Examination #2

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. The .Net System.Reflection namespace defines an abstract type Assembly that has a static method LoadFrom(fileSpec), which, if fileSpec identifies an assembly[1], loads it into the caller's process[2] and returns a reference to the loaded assembly. It also has a static method GetTypes which returns an array of Type objects. Each element of the array contains reflection information about a type defined in the assembly.

   ```
   abstract class Assembly {
     public static Assembly LoadFrom(string fileSpec);
     public static Type[] GetTypes()
       …
   }
   ```

   Write code to display all the type names and their namespaces found in the loaded assembly. Comment on using this idea to find file dependency information instead of the process described in class.

   Answer:

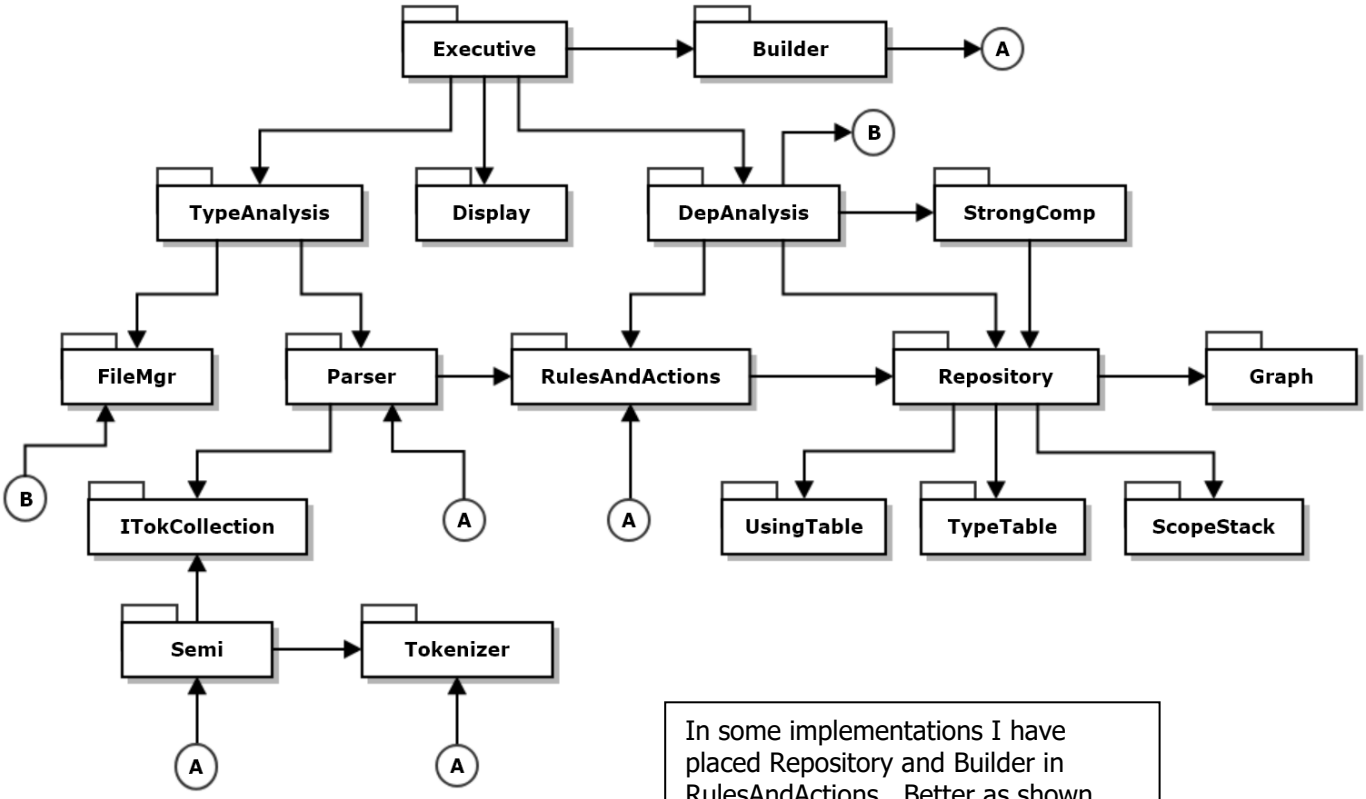
```
void displayInfo(string path)
{
  string[] files = Directory.GetFiles(path);
  foreach(string file in files)
  {
    string fileSpec = Path.GetFullPath(file);
    Console.Write("\n  {0}", fileSpec);
    string ext = Path.GetExtension(fileSpec);
    if (ext == ".exe" || ext == ".dll")
    {
      Assembly assem = Assembly.LoadFrom(fileSpec);
      foreach(Type t in assem.GetTypes())
      {
        Console.Write("\n    {0}.{1}", t.Namespace, t.Name);
      }
    }
  }
}
```

   The method above is certainly easier to apply than our parsing extravaganza. However, it only works for C#. No help for Java or C++. Our process works very well for those languages. Also, this method requires that we build code, so we can't analyze code for which we don't have a tool chain (compilers, linkers, …) installed. That is not required by our process for Project #2. Finally, this method can tell us the module where defined, but not the file.

[1] Assemblies have extensions ".exe" and ".dll".
[2] Technically, Loadfile loads the assembly into the calling Process'es default Application Domain. That factoid is not important for this question.

2. Draw a package diagram for your design of Project #3.

Answer:



In some implementations I have placed Repository and Builder in RulesAndActions. Better as shown with additional reposibilities of Pr3.

It may be reasonable to put the using and type tables in the Repository package.

3. Create a list of critical issues for Project #2, as you would for an Operational Concept Document (OCD). Please state the issue and describe how resolving that issue would affect your design.

   Answer:

   Critical issues are potential problems with the **concept**. They are issues that, if not addressed, could affect the usability of the product or the complexity of its implementation.

   - **Issue:**
     Handling many special cases in tokenizer processing, e.g., special tokens, comments, quoted strings could become complicated.
   - **Proposed Solution:**
     Since we are using the State Pattern, we will implement one state for each of the major speical cases.

   - **Issue:**
     Deciding on a next state depends on peeking into the token source stream. Often that requires looking at the next two available characters, but .Net streams do not support peeking at more than the first available character.
   - **Proposed Solution:**
     Build a TokenSource class that uses a .Net stream augmented with a character queue. That will enable peeking at multiple characters without removing them from the Token Source instance, e.g., we read as many characters we need out of the stream, but then enqueue them, so they stay in the source. When we need to pull characters from the stream, we pull first from the queue, until it is empty, then pull from the stream.

   - **Issue:**
     The parser will use the ItokenCollection interface to access semiExpressions. That avoids binding to concrete details of the Lexer. However, the parser rules need to access almost all of the Semi instance functionality.
   - **Proposed Solution:**
     Provide most of the public interface of Semi in the ItokenCollection interface. That will allow rules and actions to manipulate the token collections effectively.

   Many of the exams claim that scalability is a critical issue. It is not. The Lexer reads one token at a time and can do that for arbitrarily large files.

   Many claimed design or implementation failures are critical issues. They are not. Critical issues are concerned with potential problems with the concept (see above).

   Many claimed extensibility was a critical issue. It is not. The actions and rules are easy to amend to match the needs of different languages. I've used Lexer to analyze C++, C#, and XML, by amending the rules and actions, and, for XML, by subsitituting XmlParts for Semi. Since we use the ITokenCollection interface, that poses no problem. Just some small amounts of new design.

   Some exams cited design and implementation steps to meet requirements. These are just "business as usual" things to do, not critical issues.

   Some exams used a lot of words to say that their proposed solution is to provide a solution.

4. Write all the code to detect if a semiExpression has a specified sequence of tokens.

Answer:

Pass semi and token sequence array to function
  Set index of token sequence = 0
  Foreach token in semi
    Check for token at current index in sequence.
    If true increment index of token in sequence.
    Quit when all sequence tokens are found or reached end of semi.

Note: question does not ask for sequence of adjacent tokens.

Two alternate solutions shown below:

```
//----< uses variadic params construct >---------------------
/*
 * - Accepts a variable number of comma separated arguments
 */
public static bool hasSequence1(Semi semi, params string[] p)
{
  int index = 0;
  foreach (string token in semi)
  {
    if (token == p[index])    // p contains token sequence
    {
      if (index == p.Length - 1)
        return true;
      ++index;
    }
  }
  return false;
}

//----< uses array argument >--------------------------------

public static bool hasSequence2(Semi semi, string[] p)
{
  int index = 0;
  foreach (string token in semi)
  {
    if (token == p[index])
    {
      if (index == p.Length - 1)
        return true;
      ++index;
    }
  }
  return false;
}
```

5. What is a .Net delegate? Describe all the uses you can think of for delegates. For this description please provide an itemized list with one or more sentences per item that discusses that use.

Answer:

A delegate is a reference type that can bind to any function with a specified signature .Net code defines delegates using the syntax:

```
// define delegate type, derived from MultiCastDelegate

delegate ReturnType DelegateType ( T1 t1, T2, t2, … );

// declare instance of the delegate with a null invocation list

DelegateType dt = null;

// Subscribe to delegate

dt += new DelegateType(c.m);  // c is an instance of C with method ReturnType m(T2, T2, …)
```

Delegates are used for:

- **Event notifications:**
  Send a subscriber a notification of an event that occurred in a publisher's code.
- **Building publish and subscribe networks**:
  Define one or more publishers and multiple subscribers. Each publisher provides a public delegate property used to hold subscriptions, one for each event published.
  Each subscriber registers for one or more events with one or more of the publishers.
- **Binding to Lambdas**:
  A delegate instance can bind to a lambda like this:
  ```
  Action<T> act = (T t) => { /* do something with t */};
  ```
- **Binding processing to Threads**:
  A ThreadStart delegate can bind to a lambda or method and define a thread's activities like this:
  ```
  ThreadStart ts = new ThreadStart(x.fun);
  Thread t = new Thread(ts);
  t.start();
  ```
  where x is an instance of class X and fun is one of it's methods.
- **Dispatch Comm messages for processing**
  A dispatcher is a Dictionary with message command or id as key and a delegate as the value to invoke processing for a particular type of message.

6. What are your options for managing the life-time of remote objects when using Windows Communication Foundation (WCF)?

   Answer:

   WCF instancing mode is determined by identifying the mode you want in a ServiceBehavior attribute. Here's an example:
   [ServiceBehavior(InstanceContextMode=InstanceContextMode.PerCall]

   There are three modes:
   - **PerCall:**
     A remote object is created for each call, running on a thread associated with the caller. The object is enqueued for garbage collection at the end of the call.
   - **PerSession:**
     A remote object is created for the first call, running on a thread associated with the caller. The remote object lives for a lease period. If no calls occur during that period of time, the object is queued for disposal. Each call that occurs within the lease period renews the lease.
   - **Single:**
     Same as PerSession, except that all caller's threads access a single instance, so that instance must be made thread safe with appropriate locking.

7. Write all the code for a thread that continuously dispatches messages read from a blocking queue to one of three lambdas, using a msg.id() as the basis for selection. Write each of the three lambdas where one displays the msg using msg.show(), another announces that a message arrived, on the console, and the third exits the thread. Figuring out how to exit the thread is part of this question.

   Answer:

```
public class Dispatcher
{
  public Dictionary<int, Func<Message, bool>> storage { get; set; }
    = new Dictionary<int, Func<Message, bool>>();

  public bool dispatch(Message msg)
  {
    return storage[msg.id()].Invoke(msg);
  }
}
class MT2Q7
{
  SWTools.BlockingQueue<Message> bQ_ { get; set; } =
      new SWTools.BlockingQueue<Message>();
  Dispatcher dispatcher { get; set; } = new Dispatcher();

  void threadProc()
  {
    while(true)
    {
      Message msg = bQ_.deQ();
      if (dispatcher.dispatch(msg))
        break;
    }
    Console.Write("\n  -- exiting child thread");
  }
  // remaining code elided
}
// These lambdas are defined in elided Main function of MT2Q7

Func<Message, bool> lambda1 = (Message msg) => {
  Console.Write("\n  Message {0} has arrived", msg.name);
   msg.show(); return false;
};
mt2q7.dispatcher.storage[1] = lambda1;
Func<Message, bool> lambda2 = (Message msg) => {
  Console.Write("\n  Message {0} has arrived", msg.name); return false;
};
mt2q7.dispatcher.storage[2] = lambda2;
Func<Message, bool> lambda3 = (Message msg) => {
  Console.Write("\n  Message {0} has arrived", msg.name); return true;
};
mt2q7.dispatcher.storage[3] = lambda3;

Thread t = new Thread(mt2q7.threadProc);
t.Start();
```