**CSE 687 Object Oriented Design Project**

**Test Harness**

***Purpose:***

Developing software consisting of multiple packages requires frequent testing. If the software has complex features, we want to build it incrementally. First design and implement a very basic core with a small number of packages, then add features one-at-a-time by adding new packages, or adding a few lines of code to an existing package. Each time we add new functionality, we build and test. That way, if additions break existing code, we know where to look, e.g., in the newly added few lines of code. A test harness allows us to use that incremental process efficiently.

A test harness should allow us to define and aggregate many small tests, each of which run with exception handling and results logging. Our goal is to do that without littering our code with many try-catch blocks and many logging statements. The technique we use to do that is to define, in the test harness, an execution function that accepts a callable test object (FN #1) - function pointer, functor, or lambda - and invokes the object within the scope of a try block. A catch clause displays exceptions and returns false to indicate test failure. Otherwise the executor returns the test predicate returned by the test object, e.g., true or false for pass or fail.

The test executor also provides results logging with predefined messages. Your test harness should define several levels of logging, e.g., just pass or fail, or more detailed, test-specific messages, or very detailed debugging messages, which display a time-date stamp and the values of key test-defined variables.

The test harness provides a mechanism to link any number of tests into a test sequence. Typically, during development, we keep adding tests for each new feature, and can rerun tests just for that feature, or all the tests of all the features, to ensure that the new addition didn't break existing functionality.

***Phases:***

You will be developing this project in four phases.

**Phase #1 – Architecture & Design**

**Phase #2 – Initial Development**

**Phase #3 – Functionality Growth**

**Phase #4 – Project Completion**

***Requirements:***

**Project Requirements**

1. The test system **shall** be written in the standard C++ programming language, using the standard C++ libraries.

**Test Harness Requirements**

1. The test system **shall** provide a Test Harness which accepts any callable object that requires no arguments (FN #2) and returns a boolean predicate to indicate success or failure.
2. The Test Harness **shall**  provide a multi-level logging mechanism, intended to show:
   * just pass-fail status
   * application specific messages for pass and fail, along with the result
   * detailed debugging output that includes level-two logging and a time-date stamp
3. The Test Harness **shall** invoke a passed callable object in the scope of a try block.
4. If exceptions are thrown during execution of the test, the catch block **shall** log test failure and the contents of the exception message.
5. The Test Harness **shall** accept a Test Request XML string defining one or more test elements by naming dynamic link libraries.
6. The Test Harness **shall** create, at startup, a number of child Tester processes. The child processes should post Ready messages to the Test Harness on startup.
7. The Test Harness **shall** define methods for processing Test Request messages and sending Test Status messages.
8. On receiving a Test Request, the Test Harness **shall** parse the request into individual tests and assign each test to a single child Tester.
9. Testers **shall** load the corresponding library, create an instance of the Test Driver, and log pass-fail status of each test, along with a time-date stamp.
10. The Test Harness **shall**, on completion of processing of a Test Request, send a named, time-date stamped, test status message to the Client requesting the test.

Note: You will find it convenient to have the Test Harness provide a Blocking Queue for Test Requests and a Blocking Queue for Ready messages. It uses a child thread to extract messages from its Comm using getMessage(), and enqueue the message in the appropriate queue. The main Test Harness thread dequeues these and sends the test request to the child tester that sent the Ready message (FN #3).

**Test Executive Requirements**

1. **The test executive shall include a graphical user interface (GUI) allowing the operator to define tests to be executed by the Test Harness.**
2. **The test executive shall** support building a Test Request by selecting library files from a repository using the GUI.
3. **The test executive shall** define methods for sending and processing messages.
4. **The test executive shall**  provide facilities to transfer a Test Request to the Test Harness. (See Messaging Requirements for message details.)
5. **The test executive shall** support viewing the test status on the GUI.

**Messaging Requirements**

1. The test system **shall** implement a message-passing communication channel using sockets. This can be based on the Channel prototype found in the class [Repository](https://ecs.syr.edu/faculty/fawcett/handouts/CSE687-OnLine/Repository/).
2. The messaging system **shall** provide:
   * Source and Destination addresses
   * Message type
   * Author
   * Time-Date
   * String body:  
     Expected to hold an XML string supplying information needed to execute a specific request.
3. Each communication end point (FN #4) **shall** provide both a sender and a receiver (FN #5).

**Testing Requirements**

1. Test Drivers **shall** implement a common interface that declares a test method taking no arguments and returning a boolean value indicating test pass or failure.
2. The test system **shall** provide a test case where several TestRequests are sent in quick succession to demonstrate that the process pool executes TestRequests concurrently and that each will, on completion, post a Ready status message to the TestHarness and await the next TestRequest message.
3. The test system **shall** provide a sequence of tests demonstrating all the features of your TestHarness.

***Foot Notes:***

1. A callable object is an object that can be invoked as if it were a function. These will be described in detail in several of the lectures.
2. You can provide arguments for the execution as member data of a functor, or as data captured by a lambda.
3. Messages have "to" and "from" End Point members.
4. A communication end point is a machine address concatenated with a port number, separated by a colon character.
5. You will find that the Comm helper code provides almost all of what you need.