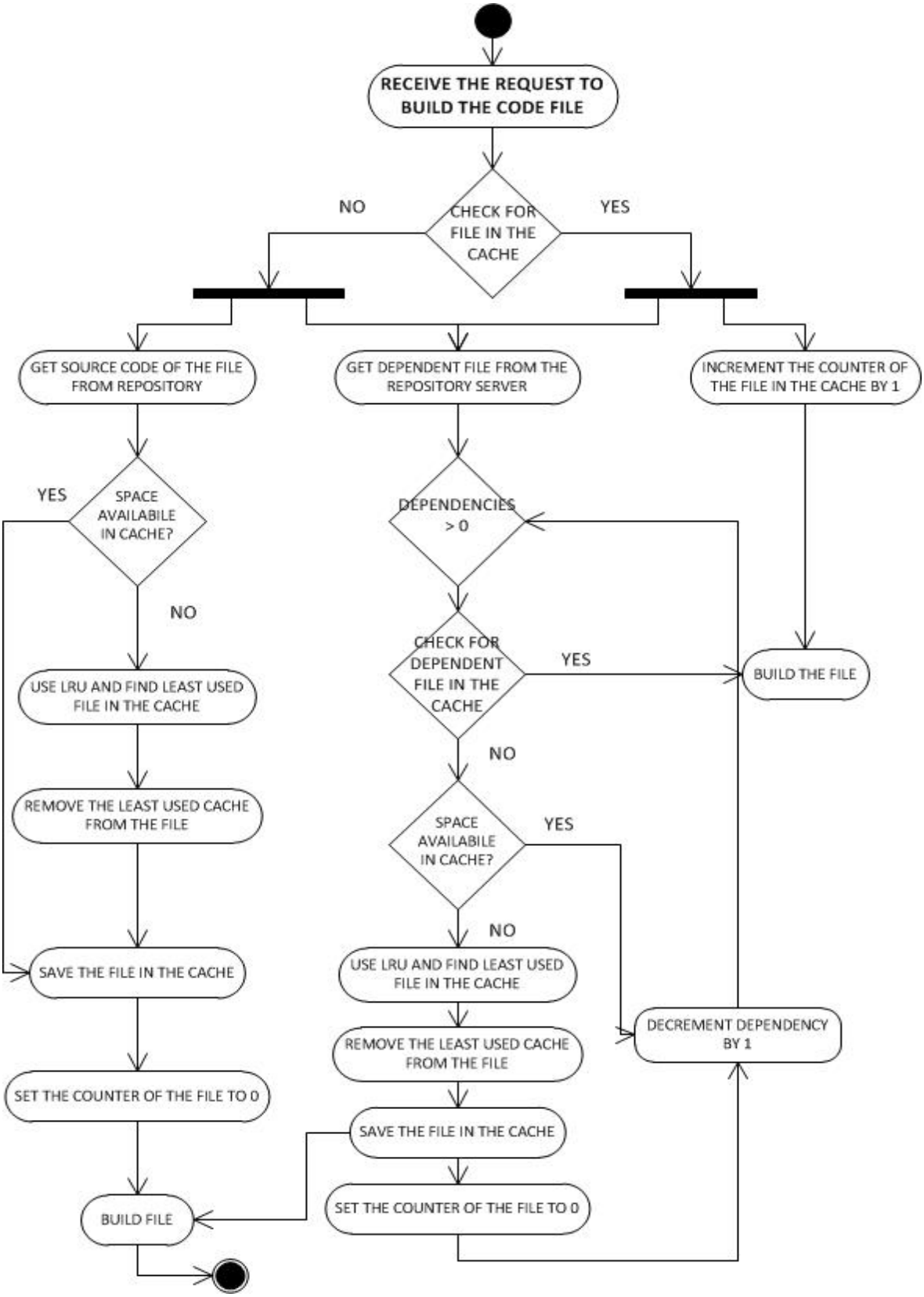


Fig 14.1: Activity diagram for cache for dependency-based builds in the build server