Project #2 - Core Build Server

Version 1.0, Revised: 08/27/2017 18:03:57
Due Date: Wednesday, October 4th

Purpose:

One focus area for this course is understanding how to structure and implement big software systems. By big we mean systems that may consist of hundreds or even thousands of packages[1] and perhaps several million lines of code.

In order to successfully implement big systems we need to partition code into relatively small parts and thoroughly test each of the parts before inserting them into the software baseline[2]. As new parts are added to the baseline and as we make changes to fix latent errors or performance problems we will re-run test sequences for those parts and, perhaps, for the entire baseline. Because there are so many packages the only way to make this intensive testing practical is to automate the process. How we do that is related to projects for this year.

The process, described above, supports continuous integration. That is, when new code is created for a system, we build 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 part of the current baseline. There are several services necessary to efficiently support continuous integration, as shown in the Figure 1., below, a Federation of servers, each providing a dedicated service for continuous integration.

click to zoom

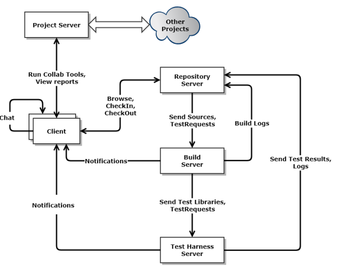


Figure 1. Software Collaboration Federation

The Federation consists of:

- **Repository:**
  Holds all code and documents for the current baseline, along with their dependency relationships. It also holds test results and may cache 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. On completion, if successful, the build server submits test libraries and test requests to the Test Harness, and sends build logs to the Repository.
- **Test Harness:**
  Runs tests, concurrently for multiple users, based on test requests and libraries sent from the Build Server. Clients will checkin, to the Repository, code for testing, along with one or more test requests. The repository sends code and requests to the Build Server, where the code is built into libraries and 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.
- **Client:**
  The user's primary interface into the Federation, serves to submit code and test requests to the Repository. Later, it will be used view test results, stored in the repository.
- **Collaboration Server:**
  The Collab Server provides services for: remote meetings, shared digital whiteboard, shared calendars. It also stores workplans, schedules, database of action items, etc.

Core Build Server:

In the four projects for this course, we will be developing the concept for, and creating, one of these federated servers, the Build Server - an automated tool that builds test libraries. Each test execution, in the Test Harness, runs a library consisting of test driver and a small set of tested packages, recording pass status, and perhaps logging execution details. Test requests and code are submitted by the Repository to the Build Server. The Build Server then builds libraries needed for each test request, and submits the request and libraries to the Test Harness, where they are executed.

The four projects each have a specific role leading to the final Remote Build Server:

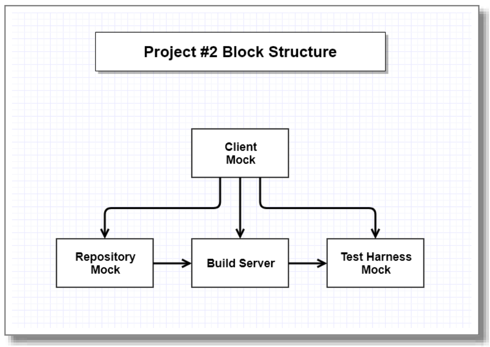
- For Project #1 you will create an Operational Concept Document (OCD) for a Remote Build Server, and a small prototype demonstrating programmable builds.
- Project #2 focuses on building the core Build Server Functionality, and thoroughly testing to ensure that it functions as expected.
- In Project #3, you will build prototypes for Process Pools, Socket-based Message-passing Communication, and for a Graphical User Interface (GUI), packages all needed for the last project. These are relatively small "proof-of-concept" projects in which you experiment with design and implementation strategies.
- Finally, in Project #4 you will build a Remote Build Server, using parts you developed in earlier projects. You will also add design details to your OCD, from Project #1, to create an "as-built" design document.

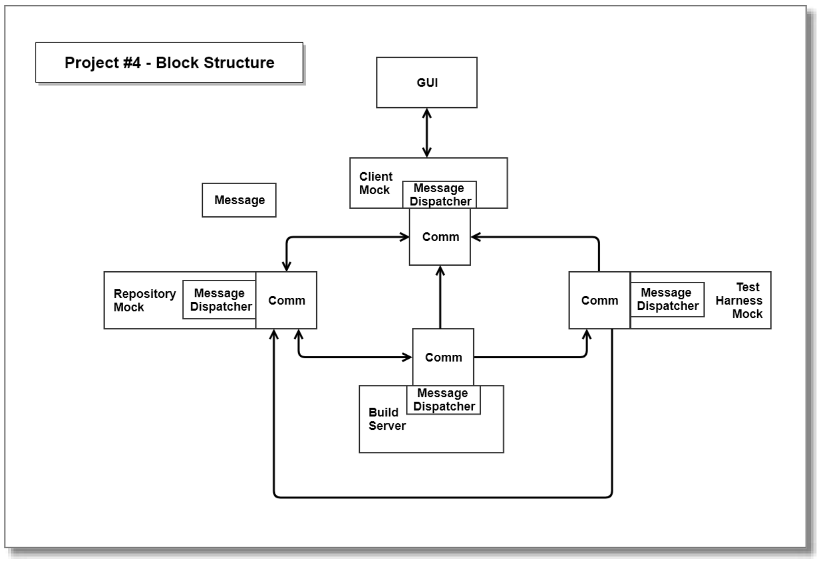
T N P B

So, for this project #2, we develop core functionality of a Build Server that will grow into a Remote Build Server over the following projects. Note that we will not be integrating our Build Server with a Federation's Repository and Test Harness Servers.

Instead, we will, in this project, build code for a single process, the Core Build Server. Its source will contain packages for mock Repository and Test Harness servers that supply just enough functionality to demonstrate operations of your core Build Server.

The core Build Server's Executive package serves as a local mock client, and the mock Repository and Test Harness are simply classes that provide interfaces for the executive to call, so there is no need for a communication channel. The mock servers will have just enough functionality to demonstrate Build Server operation.





We will discuss all of this at length in class and the help sessions.

Requirements:

Your Core Project Builder

1. **Shall** be prepared using C#, the .Net Frameowrk, and Visual Studio 2017.
2. **Shall** include packages for an Executive, mock Repository, and mock Test Harness, as well as packages for the Core Project Builder.
3. The Executive **shall** construct a fixed sequence of operations of the mock Repository, mock Test Harness, and Core Project Builder, to demonstrate Builder operations.
4. The mock Repository **shall**, on command, copy a set of test source code files, test drivers, and a test request[3] with a test for each test driver, to a path known to the Core Project Builder.
5. The Core Project Builder **shall** attempt to build each Visual Studio project delivered by the mock Repository, using the delivered code.
6. The Core Builder **shall** report, to the Console, success or failure of the build attempt, and any warnings emitted.
7. The Core Builder **shall**, on success, deliver the built library to a path known by the mock Test Harness.
8. The mock Test Harness **shall** attempt to load and execute each test library, catching any execeptions that may be emitted, and report sucess or failure and any exception messages, to the Console.
9. One interesting issue: it would be desireable to build sources from several often used languages, e.g., C#, C++, Java, etc. If you base your build process on the MSBuild library, that won't work for Java and won't work for anything on Linux platforms. Building for multiple platforms with multiple source languages may not be as difficult as it sounds. The idea is to create your own Build infrastruction that uses the compilers and their tool chains for the target platform. Essentially, you need to set up a configuration file for each platform and toolchain that identifies the paths to the tools you need, and translates your Builders commands into those needed by specific tool chains.

   This isn't required. You may simply build C# projects, using a Visual Studio generated project file and the MSBuild system. If, however, you elect to try creating the more capable Builder, you will be awarded bonus points, up to 6 points if everything works.

Note that you will only get credit for requirements that are clearly demonstrated by the operations of the Executive and it's colleagues.

---

1. In C# a package is a single file that has a prologue, consisting of comments that describe the package and its operations, one or more class implementations, and a test stub main function that serves as a construction test while building the package. This test stub is quite different from the test drivers we build with the Build Server and execute in the Test Harness. We will discuss these differences in detail in class.
2. A software baseline consists of all the code that we currently consider being part of the developing project, e.g., code that will eventually be delivered as part of the project results. It does not include prototypes and code from other projects that we are examining for possible later inclusion in the current project.
3. A Test Request is an XML file that identifies one or more tests. Each test has a test driver that implements an ITest interface and a set of packages that are associated with that test driver. So, the Mock Repository will have to know how to create Test Requests, and the Builder has to know how to parse them.

T N P B

What you need to know:

In order to successfully meet these requirements you will need to know:

1. How to programmatically build C# projects. Here's some help.
2. The definition of the term **package** and have looked carefully at a few examples.
3. Definitions for Dynamic Link Libraries - see the class text, C# 6.0 in a Nutshell.

T N P B