Урок 3

1. Управление поведением объекта службы

У вас уже должно сформироваться представление о том, что такое WCF-служба. На сегодняшнем занятии вы узнаете новые полезные черты WCF-технологии. Вы видели, что использование атрибутов применяется в WCF достаточно широко. Указывая в нашем коде тот или иной атрибут, мы просто включаем в наше приложение определенный код, ассоциированный с этим атрибутом, избавляя себя от необходимости писать такой код самостоятельно. Кроме того, вы уже видели, что сами атрибуты могут иметь собственные модификаторы. На этом занятии мы снова будем использовать атрибуты с модификаторами.

Возникал ли у вас вопрос, как ведет себя служба при обращении к ней клиента? А что происходит, когда к службе обращаются сразу десять клиентов? Создается ли свой экземпляр службы для каждого клиента, или все клиенты работают с одним экземпляром? Можно ли управлять поведением службы? Попробуем разобраться с этими вопросами.

Давайте подробнее поговорим о самом понятии «поведение». В WCF поведениями называют некоторые классы, которые позволяют изменять работу службы или клиента на разных этапах жизненного цикла приложения. Выделяют три основных типа поведений: поведения службы (ServiceBehavior, ContractBehavior), поведения операций (OperationBehavior) и поведения оконечной точки. Задавать требуемые поведения можно разными способами.

1. С помощью атрибутов можно задавать поведения служб и операций, указывая атрибуты либо для класса службы, либо для операции.

например:
атрибут ServiceBehavior у класса службы задает поведение службы:

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Single,
InstanceContextMode = InstanceContextMode.PerSession,)]
public class BankService : IBankService
{

}
```

атрибут OperationBehavior у операции службы задает поведение операции:

```
[OperationBehavior(TransactionScopeRequired = true,
TransactionAutoComplete = true)]
public void ToDeposit(double sum)
{

}
```

Однако отметьте, что атрибутами нельзя задавать поведения оконечных точек;

2. Поведения можно задавать в коде - у всех объектов, описывающих службы, оконечные точки, контракты и операции есть свойство Behaviors – коллекция, в которую можно добавлять требуемые поведения;

например:

```
ServiceHost hs = new ServiceHost(typeof(MyService));
ServiceEndpoint ep = hs.AddServiceEndpoint(...);
WebHttpBehavior behavior = new WebHttpBehavior();
ep.Behaviors.Add(behavior);
```

3. для служб и оконечных точек поведения можно задавать в конфигурационном файле в секции <serviceModel><behaviors>

например:

```
<behaviors>
  <serviceBehaviors>
    <behavior name="throttling">
      <serviceThrottling maxConcurrentInstances="5"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Разные способы использования поведений мы рассмотрим ниже. Какое поведение применять – зависит от конкретной задачи.

a. Свойство InstanceContextMode

WCF предоставляет в наше распоряжение механизм, позволяющий контролировать процесс создания объектов службы. Для управления поведением объекта службы используется атрибут ServiceBehavior, который указывается для класса службы. У атрибута ServiceBehavior есть свойство InstanceContextMode, которое и позволяет

управлять поведением объекта службы. Три допустимых значения этого свойства определены в перечислении InstanceContextMode:

- InstanceContextMode.PerCall означает, что для каждого сообщения, направленного службе, создается новый экземпляр службы;
- InstanceContextMode.Single означает, что создается один единственный экземпляр службы, и все сообщения направляются ему, независимо от количества приходящих сообщений и количества клиентов;
- InstanceContextMode.PerSession означает, что для каждого клиента создается свой экземпляр службы, и все сообщения от этого клиента поступают к его «собственному» экземпляру службы.

Надо отметить, что значения Single и PerSession не позволяют использовать привязку basicHttpBinding.

Если вы планируете, что с вашей службой будут работать несколько клиентов одновременно, то вам будет полезно еще одно свойство атрибута ServiceBehavior – ConcurrencyMode. Это свойство отвечает за возможность службы запускать новую операцию, если еще не завершена текущая операция. ConcurrencyMode также имеет три допустимых значения, определенных в перечислении ConcurrencyMode:

- ConcurrencyMode.Single разрешает доступ к экземпляру службы только из одного потока, поэтому новая операция не может быть начата, пока не завершена текущая;
- ConcurrencyMode.Reentrant разрешает доступ к экземпляру службы из другого потока, если это callback контракт; новая операция может быть начата, если запрос на нее поступил по контракту обратного вызова;
- ConcurrencyMode.Multiple разрешает доступ к экземпляру службы из нескольких потоков одновременно, т.е. новые операции могут стартовать независимо от завершения текущей операции.

b. Пример использования свойства InstanceContextMode

Чтобы проиллюстрировать все вышесказанное, создадим еще одну пару «служба-клиент». Это будет «Банковское приложение». Наше приложение будет создавать счета для обратившихся к нему клиентов, переводить на эти счета деньги клиентов и отображать текущий баланс счета. При этом счета будут просто нумероваться в очередности создания. Больше ничего наш «банк» делать не будет. Ни проценты по депозиту начислять, ни позволять клиентам снимать деньги со своих счетов ☺. Начнем с серверной части приложения. Для упрощения примера наша служба снова будет хоститься в консольном приложении. Код службы должен быть таким:

Пример 14. Служба «банковского приложения»

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ServiceModel;

namespace WCF_BankService
{
    //контракт службы
    [ServiceContract]
    public interface IBankService
    {
        [OperationContract]
        void ToDeposit(double sum);
        [OperationContract]
        double GetBalance();
    }

    //класс службы
    public class BankService : IBankService
    {
        static int id = 0;        //для нумерации создаваемых счетов
        public double Balance;    //баланс счета

        //создание нового счета
        public BankService()
        {
            ++id;
            Console.WriteLine("Создан счет № " + id.ToString());
        }
        //перевод денег на созданный счет
        public void ToDeposit(double sum)
        {
            Balance += sum;
        }

        //вывод текущего баланса счета
        public double GetBalance()
        {
            return Balance;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ServiceHost sh = new ServiceHost(typeof(BankService));

            sh.Open();
            Console.WriteLine("Для завершения нажмите<ENTER>\n\n");
            Console.ReadLine();
            sh.Close();
        }
    }
```

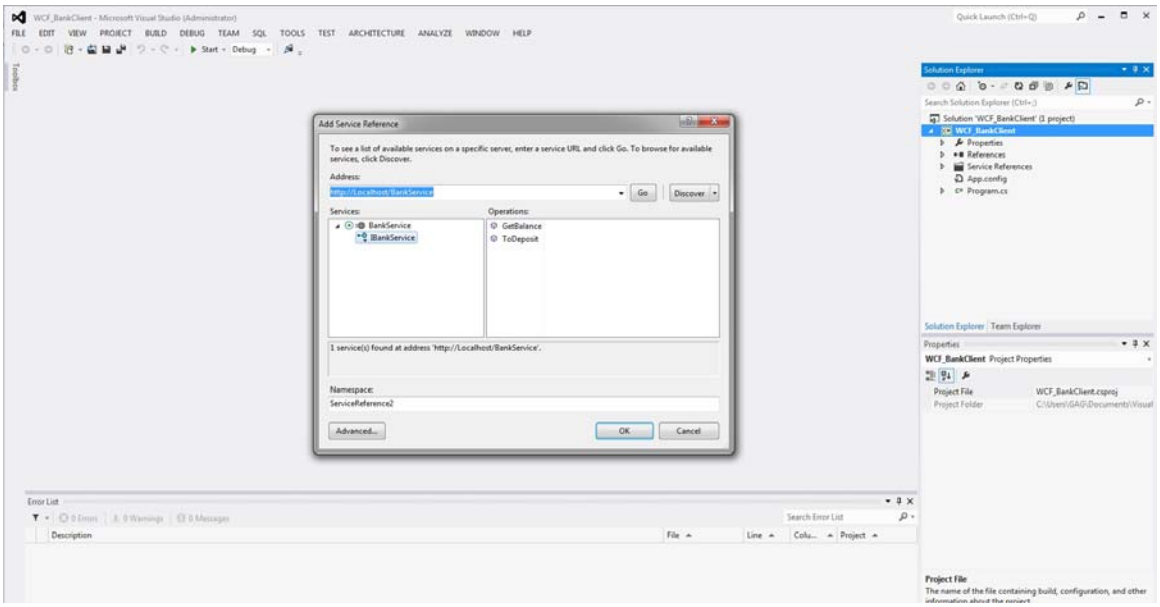
```
}
```

Затем добавьте к приложению конфигурационный файл. При создании конфигурационного файла необходимо указать привязку для оконечной точки `wsHttpBinding`, а не `basicHttpBinding`, как мы делали до сих пор. Зачем это надо, вы узнаете немного позже. Кроме того, не забудьте создать для службы mex-точку. Конфигурационный файл должен выглядеть, например, так:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
    <system.serviceModel>
        <behaviors>
            <serviceBehaviors>
                <behavior name="NewBehavior0">
                    <serviceMetadata httpGetEnabled="true" />
                </behavior>
            </serviceBehaviors>
        </behaviors>
        <services>
            <service behaviorConfiguration="NewBehavior0" name="WCF_BankService.BankService">
                <endpoint address="http://localhost/BankService/ep1" binding="ws2007HttpBinding"
                    bindingConfiguration="" contract="WCF_BankService.IBankService" />
                <endpoint binding="mexHttpBinding" bindingConfiguration="" name="mex"
                    contract="IMetadataExchange" />
                <host>
                    <baseAddresses>
                        <add baseAddress="http://Localhost/BankService" />
                    </baseAddresses>
                </host>
            </service>
        </services>
    </system.serviceModel>
</configuration>
```

Протестируйте службу. Если есть ошибки – исправьте их, если все в порядке – оставьте службу активной и перейдите к созданию клиента. Клиента тоже создадим в виде консольного приложения. Как обычно, не забудем добавить ссылку на пространство имен `System.ServiceModel` и ссылку на службу, которую только что создали. Давайте вспомним, как добавляется ссылка на службу:

- Служба должна быть запущена;
- В обозревателе решений клиента выберите из контекстного меню команду Add Service Reference (Добавить ссылку на службу…);
- В поле Address (Адрес) появившегося окна занесите базовый адрес нашей банковской службы http://Localhost/BankService и нажмите кнопку Go (Перейти);



- После добавления ссылки на службу вставьте в код клиента строку using WCF_BankClient.ServiceReference1.

Код клиента должен выглядеть так:

Пример 15. Клиент «банковского приложения»

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using WCF_BankClient.ServiceReference1;

namespace WCFBankClient
{
    class Program
    {
        static void Main(string[] args)
        {
            BankServiceClient proxy = new BankServiceClient();
            Console.WriteLine("Укажите сумму депозита:");
            double sum = Convert.ToDouble(Console.ReadLine());
            double result = 0;

            while (sum > 0)
            {
                proxy.ToDeposit(sum);
                result = proxy.GetBalance();
                Console.WriteLine("Депозит = {0}", result);
                Console.WriteLine("Укажите сумму депозита:");
                sum = Convert.ToDouble(Console.ReadLine());
            }

            Console.WriteLine("Для завершения нажмите<ENTER>\n\n");
            Console.ReadLine();
```

```
        }
    }
}
```

Все как обычно. Сначала создаем прокси-класс. Вызываем метод службы ToDeposit (), чтобы добавить деньги на свой счет в банке. Затем вызываем метод GetBalance(), чтобы посмотреть, сколько денег у нас на счете. Повторяем эти вызовы в цикле, чтобы увидеть особенности поведения службы, ради которых мы и создали эти два приложения.

Запустите клиента и в ответ на запрос «Укажите сумму депозита:» несколько раз введите какие-либо суммы. Обратите внимание на сообщение службы и на результаты работы метода GetBalance() в строке вывода клиента «Депозит =». Все очевидно, не так ли? Был создан один счет – №1, и все ваши деньги последовательно зачисляются на него, о чем говорит увеличивающееся значение, возвращаемое методом GetBalance(). Наверное, этого вы и ожидали. Теперь, не закрывая запущенного клиента, запустите еще несколько копий нашего клиентского приложения. Вы увидите, что запуск нового клиента приводит к созданию нового счета (или, другими словами, к созданию нового экземпляра нашей службы). Все суммы, введенные вами в каждом клиенте, добавляются к балансу счета, созданного для этого клиента.

Теперь перейдите к коду службы и перед определением класса BankService добавьте такой атрибут:
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)] . Перестройте проект службы, запустите службу и снова запустите несколько клиентов. Сейчас наш «банк» ведет себя иначе. Счет создается только один, независимо от количества запущенных клиентов, и деньги от всех клиентов переводятся на этот счет. В этом случае в нашем приложении создается один единственный объект службы, который работает со всеми клиентами.

Снова вернемся к коду службы и изменим значение в атрибуте, чтобы он выглядел так: [ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]. Снова запустите службу, затем запустите одного клиента и внимательно смотрите на выводимые сообщения – как службы, так и клиента. Затем запустите еще одного клиента. Вам уже понятно, что происходит? В этом случае новый объект службы создается при каждом обращении к службе, т.е. при вызове каждого метода службы. Поэтому для одной итерации одного клиента служба говорит о создании двух счетов. Один счет (экземпляр службы) создается при вызове метода ToDeposit (), а другой – при вызове метода GetBalance(). Последний метод сообщает о нулевом депозите потому, что он «считывает» баланс счета, который был создан только что, а вызывавшийся перед этим метод ToDeposit () положил деньги на депозит другого экземпляра службы.

Этот пример продемонстрировал вам основные типы поведения службы.

2. Сессии

Из предыдущего примера вы увидели, что по умолчанию служба ведет себя соответственно значению свойства InstanceContextMode.PerSession, т.е. служба старается работать с клиентами в режиме сессий. Давайте немного поговорим о сессиях. Этот термин имеет специальное значение в приложениях типа клиент-сервер.

Обычно серверная часть приложения ведет себя следующим образом. Она получает запрос от клиента, выполняет этот запрос и тут же отключается от клиента, чтобы быть готовой принять следующий запрос от следующего клиента. А что происходит, если сервер получает второй, третий или десятый запрос от клиента, с которым сервер только что работал перед этим? Как правило, сервер «не узнает» такого клиента, для него все клиенты безлики. Сервер не запоминает клиентов, с которыми общается. Так ведут себя, например, все веб-серверы, для которых любой запрос от вашего браузера будет интерпретироваться, как «первый». Вы должны понимать, что при таком режиме общения сервер просто не может сохранять какую-либо информацию о клиенте между запросами. Клиентов то много, а он один. Но иногда необходимо сделать так, чтобы сервер мог «помнить» клиента и сохранять информацию о клиенте между запросами. Это и достигается с помощью сессий. Сессия – это сеанс работы сервера и клиента, на протяжении которого, сервер и клиент «узнают» друг друга и могут хранить общую информацию от запроса к запросу.

В случае WCF когда клиент создает объект прокси-класса для службы реализующей сессии, он получает в свое распоряжение «собственный» объект службы, независимый от других объектов этой же службы. И этот «собственный» объект службы будет оставаться активным до тех пор, пока активен соответствующий клиент, т.е. пока клиент не закроет свой объект прокси-класса.

Как мы сказали, по умолчанию служба старается создать сессию при общении с каждым обратившимся к ней клиентом. Однако это не значит, что сессия обязательно будет реализована. Получится установить сессию или нет – зависит от значения свойства SessionMode атрибута ServiceContract и от используемой привязки. Свойство SessionMode имеет три допустимых значения, определенных в перечислении SessionMode:

- SessionMode.Allowed указывает, что служба создаст сессию, если входящая привязка поддерживает сессии;
- SessionMode.Required указывает, что служба требует создания сессии; если входящая привязка не поддерживает сессии – будет сгенерирована исключительная ситуация;

- SessionMode.NotAllowed указывает, что служба не допускает использования привязки, инициализирующей создание сессии.

Значение свойства SessionMode атрибута ServiceContract по умолчанию – SessionMode.Allowed. При этом транспортные сессии разрешены, но их реализация не гарантирована. Как уже говорилось, результирующее поведение службы будет зависеть также и от привязки. Например, такие привязки как BasicHttp и WSHttpBinding (без защиты сообщений и без контроля доступа) не позволят реализовать сессии на транспортном уровне, даже если указаны значение свойства InstanceContextMode.PerSession и SessionMode.Allowed. А при использовании привязок WSHttpBinding (с защитой сообщений или с контролем доступа), NetTcpBinding или NetNamedPipeBinding, сессии будут успешно реализованы.

Для того, чтобы связывать все сообщения от одного клиента с одним объектом службы, этого клиента надо каким-либо образом идентифицировать. Такая идентификация поддерживается на уровне транспортного протокола, который, как вы понимаете, определяется привязкой. Поэтому при проектировании службы реализующей сессии к выбору привязок надо относиться особенно внимательно. Если вы используете сессии, то вам будет полезно знать, что каждая сессия имеет свой уникальный идентификатор. И сама служба, и клиент могут прочитать этот идентификатор.

В пространстве имен System.ServiceModel определен класс OperationContext. У этого класса есть статическое свойство Current позволяющее получить доступ к среде выполнения текущего потока. Это свойство имеет тип OperationContext, и его свойство SessionId как раз и содержит идентификатор сессии. Другими словами, чтобы получить идентификатор текущей сессии в службе можно использовать такой код:

```
string sessionID = OperationContext.Current.SessionId;
```

Если же вы используете службу, не поддерживающую сессии, то в строковой переменной sessionID в результате выполнения этого кода вы получите null.

В коде клиента, для получения идентификатора сессии надо использовать прокси-класс. Вы должны помнить, что все прокси-классы наследуют базовому классу ClientBase у которого есть свойство InnerChannel. Поэтому можно использовать такой подход:

```
string sessionID = proxy.InnerChannel.SessionId;
```

Однако клиент сможет получить идентификатор сессии только после того, как он отправит службе запрос, инициирующий сессию. И если сессия будет открыта и будет поддерживаться правильными настройками, включая привязки. Поэтому получать идентификатор сессии в клиенте надо в «правильном» месте кода.

При использовании привязок, поддерживающих сессии, каждая привязка может добавлять некие свои специфические черты. Например, привязка WSHttpBinding использует шифрование и цифровую подпись для сообщений. Привязка NetTcpBinding организует сессии, базируясь на протоколе TCP/IP. Привязка NetMsmqBinding позволяет службе и клиенту работать вообще без постоянного подключения между ними. При этом совсем не используются протоколы TCP или HTTP. Сообщения клиента направляются не оконечной точке службы, а Microsoft службе очереди сообщений MessageQueuing (MSMQ). Эти вызовы будут затем обрабатываться службой MSMQ. На всякий случай отметьте, что служба MSMQ использует только порт 1801 и никакой другой. А привязка basicHttpBinding не поддерживает сессии, именно поэтому мы ее не использовали в нашем примере.

Возьмите на заметку одно важное правило. Не стоит в одной службе объединять контракты, реализующие поведение объекта службы в режиме PerCall и PerSession. Объединения двух таких контрактов в одной службе сильно усложнит реализацию и поддержку функционирования такой службы.

При пересылке сообщений от клиента к службе важным является вопрос о контроле доставке сообщения. От чего зависит выполняется такой контроль или нет? Это зависит от транспортного протокола, используемого в сессии. Следовательно, если нам нужен контроль доставки сообщений, мы должны проследить, чтобы все оконечные точки контракта, использующего сессию, применяли такие привязки, которые реализуют транспортный протокол, позволяющий выполнять контроль доставки. При этом контроль доставки надо активизировать явным образом и на стороне клиента и на стороне службы. Это можно сделать в конфигурационном файле на стороне службы:

```
<system.serviceModel>
  <services>
    <service name = "SessionService">
      <endpoint
      address = "net.tcp://localhost/SessionService"
      binding = "netTcpBinding"        <!-ссылка на привязку->
      bindingConfiguration = "TCPSession"
      contract = "IContract"
      />
    </service>
  </services>
  <bindings>
    <netTcpBinding>
      <binding name = "TCPSession">    <!-сама привязка->
        <reliableSession enabled = "true"/>  <!-включен контроль доставки->
      </binding>
    </netTcpBinding>
  </bindings>
</system.serviceModel>
```

И в конфигурационном файле на стороне клиента:

```
<system.serviceModel>
  <client>
```

```
      <endpoint
      address = "net.tcp://localhost/SessionService/"
      binding = "netTcpBinding"
      bindingConfiguration = "TCPSession"
      contract = "IContract"
      />
    </client>
    <bindings>
      <netTcpBinding>
        <binding name = "TCPSession">
          <reliableSession enabled = "true"/>
        </binding>
      </netTcpBinding>
    </bindings>
  </system.serviceModel>
```

Попутно обратите внимание, когда мы явно указываем, что нам необходим контроль доставки сообщений, WCF автоматически позаботится о том, чтобы сообщения доставлялись в очередности их отправки.

Однако у разработчика есть еще некоторые рычаги для управления поведением сессий. Мы можем точно управлять моментом создания сессии и моментом ее завершения. Вы помните, что каждый метод службы, реализующий контракт службы, помечен атрибутом [OperationContract]. У этого атрибута есть полезные для нас свойства. Давайте рассмотрим контракт воображаемой службы:

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface ITestSession
{
    [OperationContract(IsOneWay=true, IsInitiating=true, IsTerminating=false)]
    void Init();

    [OperationContract(IsOneWay = true, IsInitiating = false, IsTerminating = false)]
    void Do1(double n);
    [OperationContract(IsOneWay = true, IsInitiating = false, IsTerminating = false)]
    void Do2(double n);

    [OperationContract(IsInitiating = false, IsTerminating = true)]
    double Finish();
}
```

Внимательно рассмотрите значения булевских свойств IsInitiating и IsTerminating. Когда мы обращаемся к методу, в атрибуте которого свойство IsInitiating равно true, этот метод создает (или начинает) сессию. Правда, это будет только в том случае, когда такой метод вызывается первым. Однако, если такой метод будет вызван после того, как другой метод уже открыл сессию, то этот метод станет частью уже существующей сессии. Метод, в атрибуте которого свойство IsTerminating равно true, начинает асинхронное завершение текущей сессии после окончания своего выполнения. Обратите внимание, что объект прокси-класса, сессия которого завершилась, использоваться больше не может. Этот объект надо явно закрыть на стороне клиента. Таким образом, клиент может открывать или прекращать сессию вызовами соответствующих методов.

В нашем примере для начала сессии клиенту надо будет обратиться к методу Init(), при этом будет начата новая сессия. Затем клиент будет работать со службой, вызывая методы Do1() и Do2(), которые будут выполняться в границах одной сессии, но сами на сессию влиять не будут. Когда сессию надо будет прервать, клиент должен будет вызвать метод Finish(). Этот метод закроет текущую сессию и вернет некий результат.

3. Характеристика поведения службы PerCall

Рассмотрим достоинства и недостатки трех типов поведения службы. Если для свойства InstanceContextMode выбрано значение InstanceContextMode.PerCall, то мы получаем стандартную для архитектуры клиент/сервер модель, при которой для каждого клиента создается «собственный» объект сервера. Такое поведение чаще всего и является ожидаемым от таких систем. Однако главный недостаток этого подхода сразу бросается в глаза. При большом количестве клиентов, обращающихся к серверу, система должна создавать, поддерживать, а затем удалять большое количество объектов сервера. А если каждый такой объект требует значительных ресурсов, то недостаток такого подхода только усугубляется. Но и это еще не все. Очень часто клиент «держит» свой объект сервера на протяжении длительного времени, при том, что использует этот объект на протяжении времени значительно более короткого. Но в WCF эти недостатки минимизированы.

WCF реализует данную модель поведения более интеллектуальным способом. В WCF объект сервера «присваивается» клиенту только на время действия запроса от клиента. Поэтому создается и поддерживается в памяти столько объектов сервера, сколько активных запросов существует на данный момент, а не сколько клиентов обратились к серверу к данному моменту времени. А количество активных запросов обычно на порядок-два меньше общего количества клиентов. Если объект сервера клиентом не используется, WCF удаляет такой объект, даже если клиент не закрыл ссылку на прокси. Т.е. ссылка клиента на сервер остается активной, но объекта сервера, соответствующего этой ссылке может уже не существовать. Этот объект уже был создан, поработал с клиентом во время запроса, а по завершении запроса был удален. Но связь клиента с сервером (объект прокси-класса на стороне клиента) остается активной. И если клиент пришлет новый запрос – на стороне сервера будет создан новый объект сервера, который обслужит этот запрос, а по завершении запроса будет удален в свою очередь.

Дело в том, что повторное создание и удаление объектов сервера без разрыва связи с клиентом намного проще и «дешевле», чем повторное создание связи с клиентом и создание при этом объекта сервера.

Итак, поведение объекта службы, определяемое значением InstanceContextMode.PerCall, является оптимальным для большинства случаев

использования в архитектуре WCF, поскольку очевидные недостатки этого подхода в WCF минимизированы.

4. Характеристика поведения службы Single

Когда служба конфигурируется со значением InstanceContextMode.Single, то все клиенты будут подключаться к одному и тому же объекту службы. Этот объект службы создается единожды при создании среды выполнения службы и уничтожается при закрытии среды выполнения службы. Вы должны понимать некоторые издержки использования одного объекта службы. Когда к одному объекту имеют доступ много клиентов, то может возникнуть известная проблема синхронизации доступа. Чтобы избежать этой проблемы, доступ к объекту должен предоставляться только для одного клиента в каждый момент времени. Поэтому вы должны хорошо понимать, что использование значения InstanceContextMode.Single в случае, когда со службой должно общаться большое количество клиентов может приводить к замедлению работы.

5. Поведения, регулирующие нагрузку на службу

Сейчас вы видите, что для значений InstanceContextMode PerCall или PerSession для каждого клиента будет создаваться свой экземпляр службы. Иногда бывает необходимо по тем или иным причинам задать ограничения для количества создаваемых экземпляров службы. В этом случае надо воспользоваться поведением maxConcurrentInstances. Это поведение надо указать в конфигурационном файле – в разделе <behaviors> в подразделе <serviceBehaviors>. Здесь надо указать следующий элемент <behavior name="throttling">, а в нем - <serviceThrottling maxConcurrentInstances="5"/>. Как вы понимаете, в этом случае количество одновременно существующих экземпляров службы будет ограничено пятью объектами. Новые экземпляры службы не будут создаваться, до тех пор, пока не будут уничтожены какие-либо из существующих экземпляров. Понятно, что это может привести к задержкам для клиента. Эта часть конфигурационного файла может выглядеть так:

```
<behaviors>
  <serviceBehaviors>
    <behavior name="throttling">
      <serviceThrottling maxConcurrentInstances="5"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

В том случае, когда значение InstanceContextMode равно Single и ConcurrencyMode равно Multiple, WCF создаст один единственный экземпляр службы, и для каждого запроса к этому экземпляру будет создаваться новый поток для параллельного

выполнения запрошенных методов службы. Если вы хотите ограничить количество одновременно выполняющихся потоков, вы можете использовать поведение maxConcurrentCalls. Ниже приведен фрагмент конфигурационного файла, ограничивающего количество потоков до 20:

```
<behaviors>
  <serviceBehaviors>
    <behavior name="throttling">
      <serviceThrottling maxConcurrentCalls="20"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

И, наконец, в том случае, когда значение InstanceContextMode равно PerSession и ConcurrencyMode равно Multiple, WCF создаст один экземпляр службы для каждой сессии. Если вы хотите ограничить количество клиентов, работающих со службой, т.е. число одновременно открытых сессий, вы должны использовать поведение maxConcurrentSessions. В этом случае, при достижении максимально возможного количества сессий, новые сессии не будут создаваться, пока не закроется какая-либо из существующих. Ниже приведен фрагмент конфигурационного файла, ограничивающего количество одновременно подключенных клиентов до 10:

```
<behaviors>
  <serviceBehaviors>
    <behavior name="throttling">
      <serviceThrottling maxConcurrentSessions ="10"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

6. Транзакции

Изучая принципы работы с базами данных, вы должны были запомнить такое важное понятие, как транзакция. Сейчас мы поговорим о транзакциях в технологии WCF. Одним из важнейших свойств транзакции является то, что транзакция никогда не разрушает целостность системы. Транзакция либо успешно завершается и переводит систему в новое (целостное) состояние, либо, в случае какой-либо ошибки, оставляет систему в том состоянии, в котором система находилась до начала транзакции. Вообще говоря, транзакция – это цепочка определенных действий, выполняемых в определенном порядке. И если хоть одно из этих действий не может быть выполнено – вся транзакция отменяется (откатывается). Обычно транзакции требуются при работе с такими ресурсами, как базы данных или очереди сообщений.

WCF строго поддерживает основные четыре свойства транзакций: атомарность, целостность, изолированность и надежность. Транзакционность – это характеристика операции. Поэтому, если мы хотим, чтобы какая-либо операция выполнялась, как

транзакция, мы должны создать для такой операции специальное поведение. Давайте вспомним наше «банковское» приложение, и модифицируем его таким образом, чтобы метод ToDeposit() выполнялся, как транзакция. Такая модификация весьма уместна, так как этот метод изменяет состояние банковского счета, и мы должны быть уверены, что счет будет в целостном состоянии, при любом исходе: либо изменится при успешном выполнении метода, либо останется таким, как был, в случае возникновения какой-либо ошибки в работе метода ToDeposit(). Для этого мы должны изменить определение контракта и операции следующим образом (изменения выделены жирным шрифтом):

Пример 16. Служба «банковского приложения» с локальной транзакцией

```
//контракт службы
[ServiceContract]
public interface IBankService
{
      [OperationContract]
      void ToDeposit(double sum);
      [OperationContract]
      double GetBalance();
}

//класс службы
public class BankService : IBankService
{

      //перевод денег на созданный счет
      [OperationBehavior(TransactionScopeRequired = true,
            TransactionAutoComplete = true)]
      public void ToDeposit(double sum)
      {
            Balance += sum;
      }

      //здесь описаны другие члены класса

}
```

Вы видите, что в этом случае мы добавили поведение для операции ToDeposit() и выставили булевское свойство TransactionScopeRequired этого поведения равным true. Теперь всякий раз при вызове метода ToDeposit(), служба будет создавать новую транзакцию и выполнять метод в этой транзакции. Понятно, что для нашего конкретного метода ToDeposit() транзакционность не важна, но вот если бы этот метод работал с базой данных, вызывал внутри себя другие методы, то тогда создание транзакционности для него было бы просто необходимым. Мы использовали еще одно булевское свойство поведения – TransactionAutoComplete и установили его равным true. Это нужно для того, чтобы операция считалась завершенной, если в ходе работы метода ToDeposit() не возникло ошибок. При этом транзакция будет выполнена. Если

же в ходе работы метода ToDeposit() возникла любая ошибка – будет выполнен откат транзакции.

В клиенте при этом никаких изменений выполнять не надо. Клиент не знает, в каком режиме будет выполняться вызванный им метод. Когда вы запустите такой вариант приложения, в окне клиента вы будете видеть все время нулевое значение баланса. Это будет происходить потому, что из-за свойства поведения TransactionAutoComplete = true завершение метода ToDeposit() будет завершать транзакцию. И если в клиенте мы вызовем метод GetBalance() после окончания транзакции, то увидим нулевой баланс. Исправить эту ситуацию можно, поместив вызов метода GetBalance() в метод ToDeposit(), например, так:

```
[OperationBehavior(TransactionScopeRequired = true,
    TransactionAutoComplete = true)]
public void ToDeposit(double sum)
{
    Console.WriteLine("Изменение баланса");
    this.Balance += sum;
    double bal = GetBalance();
    Console.WriteLine("Баланс: {0}", bal);
}
```

Теперь мы будем видеть значение баланса в ходе выполнения транзакции, и это значение будет соответствовать введенному нами в клиенте значению. Правда, теперь значение баланса будет выводиться в окне службы. Транзакция, которую мы сейчас реализовали, называется локальной. Полный код измененной службы приведен в дополнении к уроку.

а. Поток транзакций

Однако специфика транзакций в WCF состоит в том, что иногда транзакции должны распространяться между разными службами. Ведь в SOA приложении возможна ситуация, когда клиент вызовет метод службы, который, в свою очередь, вызовет какой-либо метод другой службы. И если первый вызванный метод транзакционный, то и второй должен выполняться в рамках транзакции, несмотря на то, что он определен в другой службе. Когда транзакция распространяется на несколько служб, в WCF это называется потоком транзакций.

Такие транзакции уже сложнее и требуют участия как всех затронутых служб, так и клиента. Чтобы реализовать поток транзакций необходимо в контракте каждой службы потребовать создания сессий, выставив режим SessionMode.Required. Кроме этого, надо для соответствующих операций добавить поведения со свойством TransactionScopeRequired=true, а в контракте операции создать свойство TransactionFlowOption.Allowed. И, наконец, в определении привязки указать свойство TransactionFlow=true (при этом привязка, конечно же, должна поддерживать сессии!).

Только такие привязки, как wsHttpBinding, wsDualHttpBinding, netTcpBinding, netNamedPipeBinding и netPeerTcpBinding поддерживают транзакции.

Мы уже указывали, каким образом инициализировать свойства TransactionScopeRequired и TransactionFlowOption. Приведем пример инициализации свойства привязки TransactionFlow. Это свойство можно инициализировать как в конфигурационном файле:

```
<bindings>
      <netTcpBinding>
            <binding name = "TransactionalTCP"
                  transactionFlow = "true"
            />
      </netTcpBinding>
</bindings>
```

так и в коде:

```
NetTcpBinding tcpBinding = new NetTcpBinding();
tcpBinding.TransactionFlow = true;
```

Все эти изменения выполняются на стороне службы. Кроме них необходимо еще кое-что предпринять на стороне клиента.

Клиент должен определенным образом вызывать транзакционный метод. Самый простой способ явно использовать транзакции предоставляет класс TransactionScope, определенный в пространстве имен System.Transactions. Этот класс позволяет выделить часть кода, реализующую транзакцию. В этом классе определен метод Complete(), который запускает выполнение транзакции.

```
using (TransactionScope scope = new TransactionScope())
{
      //здесь кодируются действия транзакции
      //например, подключиться к базе данных, выполнить запрос
      //если все действия успешно выполнены – запустить транзакцию
      scope.Complete();
}
```

Ниже приведен пример кода службы, реализующий поток транзакций для нашего «банковского» приложения. Давайте предположим, что метод ToDeposit() вызывает какой-нибудь другой метод, определенный в другой службе. Именно поэтому нам и надо реализовать поток транзакций. Изменения выделены жирным шрифтом:

Пример 17. Служба «банковского приложения» с потоком транзакций

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using System.Transactions;
```

```
namespace WCF_BankService
{
    //контракт службы
    [ServiceContract(SessionMode = SessionMode.Required)]  //первое изменение
    public interface IBankService
    {
        [OperationContract]
        void ToDeposit(double sum);
        [OperationContract]
        double GetBalance();
    }

    //класс службы
    public class BankService : IBankService
    {
        static int id = 0;          //для нумерации создаваемых счетов
        public double Balance;      //баланс счета

        //создание нового счета
        public BankService()
        {
            ++id;
            Console.WriteLine("Создан счет № " + id.ToString());
        }

        //перевод денег на созданный счет
        [OperationBehavior(TransactionScopeRequired = true,
            TransactionAutoComplete = false)]               //второе изменение
        [TransactionFlow(TransactionFlowOption.Allowed)]    //третье изменение
        public void ToDeposit(double sum)
        {
            Console.WriteLine("Изменение баланса");
            this.Balance += sum;
            System.Transactions.Transaction trans =
                  System.Transactions.Transaction.Current;
            double bal = GetBalance();
            Console.WriteLine("Баланс: {0}  Transaction ID: {1}",
                  bal,
                  Transaction.Current.TransactionInformation.LocalIdentifier.ToString()
            );
        }
        //вывод текущего баланса счета
        public double GetBalance()
        {
            Console.WriteLine("Запрос баланса");
            return this.Balance;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ServiceHost sh = new ServiceHost(typeof(BankService));

            sh.Open();
            Console.WriteLine("Для завершения нажмите<ENTER>\n\n");
            Console.ReadLine();
            sh.Close();
```

```
        }
    }
}
```

В конфигурационный файл службы надо добавить такой раздел:

```
<!--четвертое изменение-->
<bindings>
  <wsHttpBinding>
    <binding name="transactions"
    transactionFlow="true" >
    </binding>
  </wsHttpBinding>
</bindings>
```

Итак, если и служба и клиент будут содержать настройки позволяющие распространение транзакций для определенных операций, тогда такие операции будут выполняться, как транзакции со всеми вытекающими последствиями: сохранение целостности системы, надежность и т.п.

Можем ли мы быть уверенными в том, что запланированные нами действия выполняются в границах одной транзакции? Да. В этом нам поможет класс Transaction, определенный в пространстве имен System.Transactions. Этот класс отвечает за реализацию транзакций, однако разработчикам обращаться к нему стоит лишь для того, чтобы посмотреть статус транзакции и ее идентификатор. В WCF введено понятие текущей транзакции – транзакции, в которой выполняется наш код. Мы можем получить доступ к текущей транзакции таким образом:

```
Transaction trans = Transaction.Current;
```

Если окажется, что trans равно null, то это будет говорить о том, что в данный момент никаких транзакций нет. Доступ к текущей транзакции можно получить как из службы, так и из клиента.

Кроме этого у объекта Transaction.Current есть свойство TransactionInformation типа TransactionInformation. В этом типе определено два важных члена: DistributedIdentifier типа Guid и LocalIdentifier типа string. Это идентификаторы распределенной и локальной транзакций. Мы воспользуемся в нашей службе идентификатором локальной транзакции, чтобы выводить его на экран.

Теперь рассмотрим, каким образом надо сконструировать код нашего клиента. Все действия, которые должны выполняться в рамках одной транзакции, в клиенте необходимо реализовать с помощью класса TransactionScope, например, таким образом:

Пример 18. Клиент «банковского приложения» с потоком транзакций

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.ServiceModel;
using WCF_BankClient.ServiceReference1;
using System.Transactions;

namespace WCF_BankClient
{
    class Program
    {
        static void Main(string[] args)
        {
            using (TransactionScope scope = new TransactionScope
               (TransactionScopeOption.RequiresNew))
            {
                BankServiceClient proxy = new BankServiceClient();
                Console.WriteLine("Укажите сумму депозита:");
                double sum = Convert.ToDouble(Console.ReadLine());
                double result = 0;

                while (sum > 0)
                {
                    proxy.ToDeposit(sum);
                    result = proxy.GetBalance();
                    Console.WriteLine("Депозит = {0}", result);
                    Console.WriteLine("Укажите сумму депозита:");
                    sum = Convert.ToDouble(Console.ReadLine());
                }
                scope.Complete();
            }
            Console.WriteLine("Для завершения нажмите<ENTER>\n\n");
            Console.ReadLine();
        }
    }
}
```

Обратите внимание на вызов метода scope.Complete(), который завершает транзакцию.

Если мы запустим нашу службу, а потом начнем обращаться к ней из клиента, последовательно вводя значения 1000, 5000 и 100000, то в окне службы мы увидим следующую информацию:

```
Создан счет № 1
Изменение баланса
Запрос баланса
Баланс: 1000  Transaction ID: d593b1b5-3c2f-45b6-8816-fd939e0d5365:1
Запрос баланса
Изменение баланса
Запрос баланса
Баланс: 6000  Transaction ID: d593b1b5-3c2f-45b6-8816-fd939e0d5365:1
Запрос баланса
Изменение баланса
Запрос баланса
Баланс: 106000  Transaction ID: d593b1b5-3c2f-45b6-8816-fd939e0d5365:1
```

Обратите внимание, что все три обращения клиента выполняются в рамках одной сессии и работают с одним объектом службы. Об этом говорит фраза из конструктора класса BankService «Создан счет № 1». Вы видите, что конструктор вызывался только один раз. Кроме того, все три обращения выполняются в границах одной транзакции, о чем говорит одно и то же значение Transaction.Current.TransactionInformation.LocalIdentifier при трех разных вызовах метода ToDeposit(). Значение свойства поведения TransactionAutoComplete сейчас равно false и поэтому каждое завершение метода ToDeposit() не завершает транзакцию. Транзакцию в этом случае завершает клиент.

7. Особенности обработки исключительных ситуаций

Рассмотрим еще один аспект поведения WCF, а именно – обработку исключительных ситуаций. Для демонстрации нам понадобится наша «математическая» служба с методом Total(). Уберите проверку на неравенство делителя нулю, перед выполнением операции деления, чтобы тело метода приняло такой вид:

```
public MathResult Total(int x, int y)
{
    MathResult mr = new MathResult();
    mr.sum = x + y;
    mr.subtr = x - y;
    mr.div = x / y;
    mr.mult = x * y;
    return mr;
}
```

Перестройте проект и запустите службу. Теперь запустите клиента, передав в качестве второго параметра нулевое значение. Вы получите следующее сообщение об исключительной ситуации:

> Необработанное исключение типа "System.ServiceModel.FaultException" произошло в mscorlib.dll
>
> Дополнительные сведения: Серверу не удалось обработать запрос из-за внутренней ошибки. Для получения дополнительной информации об ошибке включите IncludeExceptionDetailInFaults (или с помощью атрибута ServiceBehaviorAttribute, или из конфигурации поведения <serviceDebug>) на сервере с целью отправки информации об исключении клиенту, либо включите трассировку, согласно документации Microsoft .NET Framework 3.0 SDK, и изучите журналы трассировки сервера.

По умолчанию служба скрывает детали исключения, чтобы не раскрывать внешнему миру сведения
о своей реализации или инфраструктуре. Но это достаточно неудобно, особенно на этапе отладки приложения. Чтобы увидеть более подробную информацию об исключении, можно
воспользоваться свойством IncludeExceptionDetailInFaults поведения ServiceDebugBehavior. Это можно сделать несколькими способами.

Можно добавить в конфигурационный файл службы, в раздел <behaviors>, далее в раздел <serviceBehaviors>, и наконец в раздел <behavior> элемент следующего вида: <serviceDebug includeExceptionDetailInFaults="true" />. Этот элемент приведет к тому, что информация об исключительных ситуациях будет более конкретной.

Кроме этого, такого же результата можно достичь, указав в коде службы, перед классом службы следующий атрибут:
```
[ServiceBehavior(IncludeExceptionDetailInFaults = true)].
```

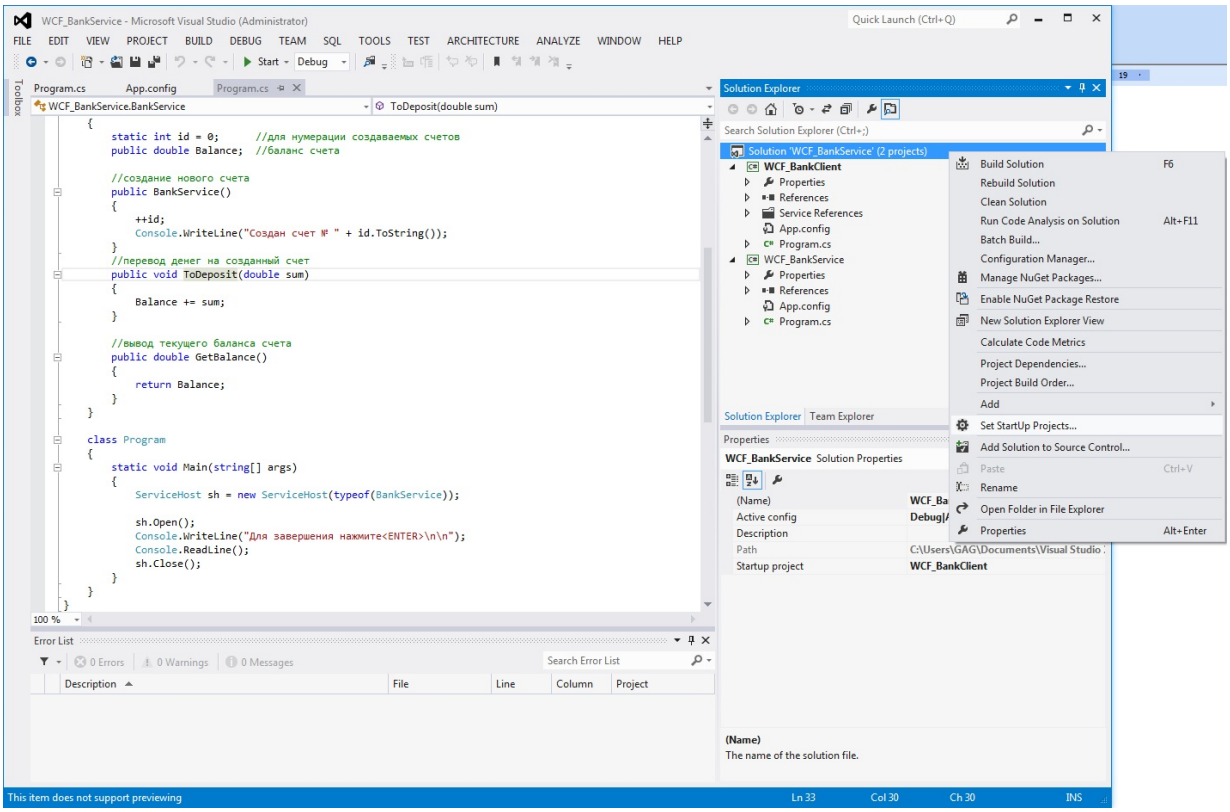
Инициализируйте свойство IncludeExceptionDetailInFaults поведения ServiceDebugBehavior значением «true» одним из этих способов и снова запустите клиента, передав вторым параметром нулевое значение. Вы снова получите сообщение об исключительной ситуации, но на этот раз сообщение будет таким:

> Необработанное исключение типа "System.ServiceModel.FaultException`1" произошло в mscorlib.dll
>
> Дополнительные сведения: Попытка деления на нуль.

Согласитесь, что теперь ситуация с ошибкой намного понятнее, чем в предыдущем случае. Однако не забывайте, что такой режим оповещения об исключительных ситуациях следует задавать только на время отладки, а затем его лучше отключать.

8. Отладка WCF приложений

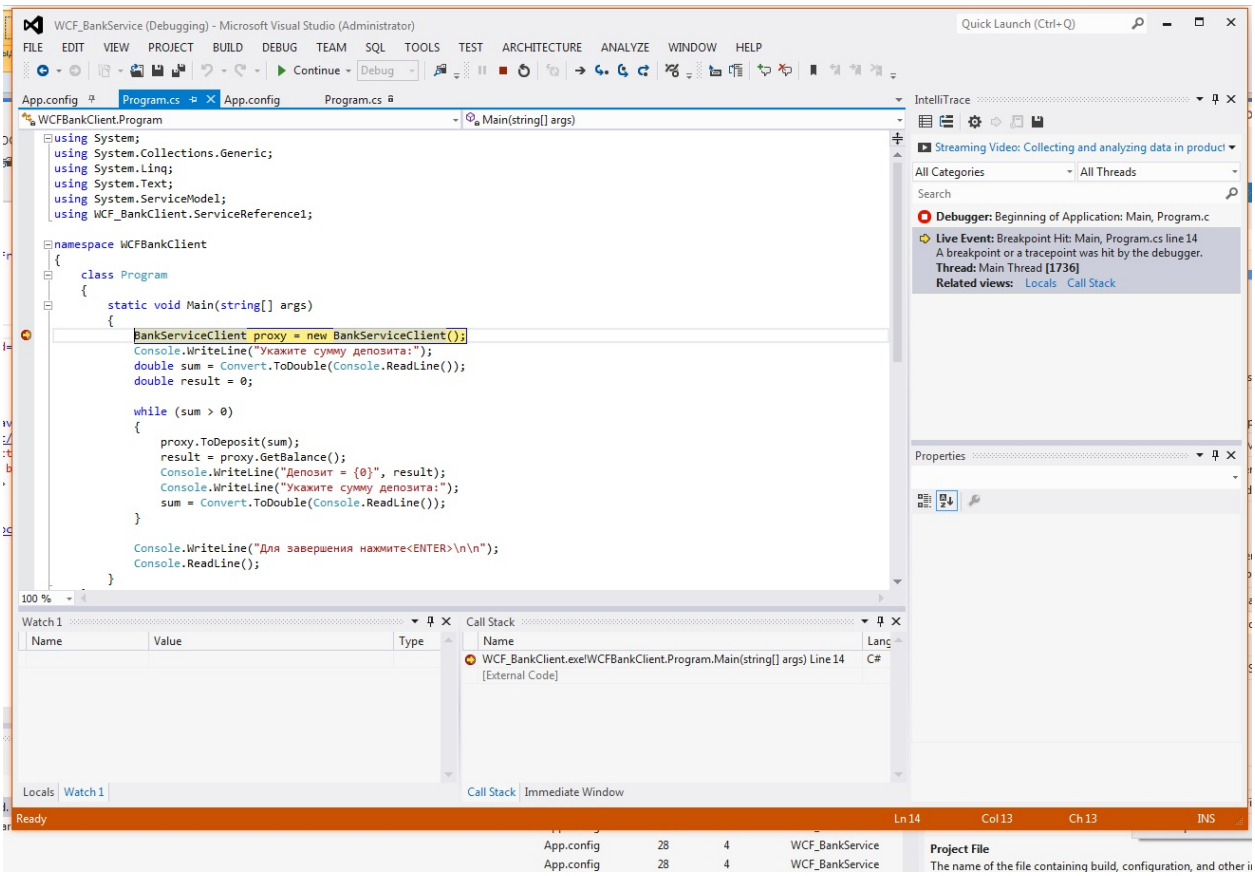
Рассмотрим процесс отладки WCF приложений на примере использования Visual Studio 2012. Прежде всего, надо позаботиться о том, чтобы и клиент и служба были созданы в одном решении (solution). Затем в обозревателе решений надо выбрать проект клиента, вызвать контекстное меню, кликнув по правой кнопке мыши, и выбрать в меню команду Set as Startup Project («Назначить запускаемым проектом»).



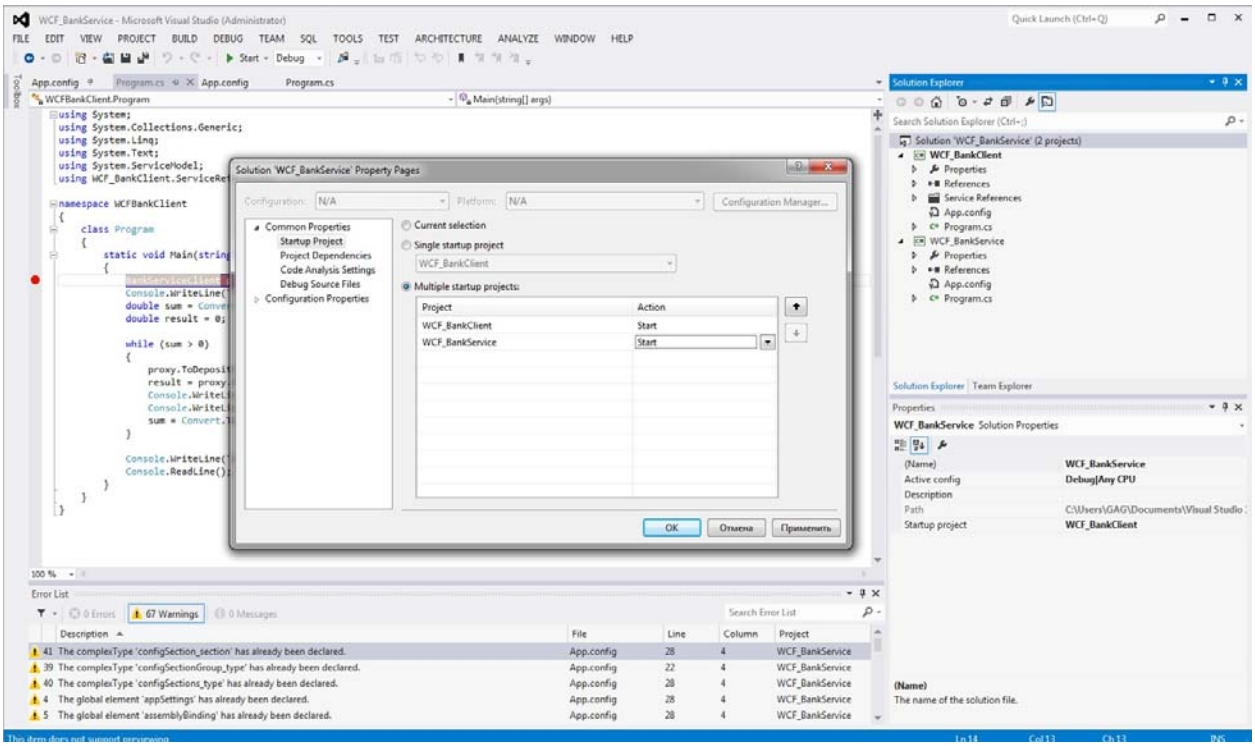
Теперь надо разрешить отладку, внеся соответствующие изменения в конфигурационный файл приложения. Чтобы разрешить отладку, надо в конфигурационный файл добавить такую секцию:

```
<system.web>
      <compilation debug="true" />
</system.web>
```

Теперь в коде клиента, где-нибудь перед вызовом службы надо установить точку останова (F9) и запустить приложения, чтобы оно выполнилось до этой точки остановка (F5). Теперь вы сможете пошагово отслеживать ход выполнения вашего приложения, выполняя строки по одной (F10). Однако это будет возможно только в том случае, когда *служба вызывается клиентом синхронно и в контракте службы нет односторонних операций*.



Понятно, что в режиме отладки и служба и клиент должны выполняться. Если служба хостится в консольном приложении, то в Visual Studio надо выполнить еще ряд действий. В обозревателе решений выделить имя решения, нажать правую кнопку мыши и выбрать команду Properties («Свойства»). При этом раскроется диалоговое окно. В правой части этого окна установить значение радио-кнопки в положение Multiple Startup Projects («Несколько запускаемы проектов»). В лист-боксе, расположенном ниже, в колонке Action («Действие») напротив проекта и службы и клиента выбрать значение Start («Запуск»). После этого нажать кнопку OK. Теперь вы получите возможность выполнять отладку и клиента и службы.



Теперь вы познакомились с еще одной современной технологией написания приложений. Вы должны понимать, что WCF вошла в нашу жизнь всего несколько лет назад, а, следовательно, вы находитесь, действительно на передовом рубеже IT-технологий. Применяйте свои знания на практике и используйте WCF там, где она поможет сделать ваши приложения более эффективными и конкурентоспособными.

Экзаменационное задание

В качестве экзаменационного задания предлагаем вам написать SOA-приложение, реализующее чат.

Серверная часть приложения должна быть службой WCF реализованной в виде dll-библиотеки размещенной в Windows службе. Клиент должен быть либо WindowsForms-приложением, либо WPF-приложением. Конечно же, клиент должен общаться со службой в дуплексном режиме.

Запуск каждого клиента должен приводить к созданию сессии, в границах которой и будет проходить общение этого клиента с серверной частью. Чат должен реализовывать как возможность «общего» чата, так и возможность клиентов общаться в режиме «один-на-один».

В окне службы должен отображаться список всех активных клиентов. О присоединении к чату каждого клиента и о выходе клиента из чата, служба должна оповещать всех активных клиентов.

Контракт службы может выглядеть, например, так:

```
[ServiceContract...]
public interface IChat
{
    [OperationContract...] //оповещение о присоединении к чату нового клиента Name
    string[] Join(string Name);

    [OperationContract...] //отправка общего сообщения msg (всем)
    void Send(string NameFrom, string msg);

    [OperationContract...] //отправка личного сообщения msg
                           //(от клиента NameFrom клиенту NameTo)
    void SendPrivate(string NameFrom, string msg, string NameTo);

    [OperationContract...] //оповещение о выходе клиента Name из чата
    void Leave(string Name);
}
```

Это всего лишь образец, который еще можно дополнять другими методами. Обратите внимание на символы «...» в атрибутах. Они говорят о том, что вам придется использовать разные свойства атрибутов для реализации сессий и других необходимых черт чата.