Project #1 - Build Server OCD

Version 1.0, Revised: 08/27/2017 15:40:19
Due Date: Wednesday, September 13th

Purpose:

The acronym OCD stands for Operational Concept Document. It's purpose is to make you think critically about the design and implementation of a project before committing to code. It also serves to publish your concept to the development team, which for this course is you (and only you). For this project we will be writing an Operational Concept Document for the remaining projects, e.g., Projects #2, #3, and #4.

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. We won't be building anything quite that large, but our projects may be considerably bigger than anything you've worked on before.

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

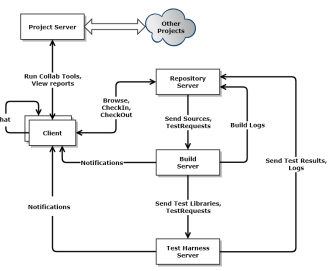


Figure 1. Software Collaboration Federation

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.

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

T N P B

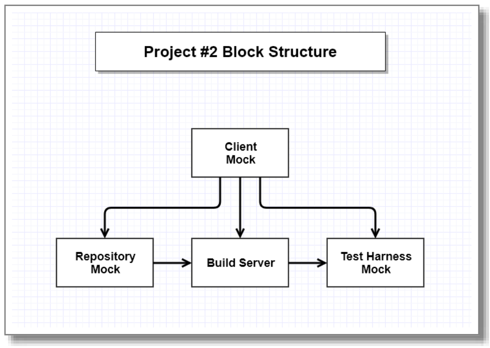
- In Project #3, you will build prototypes for Process Pools, Socket-based Message-passing Communication, and for a Graphical User Interface (GUI), packages all needed for the last project. These are relatively small "proof-of-concept" projects in which you experiment with design and implementation strategies.
- 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.

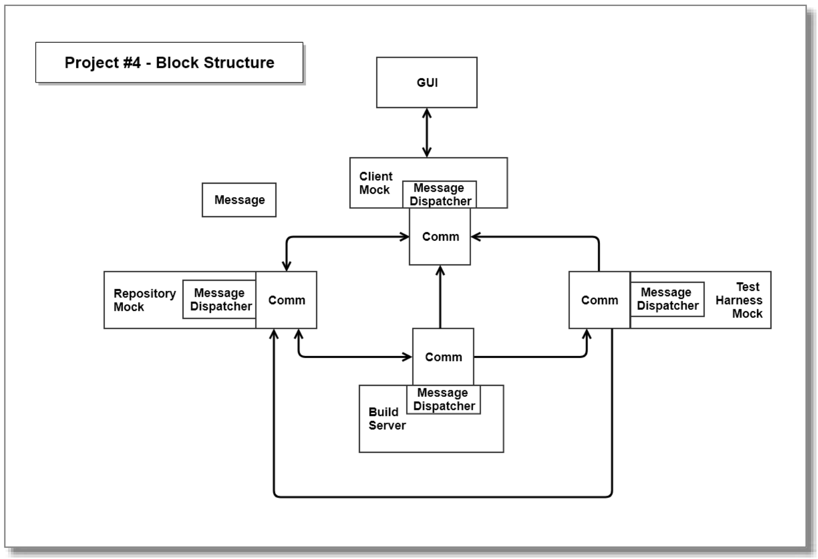
So, for this project we will develop and document a concept for a Build Server that we implement over the projects that follow. Your concept should explore:

- Project purpose and typical users - consider impact of uses on your project design.
- Ways to make the user interface(s) effective.
- Suitable partitioning of processing into packages. Provide package diagram(s) and consider the responsibilities of each package and possible implementation approaches.
- Means to present results to the user.
- Important events and critical processing.

Note that, in these projects, we will not be integrating our Build Server with a Federation's Repository and Test Harness Servers. Instead, we will build mock Repository and Test Harness servers that supply just enough functionality to demonstrate operations of your Remote Build Server. The Build Server will use a "Federation ready" communication channel to communicate with the mock servers, and we will build a mock client that has just enough functionality to demonstrate Build Server working in this environment.

So the mock Repository and mock Test Harness are simple servers, running in the Build Server's process in Project #2, and in their own processes in Project #4, using our Message-Passing Communication, to send and receive requests and replys. However, the Mock operations are simple - not nearly as complex as full up Federated servers.





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

Requirements:

Your Build Server OCD

1. **Shall** be prepared as a Microsoft Office Word file, using embedded Visio Diagrams[3].
2. **Shall** explore and describe the user interface(s) you will provide.
3. **Shall** partition processing into at least the top-level packages.
4. **Shall** describe the uses/responsibilities, activities, events, and interactions of each of the packages in your concept.
5. **Shall** use both text and diagrams for the descriptions in 4, above.
6. **Shall** prepare a Visual Studio Project builder code prototype[4] which:
   - Loads a Visual Studio project from a specified path, that refers to C# packages in subdirectories, and attempts to build the project.
   - Reports success or failure, and any warnings encountered by the build.

   Discuss the prototype results and draw conclusions about what you've learned from the prototypes, in your OCD.
7. **Shall** document prototype code you develop in an Appendix. You don't need to include source code in your OCD document (please don't), but you do need to discuss the prototype design and show outputs. Please include your source code in a zip file used to submit your project for grading.

T N P B

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. You may use alternate office suites like WPS or LibreOffice, and diagrammers like gliffy, available as a chrome app. You may also find WorkFlowy useful for organizing both documents, like an OCD, an also for thinking about code structure.
4. Here's some help - a "stackoverflow.com" thread on Building C# code

What you need to know:

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

1. The definition of the term **package** and have looked carefully at a few examples.
2. How to organize and prepare a technical document: OCD Study Guide