Core Build Server (OCD)

Project 1

Sachin Basavani Shivashankara (SUID: 267871645)
CSE-681: Software Modelling and Analysis (Fall 2017)
Sep-13-2017

Instructor: Dr. Jim Fawcett

Contents

Table of Figures

Executive Summary

"Complex software architectures are extremely hard to manage, not only in terms of the architecture process itself but also in terms of getting buy-in from large numbers of stakeholders. This, in turn, requires a very disciplined approach to identifying common architectural components, and management of the commonalities between federated components — deciding how to integrate, what to integrate, etc."[1]. The pattern/architecture that enforces this approach is called a "Federated Software Architecture".

The advantages that result from a good enterprise architecture bring important business benefits like,

- Lower business operation costs
- More Agile organization
- Improved business productivity
- A more efficient IT operation
    - Lower software development, support, and maintenance costs
    - Increased portability of applications
    - Improved interoperability and easier system and network management
    - Improved ability to address critical enterprise-wide issues like security
    - Easier upgrade and exchange of system components

So, what is Federated Architecture? Federated Architecture is expected to deliver high flexibility and agility among independently cooperating components and at the same time reduce complexity significantly. A federated architecture should be considered for problems with the root cause of unmanageable complexity. This approach is applicable for decoupling and decentralizing projects where a central one-fits-all approach cannot be applied. This approach is especially applicable for long-term migration projects.

When a new code is created for a system, it is built and tested in the context of other code. As soon as all the tests pass, the code is checked-in and becomes a part of the current baseline. There are several services necessary to efficiently support this continuous integration. In this course, we will be developing the concept for and creating, a federation of servers each providing a dedicated service for continuous integration using the federated architecture pattern.

We will be focussing on "Build Server"- an automated tool that builds test libraries. Along with the Build Server, the system consists of other modules with their own set of responsibilities. This structure supports continuous integration which is an excellent practice to follow when developing software. The subsystems of this federated structure are:

- **Repository:** Holds all code and documents for the current baseline along with their dependency relationships. It also holds test results and build images.
- **Build Server:** Based on build requests and code sent from the Repository, the Build Server builds test libraries for submission to the Test Harness.

1. TOGAF Version 9 Enterprise Edition.

- **Test Harness:** Runs tests concurrently for multiple users based on test requests and libraries sent from the Build Server.
- **Client:** The primary interface into the Federation. Provides GUI to submit code and test requests to the Repository.
- **Collaboration Server:** Provides various management capabilities like remote meetings, shared digital whiteboard, shared calendars etc.

In this document, we discuss the concept, package structure and the organizing principle of the whole federation and concept, activities and users of each of the subsystems in detail. We also discuss some of the issues which affect the key operations of the federation. We also build a prototype and discuss the setup required to programmatically build a project prototype, which serves as the base for the Build Server.

Introduction

The tool is a federated architecture that allows interoperability and information sharing between de-centrally organized modules that provides facilities to run tests, record test status and log execution details efficiently. This tool with some enhancements can be used to make software development and testing easier.

Organizing Principle

The application should support continuous integration and automated testing using modules like Client, Repository, Build Server and Test Harness which communicate effectively among themselves to increase productivity.

Structure and Partitioning

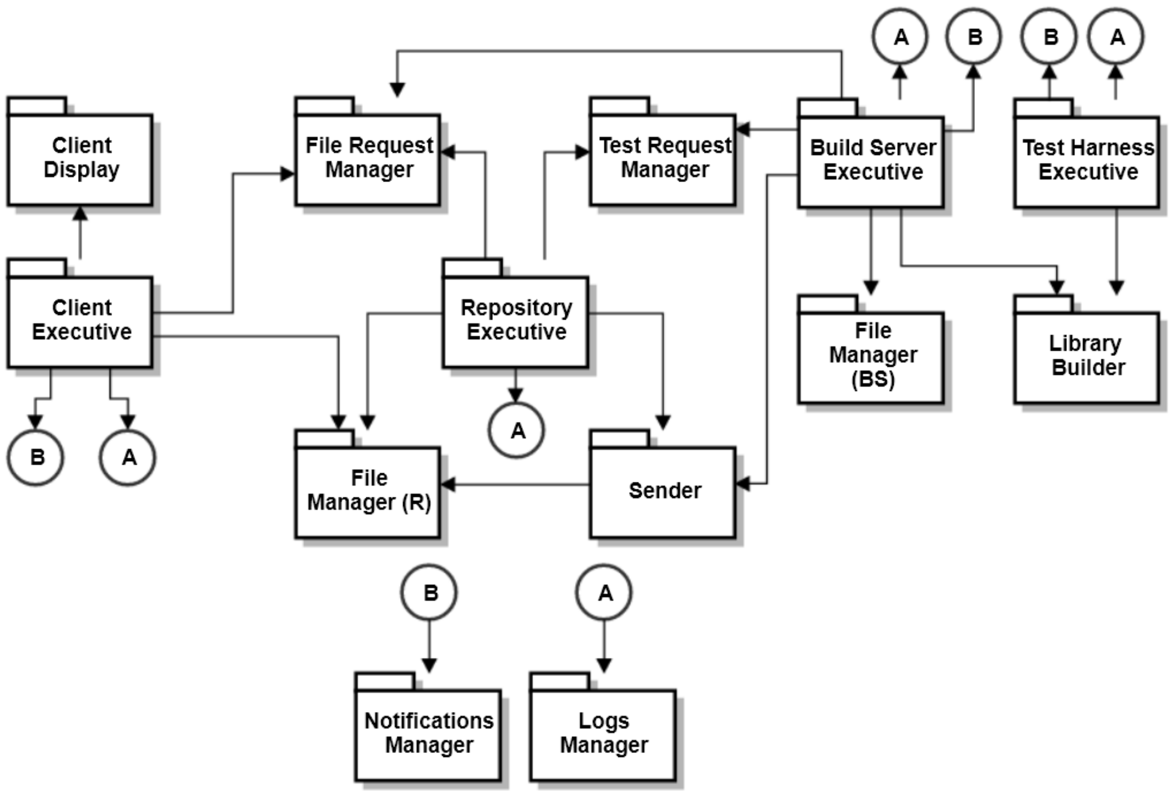


Figure 1: Overall Package Diagram

- Client Display: Gives option for the user to input the details required for running the test. The data is inputted through the Console Window (in Project 2 and GUI in Project 4). Shows the contents of the Repository. Can view the test and build logs. Can view notifications from Test Harness.
- Client Executive: Entry point into the system. Takes input from the console window (in Project 2 and through selections made on the GUI in Project 4) and generates a test request XML file. Processes logs and notifications. This package also accesses the contents of the Repository for display purpose.

- File Request Manager: Processes the requests for files from Client and the Build Server and schedules transfer of files.
- Test Request Manager: It is an XML parser package which decodes the test request XML received from the Client.
- File Manager (R): File Manager in the Repository which holds all the files required for testing. And other files like build and test logs.
- Sender: Helps in sending files to the Build Server on command.
- Repository Executive: Main package of the Repository. Controls all other packages in the Repository.
- File Manager (BS): Caches the files received from the Repository. These files are used to build the test libraries.
- Library Builder: Generates the executable (libraries) from the files received (from the Repository) based on the parsed test request.
- Build Server Executive: Main package of the Build Server. Controls the Build Server system.
- Test Harness Executive: Main package of the Test Harness system. Perform execution of the test using the files received from the Build Server.
- Notifications Manager: Receives and processes message from the Build Server and Test Harness regarding the build status and test status. Uses Client Executive and Client Display to display the notifications.
- Logs Manager: Receives and processes build logs and test logs on command from the Repository, Build Server and Test Harness. Uses Client executive and Client Display to display the logs.

Overall Application Activities

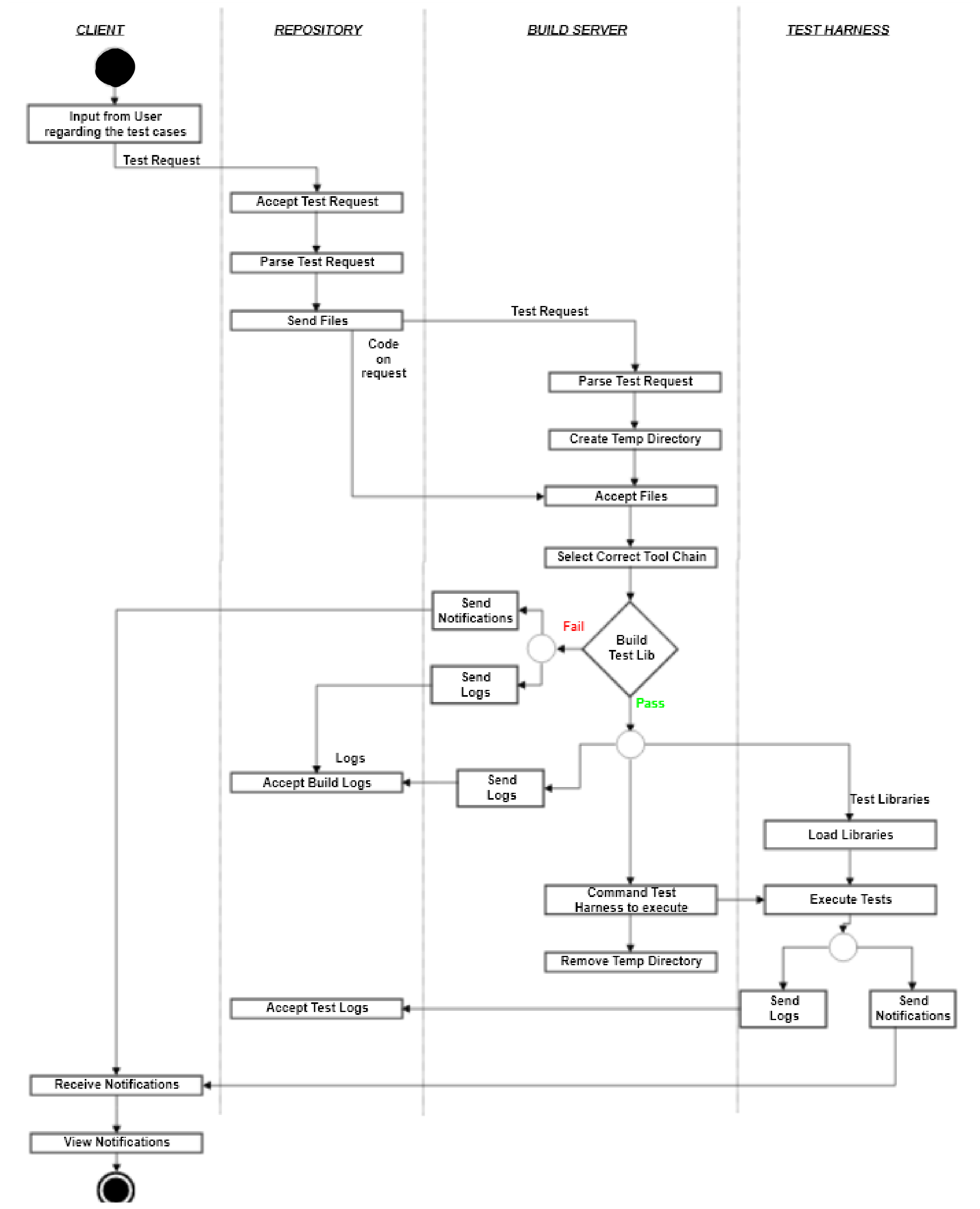


Figure 2: Overall Activity Diagram

Description:

- The User of the tool uses the GUI (to be implemented in Project 4) of the Client to select the files required for testing. Client generates the test request based on the input.
- The Client then sends the test request to the Repository.

- Commands the Repository to process the test request.
- Then, the Repository takes over. It accepts the test request from the Client.
- On command, the Repository parses the test request to understand what files are required for the tests and sends the files (test request and the code files) to the Build Server.
- Build Server on accepting the files, parses the test request to build individual test libraries.
- Then, it creates a temporary directory to cache the files into this directory.
- Selects a tool chain to run the proper compiler (C#/ C++/ Java compiler).
- Then, the Build Server attempts to build the test libraries.
- If build fails (due to a missing part or wrongly set properties etc.), sends a notification to the Client of the failure.
- If build passes, sends the test libraries to the Test Harness for execution. Also, sends a notification to the Client of the success.
- The Build Server then commands the Test Harness to execute the tests.
- The Test Harness loads the libraries received from the Build Server.
- Then, on command, executes the test.
- Sends the test logs to the Repository.
- Sends notification to the Client.
- Build Server accepts the test logs from the Test Harness.
- Client receives the build logs and test logs from the Repository on command.

Core Build Server

Concept

The main purpose of a Build Server is to build projects based on the data from the test request. This Build Server along with other systems like Repository, Test Harness provides facilities for Continuous Integration. It enforces the practice of merging all developer working copies to a shared baseline. Continuous Integration supports systems building only from the repository, this way, build packages are reproducible and traceable.

A Build Server:

- Simplifies the developer's workflow and reduce the chance of mistakes and breaking in production (due to some stray DLL's on their individual machines etc.) as it replicates the target environment. That is, it avoids "but it works on my machine" issue.
- Can be easily automated.
- Sets up builds for different versions for different stages of the application (not implemented in project).

Uses and Users

The Build Server uses the test request and the associated code sent from the repository and builds test libraries. The Test Harness uses Build Server to receive test libraries to run the tests.

QA and Test team use the build logs generated by the Build Server to document the build report.

Application Activities

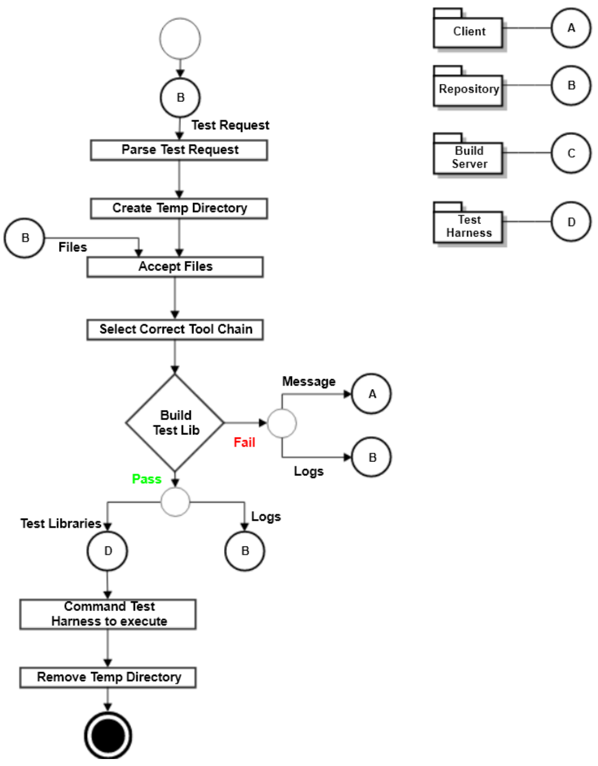


Figure 3: Activity Diagram of Build Server

The activity diagram given above shows the activities done by the Build Server to accomplish the task. Given below is the description of the activities:

- **Accept test request:** Accept test request from the Client (through Repository).
- **Parse test request to see how each test is configured:** Analyse the test request to build individual test libraries (parsing not needed if just 1 DLL is built from all the test requests).
- **Create temp directory:** The Build Server caches the code files so that it can look up its cache, the next time it is building code, before requesting to send the required code.
- **Accept files:** Request and accept the required files from the Repository.
- **Parse test request to select correct tool chain (C#/ C++/ Java):** Parse the test request to select either the C# compiler or C++ compiler or Java compiler.
- **Attempt to build test libraries:** Tries building the test libraries from the received files.
- **Send notice to client:**
    - **If build fails:** If build fails due to a missing part or wrongly set properties etc. sends a message to the client of the failure.
    - **If build succeeds:** send libraries to the Test Harness for execution.
- **Command Test Harness to execute tests:** Send notification to the Test Harness to execute the test.
- **Remove temp directory**

Repository

Concept

The main purpose of a repository is to store a set of files, as well as the history of changes made to those files and to place all software artifacts required to build a project. In this project, we implement a mock repository which holds all code and documents for the current baseline, along with their dependency relationships and also holds test results and build images. But differs from a fully functional repository as it does not provide facilities for versioning or checking-in/ checking-out files, querying the repository contents etc.

In a version controlled repository (not implemented in the project), the convention is that the system should be buildable from a fresh checkout and not require additional dependencies. It is also preferred for changes to be integrated rather than for multiple versions of the software to be maintained simultaneously. The mainline or the baseline should be the place for the working version of the application.

Uses and Users

The main purpose of a repository is to store a set of files. It stores baseline code and test logs. The user can upload and download files (code and test logs) from the repository using the GUI provided by the client. Through the client the repository keeps every user of the tool aware of the latest changes in the repository.

Whenever a developer needs to test/use some code, they need that file and all its associated files to successfully build it. Instead of checking through the references of the file manually, the repository can use the metadata to easily find the dependant files and download them all along with the required file.

Project managers can use the repository to check progress of the project by analysing the files checked into the repository. Also, when a customer reports an issue for an older version, codes and logs for that version can be easily assessed to take corrective actions.

The QA team can access and analyse build and test logs of any version to analyse and maintain the quality of the code.

Test team can easily extract the build and test logs to document the test report.

Application Activities

The purpose of the repository is to store a set of files (codes, logs etc.). And give access for data transfer to different parts of the system. The Repository does the following tasks to facilitate transfer of the required files:

- Accept test request from Client.
- On command
    - Parses test request.
    - Sends files to Build Server.
    - Accepts build logs from Test Harness.
    - Accept test logs from Test Harness.
- On command
    - Sends logs or files to the Client

Client

Concept

The main purpose of a client is to provide access to different servers of the system. That is, the client provides a GUI (Graphical User Interface) [to be implemented in Project 4] to interact with the Test Harness, Repository and the Collaboration Server. Another important purpose of the client is to view test logs and build logs. The advantage of using a GUI is that, the selections made in the GUI will be internally translated and given to these subsystems as input, thereby avoids manually keying in the input which avoids human errors.

Uses and Users

The client provides an easy way to access the different servers of the application using a GUI. It initiates the repository to send the test requests and its associated files to the build sever based on the selections made on the GUI. Client can also be used to view the build logs and the test logs generated from the Build Server and Test Harness. Client gives an easy and interactive way to peek into the application and to monitor the health of the application.

Developers can use the client to interact with the repository by checking-in the files for testing, or can check-out the required files to make enhancements (not implemented in the project). They can also use the Client to view the test results of their code files.

QA and test teams can use the client to extract the build logs and test logs to document the test/build reports. They can also use it to monitor and maintain the quality of the code.

Project managers can use the Client to check the progress of a project. They can use old test issues and test results to assess any complaints from the customer. They can also use these files to educate the team of any old issue and its resolution.

Client can be used by the TA's to check if the requirements of the project have been fulfilled.

Application Activities

The client is responsible for the initiation of the testing and to view the results after the process is complete. After it initiates, other subsystems of the application take over to successfully complete the process. Following are the tasks done by the client:

- Creates test request.
- Sends the test request to the Repository.
- Commands the Repository to process the test request.
- Accept build and test logs from the Repository.

Test Harness

Concept

A Test Harness is a collection of software and test data configured to run a test program unit. Test Harness is software constructed to facilitate Integration Testing and it allows for automation of tests. Test Harness executes test suites of test cases and generates associated test reports. The benefits of having a Test Harness are:

- Increased productivity due to automation of the testing process.
- Repeatability of subsequent test runs.
- Increased quality of software components and application.

A Test Harness has no knowledge of test suites, test cases or test reports. Those things are provided to the Test Harness by the associated framework like Build Server and Repository.

Test Harness runs tests, concurrently for multiple users, based on test requests and libraries sent from the Build Server. Clients will check-in, to the Repository, code for testing, along with one or more test requests. The repository then sends code and requests to the Build Server which then builds libraries. The test requests and libraries are then sent to the Test Harness. The Test Harness executes tests, logs results, and submits results to the Repository. It also notifies the author of the tests of the results.

Uses and Users

Test Harness uses the libraries received from the Build Server to perform testing. Test Harness provides the facility to perform Integration Testing of an application in an environment for which it has been developed. As the Test Harness tests the complete system even when only a single component is modified, it increases the quality of the system. That is, it allows for thorough testing of the application even for a small modification.

Developers make use of the facilities provided by the Test Harness to avoid rework as the whole process can be automated, thereby, reducing cost and time to the company.

QA team can monitor the quality (standards) of the application by analysing and documenting the test logs provided by the Test Harness.

Project Managers can also use these logs to monitor the progress of the project by knowing which tests are working well and which are failing.

Application Activities

Test Harness is the subsystem which executes the tests. Based on the libraries received from the Repository, the Test Harness loads and executes the tests. Given below are the steps involved:

- On command
  - Loads libraries.
  - Executes tests.
  - Sends test logs to the Repository.
  - Sends notification to the Client.

Critical Issues

1. Productivity: The main purpose of this tool is to support continuous integration and automated testing. All the dependant systems of the modified system must be thoroughly tested to ensure no breakage in the system exists. To perform all these tasks, the subsystems provide facilities like a GUI for easy inputting, automatically including the dependant files (from the metadata), programmatically generating the test data etc. that reduce rework for the developers and hence saving time and revenue for the company. Thereby, increasing productivity.

2. Communication: As the size of the application increases, the number of issues also increase. So, it is extremely important that the appropriate teams are notified of the issues so that corrective actions can be taken. Also, to ensure that all the users of this tool are on the same page, so that, depending on the communicated message a process is initiated that everybody involved in the process agrees on and benefits everyone. So, the communication system between the subsystems must be effective and efficient.

3. Configuration: The application must be versatile enough to be able to build codes of multiple programming languages. This can be done by implementing .NET environment class and .NET process class.

4. Load: As the size of the application increases, number of users (developers, managers etc.), number of files, number of tests, number of requests also increases. This situation may cause the system to operate at its limits thereby reducing the performance of the system. In order to regain the performance, we can add multiple Repositories, Build Servers and Test Harnesses to divide the load between the subsystems.

5. Long builds: When the load increases, we will have multiple requests at the same time. And, as we build the test files in sequence, the build might take a long time, which deteriorates the performance. So, we can setup the Build Server to build the tests in parallel, so that, the build takes shorter time, thereby, increasing the performance of the system.

6. Multiple Clients: When there are multiple users working on the same project, a clash can occur when two or more users try to check-in the same code at the same time. This will result in code being overwritten. So, single ownership methodology can be implemented which gives ownership of the file (when checked-in) to a single user and preventing all other users from checking-in the same file.

7. Security: Security is a major concern for large companies. When the Test Harness runs a malicious library (either from an intruder or inadvertently by a user of another application), the application might get corrupted and the security (or normal operation) of the system may be compromised. Therefore, to prevent malicious DLL’s from being executed we need to ensure the Test Harness gets the libraries only from the Build Server which builds files checked-in to the Repository by verified users. Verified users being developers, managers, QA/test team etc. i.e. only people with an approved clearance (from the company or that particular project).

Conclusion

In conclusion, the Core Build Server based on build requests and code sent from the Repository, builds test libraries for submission to the Test Harness. When new code is created for a system, we can build it and test it in the context of other code which it calls, and which call it. As soon as all the tests pass, we check-in the code and it becomes the part of the current baseline. The Core Build Server along with other systems like Repository, Client, Test Harness and Collaboration Server efficiently support Continuous Integration.

The system is divided into packages which can interact with each other to effectively perform all the tasks as per requirements. This OCD explains the concept, uses, users, the package structure and key activities of each package with relevant diagrams like package diagram and activity diagram. The document also discusses few critical issues and its implications along with the solutions.

Appendix

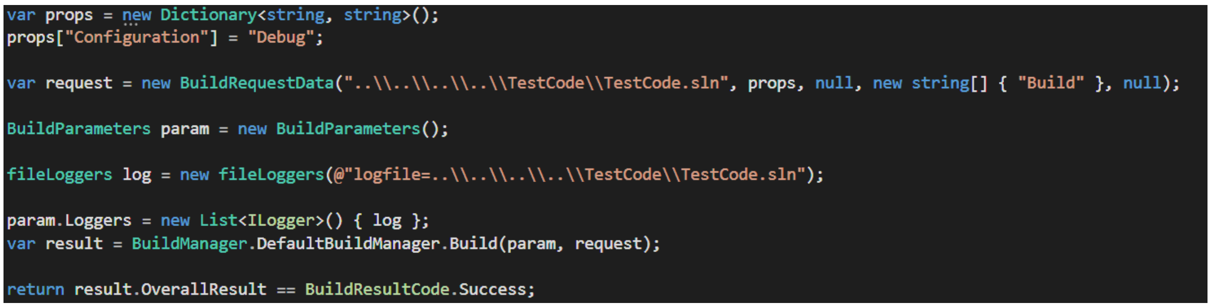
In the prototype, MSBuild framework is used to build a project. Before we start setting-up the properties of the builder we need to include few namespaces particular to MSBuild (using Microsoft.Build.Framework;, using Microsoft.Build.Execution; etc.) in the prototype program.

We use constructs like BuildRequestData, BuildParameters, BuildManager etc. provided by MSBuild to set up the parameters required for building a project. MSBuild also provides Logger construct to log the build status into a log file (or console window).

Setting-up parameters:

BuildRequestData constructor is used to set-up the build request.

BuildRequestData("..\\..\\..\\..\\TestCode\\TestCode.sln", props, null, new string[] { "Build" }, null).



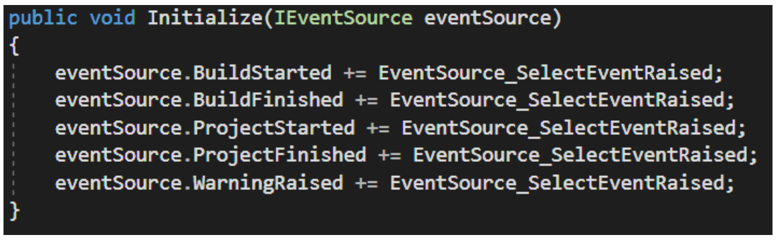
```
var props = new Dictionary<string, string>();
props["Configuration"] = "Debug";

var request = new BuildRequestData("..\\..\\..\\..\\TestCode\\TestCode.sln", props, null, new string[] { "Build" }, null);

BuildParameters param = new BuildParameters();

fileLoggers log = new fileLoggers(@"logfile=..\\..\\..\\..\\TestCode\\TestCode.sln");

param.Loggers = new List<ILogger>() { log };
var result = BuildManager.DefaultBuildManager.Build(param, request);

return result.OverallResult == BuildResultCode.Success;
```

Figure 4: Build Request setup [prototype-Appendix]

Here, we set the full path of the project file to be built (TestCode.sln in the prototype), global properties ("configuration" to "debug"), the tool version to be used, targets to build, host services to use (if any).

To view the status of the builds requested, we use the Ilogger interface provided by MSBuild. A log file can be created for every build request and the status of the builds can be logged into it. In the prototype, the status of the builds is directly displayed on the console window (logging into a file not implemented in the prototype).

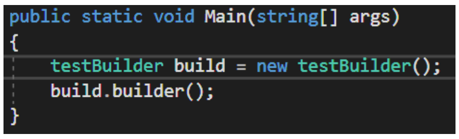
File logger generates a ton of information for the user for different events occurring during the build. For brevity, in the prototype, only certain events are logged using the "EventSource" function. Certain events like start/end of build, start/end of the selected project for build, any warnings during build are shown on the console window.



```
public void Initialize(IEventSource eventSource)
{
    eventSource.BuildStarted += EventSource_SelectEventRaised;
    eventSource.BuildFinished += EventSource_SelectEventRaised;
    eventSource.ProjectStarted += EventSource_SelectEventRaised;
    eventSource.ProjectFinished += EventSource_SelectEventRaised;
    eventSource.WarningRaised += EventSource_SelectEventRaised;
}
```

Figure 5: Logger setup [prototype-Appendix]

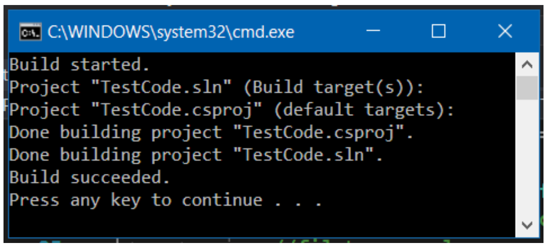
The build request is called from the Main function to build the project.



```
public static void Main(string[] args)
{
    testBuilder build = new testBuilder();
    build.builder();
}
```

Figure 6: Build in Main() [prototype-Appendix]

When we build and run the prototype and if the build setup is done properly the prototype builds the project specified in the setup (TestCode.sln in our case). This generates a .dll file (or .exe file if the project is not setup as a class library) in the \bin folder of the project to be built. The following output is generated for the build setup shown above.



```
C:\WINDOWS\system32\cmd.exe
Build started.
Project "TestCode.sln" (Build target(s)):
Project "TestCode.csproj" (default targets):
Done building project "TestCode.csproj".
Done building project "TestCode.sln".
Build succeeded.
Press any key to continue . . .
```

Figure 7: Output [prototype-Appendix]

The prototype forms the base of what we will be building for the Build Server in Project 2 and Project 4. Here, the solution path for the project to be built is hardcoded, in Project 2 and Project 4 this input will be provided by the Client in the form of test requests (<tested> tag in the test request XML). But this idea of building another project programmatically forms the core of our Build Server. The DLL's generated from the builds can then be transferred to the Test Harness to be executed by it. Also, by effectively utilizing the facilities provided by the ILogger interface we can generate build logs which display relevant information.

References

1. Wikipedia: 1, 2.
2. TOGAF Version 9 Enterprise Edition
3. Stack Overflow: 1, 2, 3.
4. CSE-681 SMA resources: Instructor's (Dr. Jim Fawcett's) resources on the website.