**Project #5:**

**Goals: (state top-level then interactive)**

1. Test Harness (tests new and changed certified code):
	1. Automated execution of versioned tests at the package, component, subsystem, and system levels, packaged into Test Suites.
	2. Persistent association of each test with the versions of the test driver, production code, TVGs, and Loggers used for the test.
	3. Persistent storage and retrieval of versioned tests, logs, and test results (uses repository), organized by Test Suite.
	4. Unit of testing is the Test Suite. Each Test Suite is executed on its own thread.
	5. Sends package RI notification of completion – green if all pass, otherwise red.
	6. Client may retrieve results and logs from Repository by clicking on notification.
	7. **Organizing Principle**: test suite if successful is rebuilt to become a library (dll) and all following tests are at that level, unless we change the suite contents.
		1. This implies that there is a single Suite test as well as individual package tests.
		2. So suite is an aggregation tool.
2. Repository (holds certified code):
	1. Persistently stores versioned, authenticated packages, libraries, test results, package publications.
	2. On check-in builds and publishes package. Check-in fails if these fail.
	3. Accepts Check-in Suite that automates check-in of each of the suite’s packages. If version of a package in suite exists in repository it is reported, not checked-in, and not failed.
	4. Retrieves systems, subsystems, components, and packages each with a single name. Uses dependency information to achieve one name retrieval
	5. Builds systems, subsystems, components, and packages when changes are checked in.
	6. Uses caching to avoid building existing libraries (two versions of a library are considered to be distinct).
3. Client (Initiates activities in Repository and Test Harness):
	1. Supports Suite of check-in of packages. Probably always the same as a Test Suite. Unit of check-in is package.
	2. Supports requests for and viewing of results of Repository tools execution on Repository contents.
	3. Supports query of test logs and test summaries from Repository.

**Flows: (interactive)**

1. Client -> Tester -> Repository -> Loader -> Test with TVG & Logger -> Repository -> notify client
	1. Client -> create Test Suite -> post to Test Harness -> Test Harness pulls files from Repository
	2. One logger is a test report generator all tests are required to use (along with others if they so choose) so test summary reporting will be consistent across all tests and searchable.
2. Check-in new -> analyze dependencies -> create manifest -> Checkin version 1
3. Check-in -> build -> test -> ( accept -> create new manifest version ) or reject -> increment version -> Publish
	1. Check-in root package (building upward) -> set links -> build -> test all Suite contents -> no changes to any linked component -> increment version -> publish
	2. Check-in non-root package (fixing, adding functionality) -> link downward as needed -> build -> test suite contents -> test all dependers -> increment version -> publish
4. Client -> Repository -> Publish -> source code & manifest text -> HTML, CSS -> Wiki (should this be automatic part of check-in ??)
5. Client -> Repository -> execute analysis tool -> persist result -> notify client

**Parts: (state top-level then interactive)**

1. **Test Harness**
	1. Test – interface and abstract class (possibly a default implementation ??)
	2. TVG & Loggers – interfaces and default implementations
		1. Dictionary that serializes to XML and de-serializes.
	3. Configurer/file manager
	4. Test Suite parser
	5. Cashe manager
	6. Communication
		1. Asynchronous Message passing contract
		2. File download contract
		3. Message builder and Parser
2. **Repository** (is a hierarchal database)
	1. File manager
	2. Check-in
		1. Builder
		2. Metadata creator, parser, scanner
		3. Version control (must work with Metadata)
		4. Hashing
		5. Publication (html pages for source code and documentation)
	3. Query
	4. Tools
		1. Dep anal
		2. Integrity analysis (compare hashed checksum with checksum)
		3. PrettyPrinting – merge code with metadata text and ???
	5. Communication
		1. Same as Test Harness
	6. Library Cache manager
	7. Navigator uses metadata
3. **Client**
	1. GUI
	2. Test Suite builder
	3. Communication
		1. Same as Test Harness
	4. Local tools
		1. Depanal
		2. Prettyprint
	5. Local Test Harness
	6. Local Repository

**Document Outline:**

1. Executive Summary – What you found, not what you did, ***results***, and ***conclusions***
2. Introduction – Continuous Integration and Test System, excerpts from Project #5 Statement.
3. Uses – Mgr, Dev, TL, QA, Customer. Their goals, tasks, design impact of meeting their goals. (Review Project Center uses)
4. Structure – CIT (steal diagrams), Test (Suite Hierarchy), Packages for client, repos, test harn, comm.
5. Critical Issues – continuous integration, test modes & state, taking results into information, performance

**OCD is not a collection of parts of documents from the web.**

**Midterm Makeup Part II – Project #5 Questions**

1. Write an outline for Project #5 OCD, e.g., all its main section titles and all its subsection titles. Try to have this make a coherent story.
2. Enumerate all the things you need to describe in a Test Suite. Write the XML for a sample Test Suite. Write LINQ code to a) build, and b) parse the Test Suite you defined.
3. A flow is an outline of an architectural thread, e.g., all the processing that is a consequence of some event. Write the flow for a check-in request of a Suite (the event) in which one component fails to build, one component builds but fails in test, and other components pass both build and test. By component we mean one or more packages that perform some specific function (like the parser, semi, and toker packages). You may find it convenient to answer this with three flows – one that succeeds, one that fails in test, and one that fails in build.
4. Draw an activity diagram for all the flow details described in the previous question.
5. Write a csc command that builds the parser package (which must have the lower level packages on which it depends) as a library. You test this in the Visual Studio command Prompt, e.g., Start button -> all programs -> Microsoft Visual Studio 2010 -> Visual Studio Tools -> Visual Studio Command Prompt (2010). Csc is the C# compiler and builder.
6. Write the code for a logger that stores log items in a dictionary and, on command, persists its contents to an XML file. This could be the required prototype you supply with the Project #5 OCD.
7. Write pseudo code for the creation and also for the parsing of the Repository’s package manifest (the XML file that holds a description of the package – like a manual page – and also holds links to its dependencies (by linking to their manifests).
8. Draw a class diagram for a message-passing communication service where all client messages are sent to the same BlockingQueue<Message> for processing on a single server thread. You may assume that the service uses either per-call or per-session activation.

**Bonus Prototypes**

1. Repository manifest builder and scanner with tests showing how it works.
2. Wizard to quickly build source code for test packages – inputs are name, name of production code package(s). Uses ITest and AbstractTest (see midterm solutions) and default TVG and Logger.
3. Test Suite builder that uses csc to compile and build test libraries. Required to include timing, using high resolution timer, and timed results in a graph that shows time to compile versus lines of code.
4. WPF Client GUI that mocks up client operation and illustrates how notification will work.
5. Communication system with a service contract that has asynchronous PostMessage and streaming file download operation contracts, e.g., pushes messages and pulls files.
6. Package authenticator that hashes a source file’s checksum, embeds it as a comment in the file, and authenticates by checking the hashed checksum with a computed checksum (after removing the comment). It is acceptable to substitute some other identifier for the checksum if you can think of a good one.
7. Write code for a version manager that accepts an unversioned file and converts it to a versioned file. The manager should also accept a versioned file and increment its version number. You may version by name or by directory. However, the version manager should also insert a version number comment at the top of each accepted source file. Examine every file generated for a a) C# console application, and b) a C# WPF application. How might you manage versioning of the additional files you find? Do you need to do that?