Examination #3

Name:_____ **Instructor's Solution** _____ SUID:_____

This is a closed book examination. Please place all your books on the floor beside you. You may keep one page of notes on your desktop in addition to this exam package. All examinations will be collected promptly at the end of the class period. Please be prepared to quickly hand in your examination at that time.

If you have any questions, please do not leave your seat. Raise your hand and I will come to your desk to discuss your question. I will answer all questions about the meaning of the wording of any question. I may choose not to answer other questions.

You will find it helpful to review all questions before beginning. All questions are given equal weight for grading, but not all questions have the same difficulty. Therefore, it is very much to your advantage to answer first those questions you believe to be easiest.

1. Why are queues often used in multi-threaded code? What are the important queue semantics for use in applications that use more than one thread?

   Answer:

   Queues support sharing of messages without requiring a hand-off. The sender may enqueue the message at any time, and the receiver may dequeue the message at any other time after it is enqueued.

   Using a Thread-safe BlockingQueue<T> is the simplest way to communicate between threads that share the queue. No user locking is required because that is provided by the BlockingQueue<T> internally.

   Queues support asynchronous operations since the enqueuer simply posts its message to the queue and returns immediately, without waiting for a dequeuer. Multiple concurrent enqueuers each use the BlockingQueue<T> without regard to the activities of the others due to its internal locking.

   Important semantics of the .Net queue<T> are:
   - First-in, First-out behavior
   - No dependencies between enqueuer and dequeuer other than dequeuing happens after enqueuing
   - Queue content type specified by the application – Messages for Project #4.

   Additional important semantics of the BlockingQueue<T> are:
   - Queue will block on an attempt to dequeue an empty queue and will wake and dequeue after another thread has enqueued.
   - Thread safety for all queue operations provided by the queue using a Monitor and lock.

2. Write all the code needed to support concurrent searches for a string of text in files whose names are stored in a BlockingQueue<string>[1].

Answer:

```
public string textContents(string FQfileName)
{
  StreamReader sr = null;
  try
  {
    sr = new StreamReader(FQfileName);
  }
  catch { return ""; }
  return sr.ReadToEnd();
}

List<Task<bool>> tasks = new List<Task<bool>>();
int fileCount = fileNames.size();
string text = "foobar";

while (fileNames.size() > 0)
{
  string fileName = fileNames.deQ();
  Task<bool> t = Task.Run(
    () => {
      return demo.textContents(fileName).Contains(text);
    }
  );
  tasks.Add(t);
}

for(int i=0; i<fileCount; ++i)
{
  if(tasks[i].Result == true)
  {
    Console.Write("\n  found \"{0}\" in \"{1}\"", text, "file" + (i+1).ToString());
  }
}
Console.Write("\n\n");
}

// Pseudo Code:  See MT3Q2.cs for complete sol.  See MT3Q2b.cs for sol with Threads
//   foreach filename in queue
//     create task that
//       attempts to open file as text stream
//       if successful
//         read file to end as single string
//       otherwise
//         string is empty
//       task returns true if text occurs in file string
//     run task and add task to list of tasks
//   foreach task in list wait for completion then report true results
```

[1] If you don't remember the stream and string semantics needed for this processing you will get partial credit for writing pseudo code.

3. What is C# reflection. Write a code fragment that shows how you can extract type information from an instance of a C# class using reflection. You may choose the kinds of information to extract.

   Answer:

   Reflection is the process of programmatically extracting type information stored in an assembly's metadata when it is compiled.

```
MT3Q3 mt3Q3 = new MT3Q3();
Type t = mt3Q3.GetType();
Console.Write("\n  name of this class is {0}", t.Name);
Console.Write("\n  namesapce is {0}", t.Namespace);
Console.Write("\n  its methods are:");
MethodInfo[] mis = t.GetMethods(
  BindingFlags.Instance | BindingFlags.Public |
  BindingFlags.NonPublic | BindingFlags.Static
);
foreach(MethodInfo mi in mis)
{
  Console.Write("\n    {0}", mi.Name);
}
```

4. In the context of WCF, what is meant by an instancing policy and for what purposes is it used? What policy will you use for Project #4 and why?

   Answer:

   WCF Instancing policies describe the way instances of service objects are managed by the service host. Specifically, the policies describe lifetime management of these instances.

   There are three main policies:

   - PerCall policy creates a service object for each caller, running on its own thread, when a service call is made, and disposes the object at the end of the call.

   - PerSession policy creates a service object for each caller, running on its own thread, when a client makes its first call. The service object lives for a lifetime determined by its lease time. If the client makes another call before the lease expires, the lease is renewed. Otherwise the object is disposed. Note that PerSession requires a binding that supports sessions, e.g., WSHttp. BasicHttp does not support sessions.

   - Single policy creates a single service object for all callers. This object lives for a leased lifetime and any request by any client renews the lease. If no additional calls are made after the lease has started, then the object is disposed. Note that this implies that the service object must lock all its member data and static local data, since more than one client thread may access the service object concurrently.

   For Project #4, PerCall instancing is a good choice. The service objects don't have any unique state and so require little effort to instantiate and they have no dependencies from call to call. They simply deposit incoming messages in a shared blocking queue.

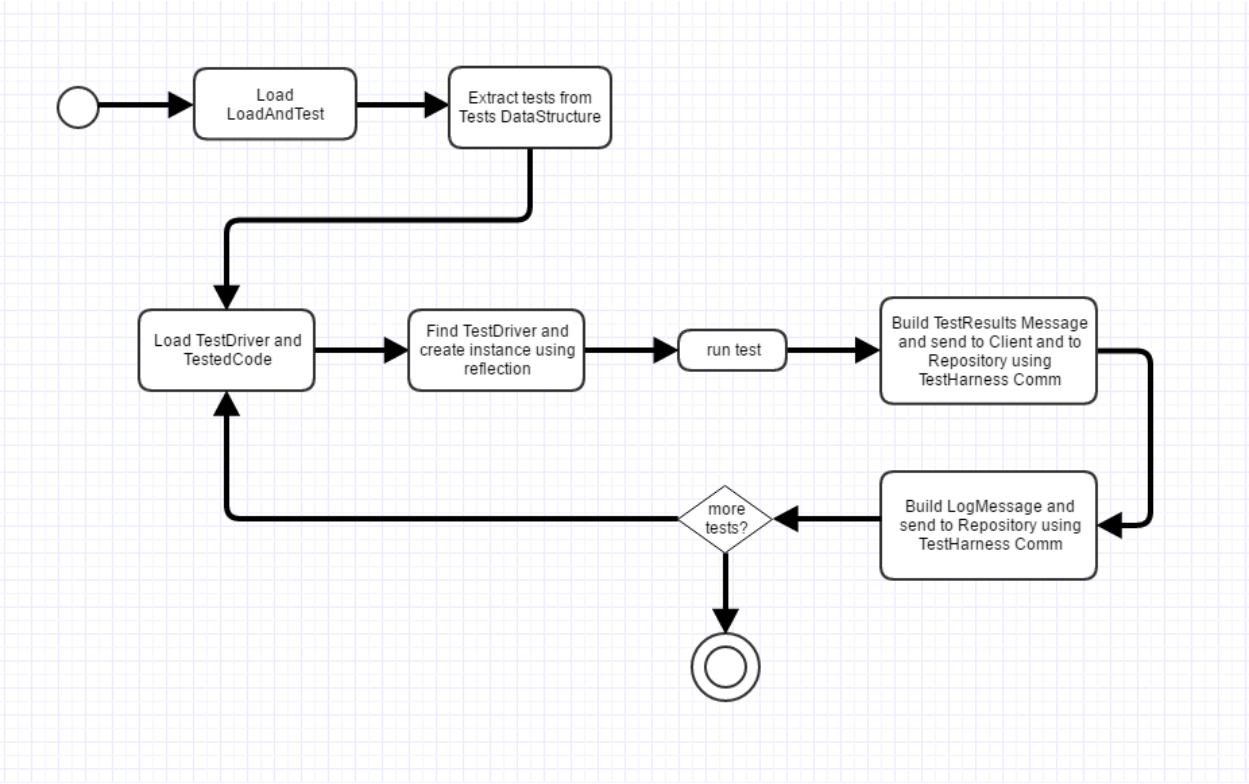
5. Write all the code for a receive thread that dequeues a message, extracts a string body, and causes that string to be written to a WPF window's listbox. Assume that the message was enqueued in a BlockingQueue<Message> instance.

   Answer:

```
delegate void NewMessage(string body);
event NewMessage OnNewMessage;

//----< receive thread processing >------------------------------

void ThreadProc()
{
  while (true)
  {
    // get deserialized message from receive queue
    rcvdMsg = recvr.GetMessage();

    // call window functions on UI thread
    this.Dispatcher.BeginInvoke(
      System.Windows.Threading.DispatcherPriority.Normal,
      OnNewMessage,
      rcvdMsg.body);
  }
}

//----< called by UI thread when dispatched from rcvThrd >-------

void OnNewMessageHandler(string msgBody)
{
  listBox2.Items.Insert(0, msgBody);
  if (listBox2.Items.Count > MaxMsgCount)
    listBox2.Items.RemoveAt(listBox2.Items.Count - 1);
}

//----< subscribe to new message events >-----------------------

public Window1()
{
  InitializeComponent();
  Title = "UI Demo";
  OnNewMessage += new NewMessage(OnNewMessageHandler);
  ConnectButton.IsEnabled = false;
  SendButton.IsEnabled = false;
}
```

6. Draw an activity diagram for all the processing you will implement in a child AppDomain in Project #4.

   Answer:



7. Enumerate all the important features of the C# object model, and say a few words about each. Please fit your answer on this page.

   Answer:

   The C# object model:
   - Defines two kinds of types: value types that live on the program's stack frame set and reference types that live in the program's managed heap.
   - Supports four class relationships:
     - Inheritance of reference types from other reference types
     - Aggregation of one or more reference types by another reference type
     - Composition of value types by a reference type
     - Using of reference and value types by a another reference type
   - Provides garbage collection and exception handling for all reference types.
   - Supports reflection operations using the Type class and getType method.
   - Provides access control through use of the key words: public, protected, and private
   - Supports releasing unmanaged resources using the Dispose method