Project #5 – Source Code Repository Architecture Document

due Monday, Dec 9th

Version 2.3

Purpose:

This project develops an Operational Concept Document (OCD) for a Source Code Repository client and server. The purpose of the Repository is to enable insertion and extraction of source code files into a project's baseline and to display information about their properties and relationships. Authenticated check-in and versioning are required.

The Repository client displays information about Repository files in a Graphical User Interface (GUI) with several panes used to display some metadata properties and text of the current file and panes to show the child and parent dependencies. Clicking on a child or parent makes that file the current file. File caching on Repository and its clients is required to avoid unnecessary file transfers.

Context:

This Repository is one member of a server federation, illustrated in Figure 1., designed to support software development. Other members of the federation are a Test server, providing a test harness supporting continuous integration, and a Project server that stores and publishes project management information and supports collaborative communication through virtual meetings[1] and shared document resources.

All of the federated servers are designed as distributed virtual servers. This means that: they can be run in virtual environments, server contents can be distributed across several server instances, and that they provide a cloning facility that will allow any authenticated user to create an instance on a local desktop with a specified subset of the source server contents, and that new and modified contents can be checked into the source server. This allows local use of the development environment for initial creation and testing before inserting the new products into the project's baseline.

In Figure 1, below, we show the Federated Server context in which the Repository server must operate. We see that all the servers collaborate through message-passing communication to provide a powerful software development environment. The Repository's role is to manage a project's current baseline, e.g., the collection of its documentation and source code.

The Project server manages all of the data required to manage a large project, e.g., definition of work packages and their schedules and assigned responsible individuals, and means to support collaboration between project teams, some of which may be in remote locations. As an example, Figure 2 shows a hypothetical package structure for the Project server. This illustrates the level of detail needed to capture the concept for a large federated system. The Repository server will need a similar, though not identical, package structure.

The Test server supports continuous test and integration. Whenever a new package is inserted or an existing package has a new version checked into the Repository the Test Harness runs a sequence of tests against the modified subsystem or entire project to see if anything breaks. This implies that every check-in of new or modified source code should be accompanied by one or more tests to be added to a test sequence for the project baseline.

Figure 3 provides a class diagram for a major part of the Test server. This is a rare case where classes and class relationships help to clarify the system concept. Usually class diagrams are reserved for design documentation.

[1] This might be implemented by automating google+ hangouts.

Figure 1 - Software Development Collaboration System

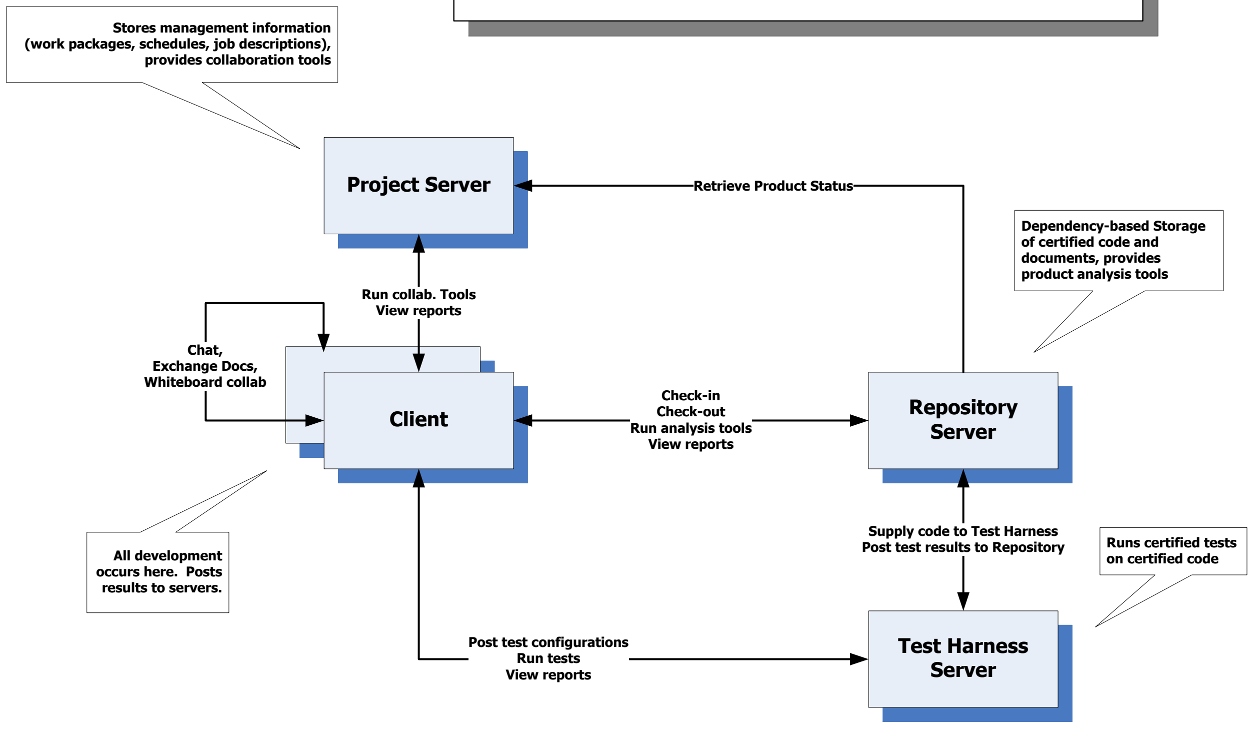


Figure 2 – Project Server Packages

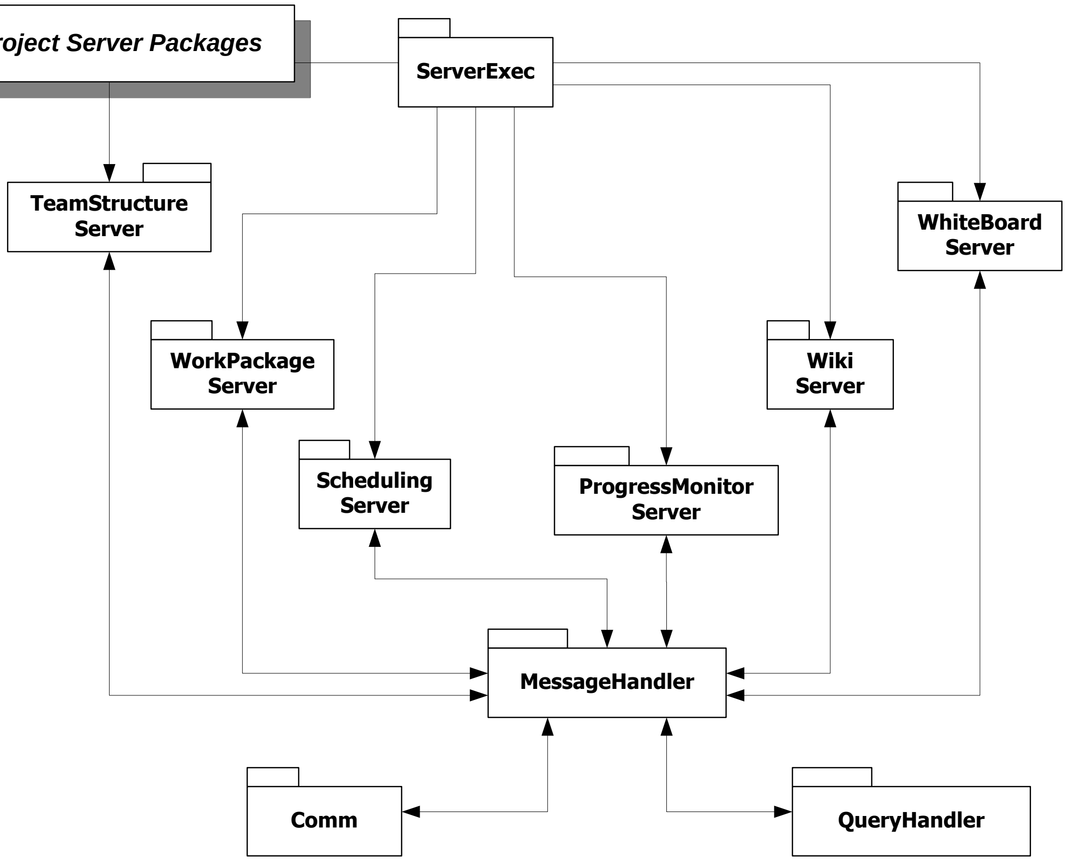
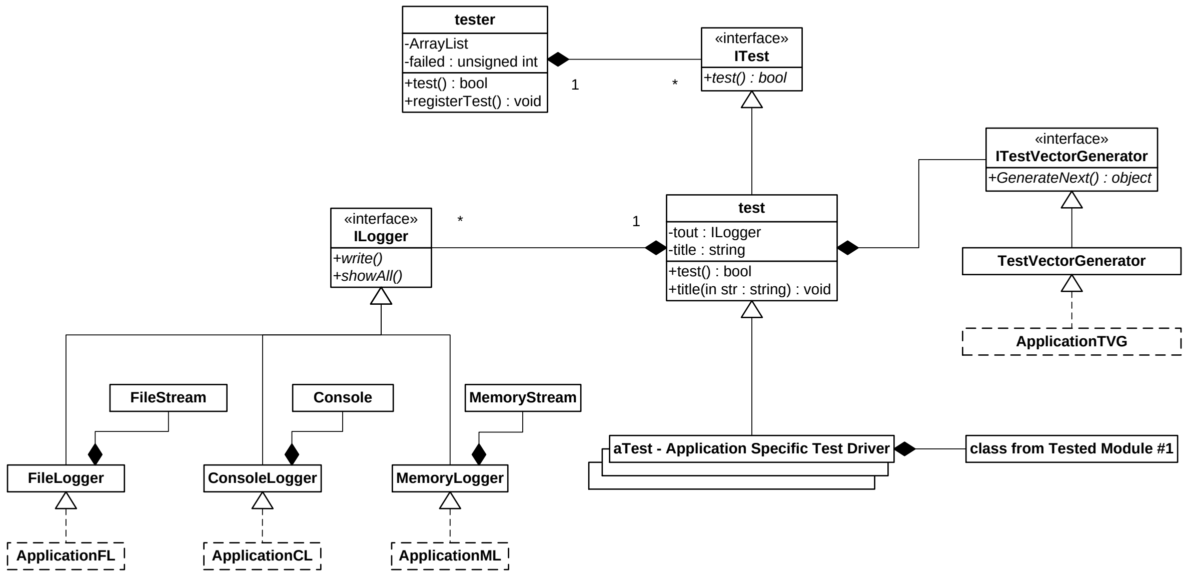


Figure 3 - Test Harness Structure



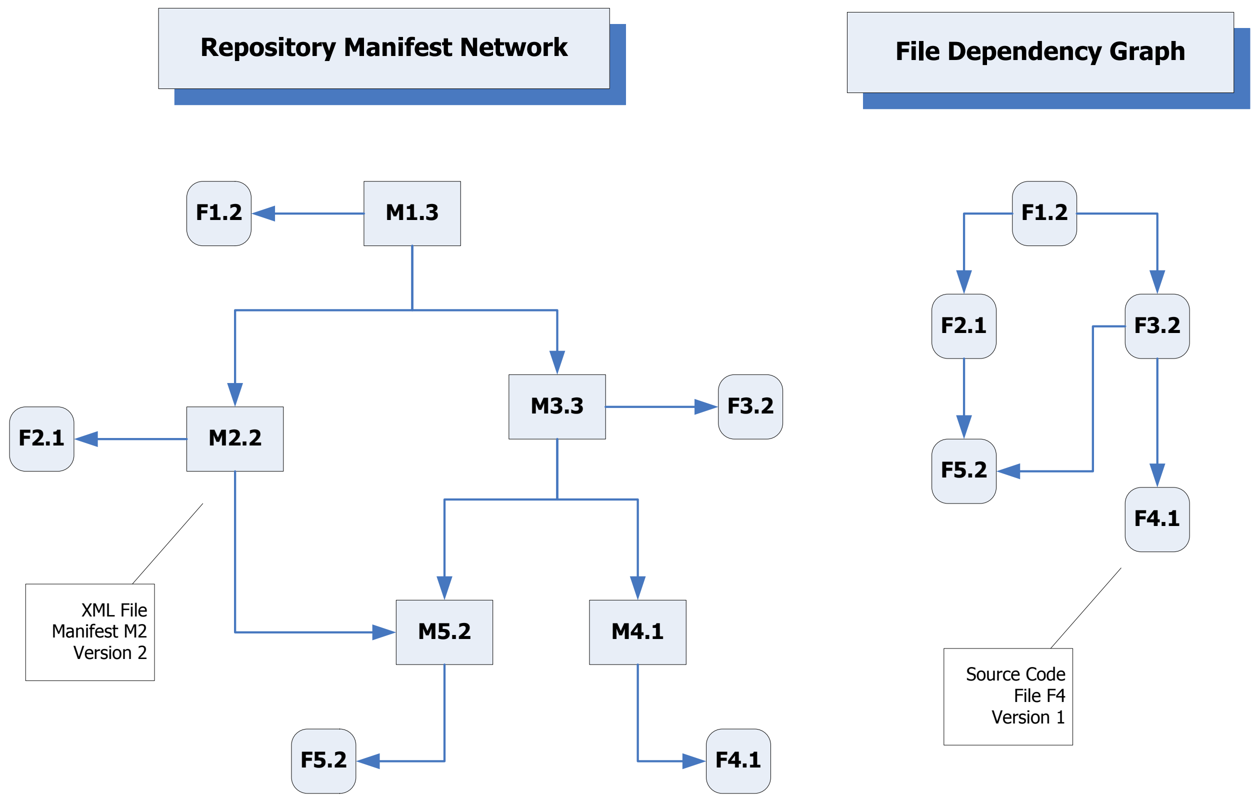
Repository Relationships:

Figure 4 illustrates how the Repository server manages dependency relationships between the resources it holds. Note that the metadata graph is virtual. It does not exist as an in-memory data structure. Instead all the graph relationships are embedded into metadata files. To traverse the graph we open each metadata file in turn to get navigation information, closing the file when we no longer need that information. Often that will be part of a recursive search. The file is opened when we first navigate to that node of the graph, and is closed when the recursion pops out of that node.

When a modified file is checked into the Repository a new version of its metadata file is created and added to the repository contents along with the modified file. These do not replace the previous versions. Any packages that link to the old version remained linked to the old version. When the owner of a parent package wishes to update to the new version, without changing the parent contents, a new version of the parent metadata is created that links to the old parent and has a dependency to the new version. This allows existing code that works to continue with their working components until it becomes convenient for the owner to update.

How will you make it convenient to support these relationships? What functionality must be provided by the client and the Repository so that human users find it very easy to use this facility?

Figure 4 – Repository Dependency Relationships



Repository Policies:

One reasonable policy for check-in is that dependencies must already exist in the repository. If we use that policy we will need to define an open check-in state that allows changing parts of the check-in without creating a new version. So, for example, if we which to check-in a package that has children not yet in the Repository we can do an open check-in and add the child packages whenever we are ready. When all of the dependencies are resolved, we close the check-in. You will need to think about policies that control what you can do with an open check-in.

In order to allow projects to craft their own policies, it would be useful to support rule-based policies, in much the same way that the Parser supports code analysis rules. One such policy concerns ownership. That is, who is allowed to check-in a particular package? To enable ownership management you will need to provide an owner element in file metadata. The same is true for open check-ins.

Requirements:

The Repository:

1. The Repository client **shall** support insertion and extraction of files and their metadata from a, possibly remote, Repository server using Windows Communication Foundation (WCF). Files **shall** be stored in the Repository in a suitable directory structure with metadata that indicates categories selected by the users. We will assume that each GUI has a unique directory because two or more GUIs may have files with the same name but different contents.

2. The Repository **shall** provide a separate tool that searches metadata files to build a map of parents for each file in the repository. This tool is run each time the Repository starts. When a file is inserted into the Repository its parent relationship with each of its children is recorded in the map.

3. The Repository Client **shall** be a WPF application that provides an information view with two information panes, one for metadata and one for file text. The client **shall** also provide two navigation panes, one showing the current file's children and one showing its parents. Clicking on either a child or parent changes the current file to the file represented by the clicked link.

4. The Repository Client **shall** provide a second file management view that supports local browsing for files to insert into the remote Repository and means for the user to provide description text and lists of child dependencies and keywords. The metadata tool, implemented in Project #2, **shall** be used to automatically build a metadata file from the information supplied here. This view **shall** also support editing metadata information for the current file.

5. The Repository **shall** accept text query messages[2] and return the fully qualified name[3] of every file containing that text in a clickable list. Clicking on any item in the list makes that file the current file in the client display.

6. The repository **shall** accept metadata query messages specifying metadata tags and return the contents of each element of the metadata for every file in the file set. This information is used by the Repository Client to build its display information. If a file does not have an associated metadata file, the query **shall** emit an error message and continue processing the remaining files.

7. Your project submission **shall** include two prototypes, one of which is a recursive search tool that uses the metadata graph, shown in Figure 4., to make Text and Metadata queries against a package and all its dependencies. You may select another topic that interests you for the second prototype.

8. Your submission **shall** supply a zip file containing the Project #5 Operational Concept Document and the solution for your project prototypes and all their files. It **shall** also include compile.bat and run.bat files that compile and run your project prototypes without the user needing to specify any information. Please make sure these work as expected when you extract the zip in an empty directory at any level in your computer's file system. In order for that to work you must use relative directory paths in your Repository and Client code.

Note:

You are not asked to implement the Repository virtual server nor to consider the Project and Test server concepts. You are asked to develop a concept for the Repository server and prototypes to illustrate the concept and explore its feasibility. There are many interesting questions on which this Project Statement has been nearly silent:

1. How are the virtual servers implemented?
2. How are rule-based policies implemented?
3. What kind of notifications and other communications are needed that the Repository must receive and process or emit?
4. How is file caching implemented?
5. What about security issues?

[2] A query message should contain one or more categories to filter the search.
[3] Returning the filename and category may be more effective than returning a physical server path.

6. How can ownership be configured to support change in a controlled but flexible and low-friction way?
7. How are server contents distributed across multiple servers?
8. How are queries routed across multiple servers?
9. What are the Repository packages? How do they communicate? What are their activities?
10. How will the Repository integrate into the server Federation? What facilities does it need to provide to support interactions with the Project and Test servers?
11. What facilities will the Project and Test servers need to provide so the Repository can carry out its mission?

There are many more questions to ask and explore. Part of your grade will be to identify interesting ideas and issues, some not discussed here, and explore them in moderate detail.