

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. Suppose that dependency relationships found in Project #4 are stored in a dictionary like this:

Dictionary<string, List<string>> DependencyRelationships

where the key string parameter is the name of a package and the value List<string> is a list of all the packages that it depends on. Write all the code to invert those relationships, e.g., we create the same kind of data structure but the List<string> contains all the names of packages that depend upon the key package. This is useful for assessing the impact of changes in a package. It tells us which packages we must test to insure no breakage due to the change.

Answer:

```
using Dependencies = Dictionary<string, List<string>>;
```

```
class MT3Q1
```

```
{
    Dependencies deps = new Dependencies();

    Dependencies analyze(/* define fileSet here*/)
    {
        // simulate dependency analysis
        deps["Pkg1.cs"] = new List<string>();
        deps["Pkg1.cs"].Add("Pkg2.cs");
        deps["Pkg1.cs"].Add("Pkg3.cs");
        deps["Pkg2.cs"] = new List<string>();
        deps["Pkg2.cs"].Add("Pkg3.cs");
        deps["Pkg2.cs"].Add("Pkg4.cs");
        deps["Pkg3.cs"] = new List<string>();
        deps["Pkg3.cs"].Add("Pkg5.cs");
        deps["Pkg4.cs"] = new List<string>();
        deps["Pkg5.cs"] = new List<string>();
        return deps;
    }
    //---- Answer starts here -----
```

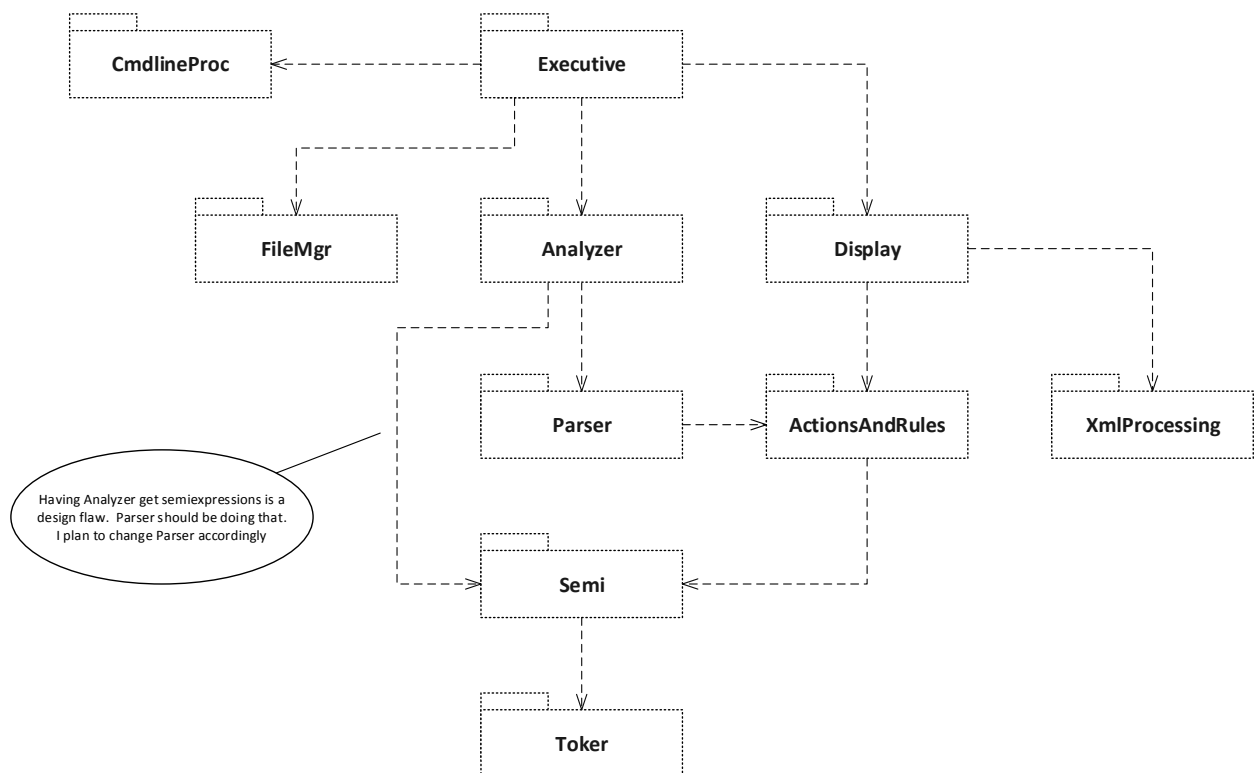
```
    Dependencies invert(Dependencies deps)
    {
        Dependencies inverted = new Dependencies();
        foreach (string key in deps.Keys)
            inverted[key] = new List<string>();
        foreach (string parentPkg in deps.Keys)
        {
            foreach (string childPkg in deps[parentPkg])
            {
                inverted[childPkg].Add(parentPkg);
            }
        }
        return inverted;
    }
    //---- Answer ends here -----
```

2. Draw a package diagram representing the code required for Project #2. Describe the interactions of each of the packages with the other packages in the diagram.

Answer:

Your grade depends on the accuracy of the package diagram, including call flows, and the clarity of your descriptions of interactions.

- Executive parses command line then creates instances of the primary types in FileMgr, Analyzer, and Display and starts them in left-to-right order.
- FileMgr returns a fileset specified by path and patterns passed to its search function.
- Executive loops over the fileset invoking analysis for each.
- Analyzer attaches the file passed by the executive and starts analysis by starting an instance of the Parser. It then collects semiexpressions, semi, from the attached file and calls Parser.parse(semi) with each one in turn.
- Parser.parse(semi) passes semi to its rules, defined in Actions and Rules.
- Semi retrieves tokens from Toker and decides when its sequence is complete.
- Each rule in ActionsAndRules defines a detector for a particular grammatical construct. When the rule matches a passed semiexpression it fires its actions. Each action uses the semiexpression passed to it to extract information and save to a Repository, also part of ActionsAndRules.
- Finally Display retrieves analysis data from the Repository, filters and formats into useful information and displays on the user's console and writes XML of results.



3. Assume that the processing needed to invert type relationships is wrapped in a function:

```
Dictionary<string, List<string>> invert(Dictionary<string, List<string>> TypeRel) { ... }
```

Write the code to run that processing on a threadpool thread. Show how you will eventually retrieve the result.

Answer:

Note:

I meant invert **dependency** relationships for Dictionary<string, List<string>> **DepRel**, but that isn't what I asked above.

That discrepancy doesn't affect what you were asked to do, e.g., run a function on a threadpool thread. I show here how to do that. See MTF14-Code -> MT3Q3a for the complete code. If you want to see how to invert a type relationship then look at MTF14-Code -> MT3Q3b.cs

```
static void Main(string[] args)
{
    "MT3Q3 - Dependency Inversion on Threadpool Thread".Title();
    MT3Q3 q1 = new MT3Q3();
    Console.WriteLine("\n show parent-child relationships");
    Dependencies deps = q1.analyze();
    deps.ShowDeps();
    Console.WriteLine("\n show child-parent relationships");
    Task<Dependencies> t = Task<Dependencies>.Run(() => q1.invert(deps));

    // do other work here

    t.Result.ShowDeps();

    // Note: we could also display the results in a continuation

    Console.WriteLine("\n");
}
```

4. Describe the syntax and semantics of passing a reference type by reference to a WCF proxy. A proxy is an object, created for WCF communication, with the interface of a remote service object.

Answer:

The syntax is that same as for a call to a local object. The proxy method uses a ref qualifier for the passed parameter declaration and the client uses a ref qualifier for the method invocation. Syntax looks something like this:

```
IChannel proxy = CreateServiceChannel(...);  
RefType rt = new RefType();  
proxy.SomeMethod(ref rt); // Service interface declaration for SomeMethod uses ref too
```

Surprisingly the semantics are the same as well. Thus, if the remote object changes the state of the referenced object in its managed heap, the client sees the same change in state of a local object in the client's managed heap. If the remote object replaces the referenced object with another of the same type by calling the new operator, then the client will also see a new object with the same state as the remote object's new referenced object.

For an example of this behavior look at the SelfHosted_StringsService in
Handouts/CoreTechnologies/SocketsAndRemoting/code/WCF_Fawcett_Examples.

The implications of this are that the channel will match on the client's end anything that happens to the passed reference to the object on the server. I was very surprised the first time I observed this behavior.

5. When you use WCF to establish communication for Project #4 you will have to choose an instancing model. Please describe the types of instancing models and why you will pick one specific model for your project.

Answer:

The instancing models are:

- PerCall:
For each call creates an instance of the remote object running on a thread dedicated to the caller. So multiple concurrent clients each get their own remote object instance running on a thread dedicated to that one instance. When the call completes the remote instance becomes available for garbage collection.
- PerSession:
For each caller an instance of the remote object is created on the first call, running on a dedicated thread. This instance lives for a leased lifetime. If the client makes another call within the lease time the lease is renewed. If no call comes in after the first within the lease time then the object is released for garbage collection.
- Singleton:
A singleton instance serves multiple concurrent clients each running their own threads. Thus the singleton must provide for sequential access to shared resources by locking. The Singleton instance lives for a leased lifetime, with the same behavior as for PerSession instancing.

You choose PerCall if calls are infrequent and/or the remote object is cheap to construct. Often the remote object has no state and so is cheap to create and destroy.

Choose PerSession if you need to save state in the remote object between calls. Also, use this method of instancing if the remote object is expensive to create.

Use Singleton to share data between multiple, possibly concurrent, clients. Be sure to provide for exclusive access to any shared data.

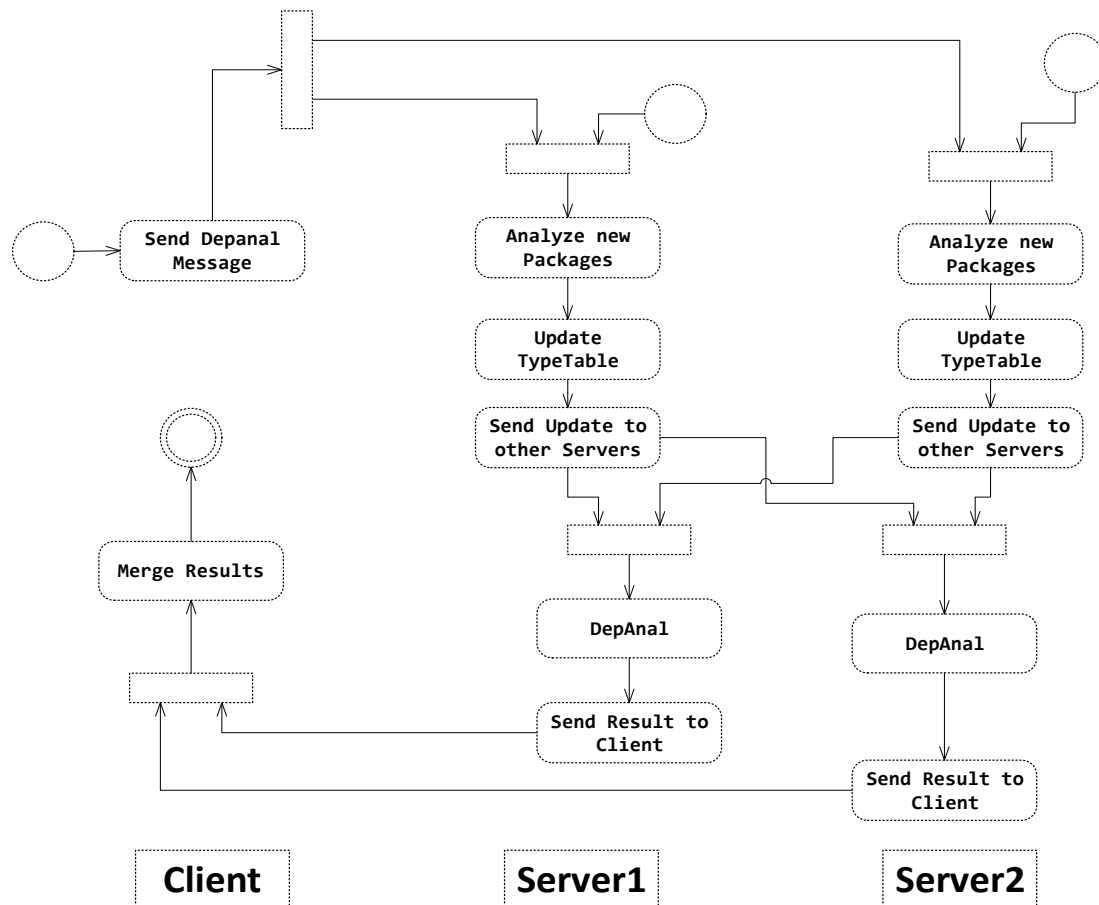
6. When a new package is added to a Project #4 server describe all of the processing necessary to update that server's type table and ensure that the type tables on a known set of other servers are also updated to reflect the newly introduced types. Please support this discussion with an appropriate diagram.

Answer:

To update a server's TypeTable the server needs to analyze newly added files for type information and add to the TypeTable. If the server does not keep track of when TypeTable updates occur relative to the addition of new files, then the server will have to analyze all of its packages for type information and reconstruct its TypeTable.

Adding new type information to a TypeTable is easy if the TypeTable has been designed well. See for example MT4Q1. Once the TypeTable has been updated the new type information needs to be sent to all the other active servers, perhaps as a TypeTable shard (the rows of new information).

The update process is shown in the Figure below. Note that any analysis for type or package dependencies need to wait for TypeTable inputs from all servers. Also, the analyses should run on the servers, not the client, as we don't want to pass around a lot of files. The diagram assumes that package analysis is lazy. That is, it doesn't happen until a client makes an analysis request. Then all unanalyzed packages are processed.



7. What are the most significant responsibilities of the .Net XDocument class?

Answer:

The XDocument class, defined in System.Xml.Linq, is a container for XElement objects describing all the parts of some XML document.

The XDocument class provides facilities to:

- Load and parse an XML document, building an internal parse tree of XElement objects which describe the various elements of the document.
- Build a new document or edit an existing document by adding or deleting XElement objects from parent XElement objects.
- Save the document to a file using XDocument.Save(string filename).
- Emit a string with the document's XML using the XDocument.ToString() method.
- XDocument extension methods return IEnumerable results to support LINQ queries.

Note that XDocument treats its XDeclaration differently from every other XElement. It holds the declaration in the XDocument.XDeclaration property and does not include this in the ToString() output.