Proxy Pattern

Estimated time for completion: 60 minutes

Overview:

This lab is broken into two parts, first part you will build a proxy by hand and in the second part make use of the transparent proxy support offered by .NET to automatically build a proxy for you.

Goals:

- Use the Proxy Pattern not just for network remoting, but for providing initialization of expensive objects on demand
- Use of Transparent proxy to layer in functionality onto any type derived from MarshalByRef

Lab Notes:

Part 1: Create a virtual proxy to load a list on demand

Your first task is to implement a new type of list that is only populated when the list is first accessed, but still looks like a normal list as far as the client is concerned. The fact that the data is loaded on demand will be something the client of the list is unaware of. This will result in keeping the main application logic simple whilst only taking the resource hit when the data is first required. In order to implement this solution you will use the Proxy Pattern.

1. Open the initial solution, familiarize yourself with the code. The application simply allows the user to print out a set of even numbers, primes, Fibonacci numbers or a set of primes and Fibonacci numbers. Creating some of the sets of numbers is expensive so we only want to create them once, so the code has been structured to simply create all the sets of numbers at startup, all the code then references these lists. Whilst this works in only creating these sets once it does produce a long startup time calculating all the sets of numbers and the user may not need all the sets. What would be better is to create the lists on demand.
2. You could change the main application logic so before using a set check if the reference is null, and if so create a new instance. But that smells on two fronts one we are having to modify every location where the list is referenced to check for null, two we are also going to have to embedded at this point the knowledge of how to create the list. What would be more optimal would be to pass a proxy to the list, such that when the proxy is first accessed it then goes through the expense of creating the object for real, and from then on it simply delegates all calls to the list it has just created.
3. What you will do instead is to build a virtual proxy, a type that looks like a list but its implementation will on first call create an instance of the list. In order to use the proxy pattern the proxy and the subject must share a common interface, unfortunately in the case of List<T> there is no common interface that represents all the functionality of the

type, (The Java classes have an abstract List allowing different implementations of a List to exist). The closest we can get is IList<T> but this falls short of all the operations you can do on a List<T> but for our example it will suffice.

4. Note there is a work around to this by creating your own Collections type hierarchy and using Adapters to make use of the .NET implementations.
5. So the first step is to refactor inside main not to use List<T> but to use IList<T>, and to make each of the CreateXXXX methods return something of type IList<T>, we now have a solution were the code is now largely written to interface as opposed to concrete implementation of the List. Compile the solution and make sure it still works.
6. You will now create the proxy, create a new class called VirtualList<T>. Since its a proxy for any kind of object that implements IList<T> then it must implement the IList<T> interface. Generate stubs foreach of the methods (Tip after having typed class VirtualList<T> : IList<T> a smart tag will appear under the interface hit CTRL+ENTER to get the smart tag menu and select the first item implement interface)
7. You now have a class that can be used anywhere where IList is used. The goal of the virtual list is to create the actual list the first time the list is accessed, so each of the methods in your proxy you will need to decide if the list has been created and if not create it. Then call the actual implementation of the list. To do this you need to hold onto any list created inside the proxy, you also need to a way of deciding how to create the list. The first part is easy add a field to the VirtualList class to hold onto the created list, call it subject and it should be of type IList<T>. The second part is a bit more tricky, as the VirtualList is for any type of list it doesn't know the concrete instance of the List to use nor does it know how to populate the list, we somehow need to pass that knowledge in at runtime (Smells of a pattern...Strategy anyone).
8. To pass in the creation behaviour of the list you will need to define a delegate type that can return an object of type IList<T>. Add a constructor to the VirtualList<T> type that takes an instance of the delegate as a parameter and stores it in a field of the same type, to be used later when a call is made to any of its list methods.

```
public delegate IList<T> CreateAndPopulateList<T>();

public class VirtualList<T> : IList<T>
{
private IList<T> list;
private CreateAndPopulateList<T> populateMethod;

public VirtualList(CreateAndPopulateList<T> populateMethod)
{
    this.populateMethod = populateMethod;
}

...
}
```

9. Inside each of the IList methods implemented inside the VirtualList you need to check if list field is equal to null and if so call the supplied list creation behavior. Writing this for each method is tedious so create a single method called EnsureListLoaded and place the

relevant code in here and then simply place a call to this method as the first task in each List method.

```
public  int IndexOf(T item)
{
   EnsureListLoaded();

   return list.IndexOf(item);
}

...

private void EnsureListLoaded()
{
   if (list == null)
   {
      list = populateMethod();
   }
}
```

10. Check everything still compiles.
11. Now it's time to make the necessary changes inside Main, simply replace all the initial lines that make calls to CreateXXXX, with code that creates instances of VirtualList<T> were the strategy for the creation is the method that was previously called to create the list.

```
IList<long> primes = new VirtualList<long>(CreateListOfPrimes);
            IList<long> evens = new VirtualList<long>(CreateEvens);
            IList<long>fibs = new VirtualList<long>(CreateFibs);
```

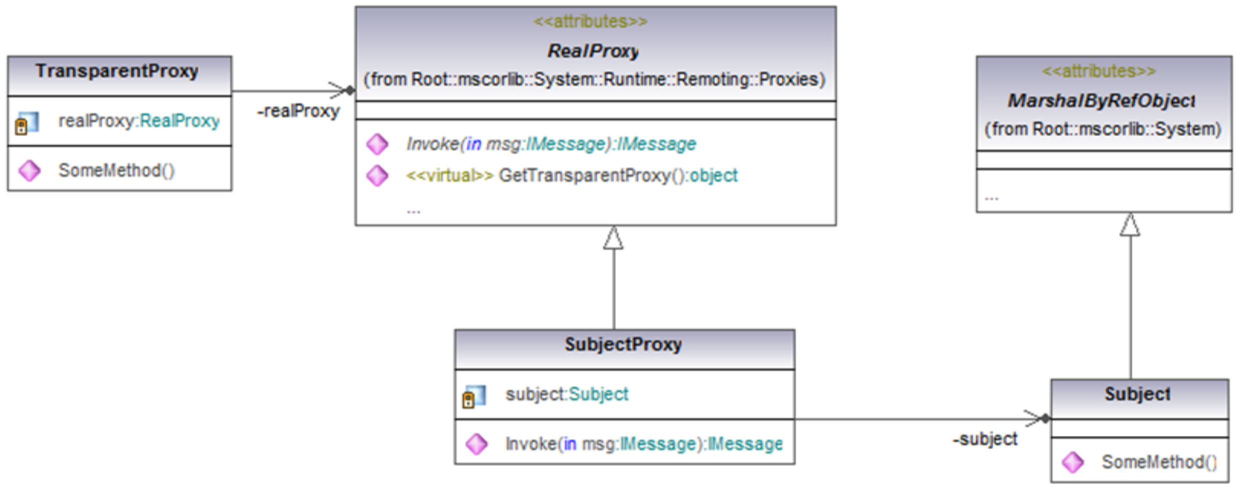
12. Now you have a solution that when run, immediately interacts with the user. When the user wants to use prime numbers then and only then do they take the hit. Multiple uses of prime numbers still only means the hit is taken once and we have not modified the core application logic with if null style code, nor had to change the implementation of the list.

Part 2: Using Transparent Proxies

In this part you will use the transparent proxy functionality in .NET to create a general purpose proxy. In this case it will be a performance counter proxy. The idea being that you wish to record the number of times a method has been called, useful fo r performing diagnostics. Windows has a standard way for applications to publish these kind of statistics through the use of performance monitor counters, perfmon is a tool that can be run to visualize these counters whilst the application is running.

The application you will be adding perfmon counters too is a model of a simple bank.   The bank is concerned to see as soon as possible when there is a run on the bank, for that reason you have been tasked with adding the necessary functionality so that during the day bank officials can see the rate of bank transfers into and out of accounts.   Run the client application and you will see a serious of - + and = signs appearing in the console, this represents the activity in the bank. Hit *the R key to simulate a run on the bank and you will see loads of - signs go wizzing by...press C when calm has been restored to the market.*

1. Load the solution (Monitoring.sln) and familiarize yourself with the project you will notice that there is a BankAccount assembly this contains a set of types that represent a bank account system. A Client assembly that contains some code to exercise the bank account system, and lastly an assembly containing a facade around the performance monitor counters api.
2. When part of the system wishes to perform an operation on an account, it requests the account object from the Bank object, this is therefore an opportunity to wrap the account object with a transparent proxy and return the transparent proxy (TP) back to the client. In order for the BankAccount class to be invoked via the proxy it must derive from MarshalByRefObject modify the BankAccount class to derive from MBRO
3. In order to create a transparent proxy you need to make a call to GetTransparentProxy against an instance of a RealProxy. The RealProxy class has an abstract method on it called Invoke, each time a call is made via a Transparent proxy it gets routed to the invoke method of the Real Proxy that created it. Thus the RealProxy allows you to decide how to deal with the request, before and after invoking the functionality on the subject.



TransparentProxy
realProxy:RealProxy
SomeMethod()
-realProxy
<<attributes>>
RealProxy
(from Root::mscorlib::System::Runtime::Remoting::Proxies)
Invoke(in msg:IMessage):IMessage
<<virtual>> GetTransparentProxy():object
...
<<attributes>>
MarshalByRefObject
(from Root::mscorlib::System)
...
SubjectProxy
subject:Subject
Invoke(in msg:IMessage):IMessage
-subject
Subject
SomeMethod()

4. Create your own RealProxy that will be responsible for updating performance monitor counters, call it MethodCounterProxy and create it in the Monitoring Assembly. Unlike the VirtualList Proxy in this case have a constructor that will take a reference to the object the Proxy will wrap up, and store it in a instance field called subject, the Real proxy base class constructor also needs calling with the type of object it is expected to produce the TP for. Finally override the Invoke method, and implement a simple stub, check everything compiles

```
public class MethodCounterProxy : RealProxy
{
   private MarshalByRefObject subject;

   public MethodCounterProxy(MarshalByRefObject instance) :
   base(instance.GetType())
   {
      this.subject = instance;
   }

   public override IMessage Invoke(System.Runtime.Remoting.Messaging.IMessage
   msg)
   {
      return null;
   }
}
```

5. Now implement the code inside the invoke method to actually call the app, one way of doing this is to use the RemotingServices class to Execute the message you are passed via the Invoke call. The RemotingServices.ExecuteMessage() method takes a reference to the subject and an object of type IMethodCallMessage the parameter you receive via the Invoke method call is an object of this type but you will have to make a speculative down cast to get it into that type, so create a local variable of type IMethodCallMessage and assign it the value of msg as supplied via the Invoke call, and make the necessary call to RemotingServices.ExecuteMessage()

```
public override IMessage Invoke(System.Runtime.Remoting.Messaging.IMessage
   msg)
{
   IMethodCallMessage methodCall = (IMethodCallMessage)msg;

   IMessage result = RemotingServices.ExecuteMessage(subject, methodCall);

   return result;
}
```

6. You have now created a simple pass through proxy.
7. Now you have the simple proxy, modify the GetAccount method inside the Bank class to not simply return a BankAccount object but to return the Transparent proxy. To do this create an instance of your Real Proxy and then call the GetTransparentProxy method in order to create the TP, and return the TP

```
public Account GetAccount(int accountNumber)
{
   Account account = (Account) new
   MethodCounterProxy(accounts[accountNumber]).GetTransparentProxy();

   return account;
}
```

8. Rerun the code it should continue to work as before
9. Now for the fun part plugging in the performance monitor counters. Performance monitor counters are organised into categories, the MethodCounters type supplied in this solution encapsulates the performance monitor counters, by creating a performance monitor category for each given .NET type it is requested to monitor. Each method call inside the given type is then represented as two counters a total number of calls and a number of

calls per second. To get to an instance of MethodCounters, call the static method on the class called GetMethodCounters supplying the type you wish to monitor, this will return an object of type MethodCounters that can be used to record each method call. Modify the constructor of your real proxy class to get an instance of the MethodCounters and store it in a private field

```
public class MethodCounterProxy : RealProxy
{
   private MarshalByRefObject subject;
   private MethodCounters counters;

   public MethodCounterProxy(MarshalByRefObject instance) :
   base(instance.GetType())
   {
      this.subject = instance;
      counters = MethodCounters.GetMethodCounters(instance.GetType());
   }
...
```

10. Now modify the Invoke method to call the MethodInvoked method on the MethodCounters object, supplying it the name of the method called, this can be found from the property MethodName on the methodCall object.

```
public override IMessage Invoke(System.Runtime.Remoting.Messaging.IMessage
   msg)
{
   IMethodCallMessage methodCall = (IMethodCallMessage)msg;

   IMessage result = RemotingServices.ExecuteMessage(subject, methodCall);

   counters.MethodInvoked(methodCall.MethodName);

   return result;
}
```

11. Now re run the code without the debugger attached using CTRL-F5 or disable the use of vshost via the project properties. Run perfmon
    a. Remove all perfmon counters currently displayed
    b. Add new counters, in the category section you will find an entry called TheBank.Account, select this, the list of counters should then be displayed two for each method call, select the ones labeled Credit/s, Debit/s and Balance/s, and in the instance selection select client.exe, and then hit add. You will then see three lines bouncing up and down representing the number of method calls per second, to make it more interesting cause a run on the back using the R key inside the client console window you will see a sharp rise in the number of Debit/s per second, press C to calm things down again.
12. What you have built is a general purpose method performance monitor proxy that can be used with any type as long as it derives from MarshalByRefObject. However it is a little debatable if this is truly a proxy since whilst it uses Transparent Proxy technology it

really smells more like a decorator, in fact the distinction between decorator and proxy can be very hard at times, and it typically comes down to intent.