M Überblick über die 12. Übung

- Advanced .NET Remoting
  - Remotekonfigurationsdatei
  - WSDL, SOAP und UDDI
  - Metadaten-Assembly: soapsuds
  - Proxies
  - Sinks
  - Asynchrone Aufrufe
  - Parameterübergabe
  - Contexts
  - Leased-based Garbage Collector

1 Remotekonfigurationsdatei

- Zur einfachen Konfiguration der Kanäle und registrierten Objekte
- Format: XML
- Position und Name: beliebig, meist jedoch in Anwendungskonfigurationsdatei

```
<configuration>
    <system.runtime.remoting>
        ...
    </system.runtime.remoting>
</configuration>
```

- Konfigurationsmöglichkeiten
  `<lifetime />`, `<channels />`, `<service />`, `<client />`
- Einlesen:

```
RemotingConfiguration.Configure("Customer.exe.config");
```


1 Remotekonfigurationsdatei

- Server

```
<application>
  <service>
    ...
  </service>
  <channels>
    <channel port=4711
      type="System.Runtime.Remoting.Channels.Http.HttpChannel,
            System.Runtime.Remoting, Version=1.0.3300.0,
            Culture=neutral, PublicKeyToken=b77a5c561934e089" />
  </channels>
</application>
```

- Client

```
<application>
  <client>
    ...
  </client>
  <channels>
    <channel type="..." />
  </channels>
</application>
```


1 Remotekonfigurationsdatei

- Kanal-Vorlagen:

```
<configuration>
    <system.runtime.remoting>
        <application>
            <channels>
                <channel port=4711 ref="MyChannel" />
            </channels>
        </application>
        <channels>
            <channel id="MyChannel" type="..." />
        </channels>
    </system.runtime.remoting>
</configuration>
```

- vordefinierte Vorlagen z.B.: `http`, `tcp`

```
<channels>
    <channel port=4711 ref="http" />
</channels>
```

1 Remotekonfigurationsdatei

- ClientProviders / ServerProviders

```
<channels>
  <channel port=4711 ref="http">
    <serverProviders>
      <formatter />
      <provider />
    </serverProviders>
  </channel>
</channels>
```

  - Provider: Definition der Sinks, durch die die Nachricht gereicht wird (z.B. zur Kompression, Verschlüsselung usw.)
  - Formatter: wie wird die Nachricht serialisiert (z.B. soap/wsdl, binary)?

1 Remotekonfigurationsdatei

- serveraktiviertes Objekt: Server (keine Referenz auf DLL mehr nötig!):

```
<application>
    <service>
        <wellknown mode="Singleton"
            type="BankLibrary.Bank, Bank"
            objectUri="MyURI.soap" />
    </service>
    <channels>
        <channel port=4711 type="http" />
    </channels>
</application>
```

- serveraktiviertes Objekt: Client

```
<application>
    <client>
        <wellknown type="BankLibrary.Bank, Bank"
            url="http://localhost:4711/MyURI.soap" />
    </client>
    <channels>
        <channel type="http" />
    </channels>
</application>
```


1 Remotekonfigurationsdatei

- clientaktiviertes Objekt: Server

```
<application>
    <service>
        <activated type="BankLibrary.Account, Bank" />
    </service>
    <channels>
        <channel port=4711 id="http" />
    </channels>
</application>
```

- clientaktiviertes Objekt: Client

```
<application>
    <client url="http://localhost:4711" >
        <activated type="BankLibrary.Account, Bank" />
    </client>
    <channels>
        <channel type="http" />
    </channels>
</application>
```


2 WSDL, SOAP und UDDI

- WSDL: Web Services Description Language
  - sprich: “wiz-dull”
  - XML-Schema zum Beschreiben von Web Services
  - Schnittstellendefinition
  - nur zustandslos, nur *call-by-value*

```
<message name='Account.depositInput'>
    <part name='amount' type='xsd:int'/>
</message>
<message name='Account.depositOutput'>
    <part name='return' type='xsd:int'/>
</message>
<portType name='AccountPortType'>
    <operation name='deposit' parameterOrder='amount'>
        <input name='depositRequest'
               message='tns:Account.depositInput'/>
        <output name='depositResponse'
               message='tns:Account.depositOutput'/>
    </operation>
</portType>
...
```

2 WSDL, SOAP und UDDI

- SOAP: Simple Object Access Protocol

```
<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Header> <t:transid>1234</t:transid>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <m:deposit>
      <amount xsi:type="integer">3</amount>
    </m:deposit>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

  - einfaches Request/Response-Protokoll (im Beispiel: Antwort des Server **`<m:depositResponse>`**)

- UDDI: Universal Description, Discovery and Integration
  - Art Namensdienst, SOAP-API zum Registrieren und Suchen von Diensten
  - Zurückgegeben werden URLs auf WSDLs
  - (freie) Implementierungen u.a. von Novell, Sun, SourceForge

3 Metadaten-Assembly

- Problem: Client benötigt Metadaten zur Erstellung des Proxies.
- Alternativen
  - Assembly muss auch auf Clientseite vorhanden sein (oft nicht erwünscht oder möglich).
  - entferntes Objekt von Interface (oder Basisklasse) ableiten:
    Interface-Assembly ist sowohl beim Client als auch beim Server vorhanden.
    - Nachteil: von einem Interface kann man kein Objekt erzeugen.
    - → keine clientaktivierten Objekte
    - → Serveraktivierte Ojekte müssen mittels **`Activator.GetObject`** angesprochen werden.
- Metadaten-Assembly:
  Assembly mit gleichem Namen und gleichen Klassen inkl. Methoden wie auf der Serverseite, jedoch ohne Implementierung
  - **`soapsuds.exe`**

3 Metadaten-Assembly: soapsuds auf Clientseite

- Auf Clientseite:

```
Win> soapsuds -url:http://localhost:4711/MyURI.soap?wsdl
     -oa BankProxy.dll
```

- erzeugt ein Proxy-Assembly, Proxy kann mittels **`new`** instantiiert werden, Serveradresse ist fest codiert (Client benötigt keine Remoting-Konfiguration)
  - "wrapped-Proxy"
  - Metadaten werden von der entfernten Anwendung geladen oder aus lokaler Assembly extrahiert.
  - nur mit Http-Channel und SOAPFormatter möglich
  - Aufruf beim Client einfach mit:

```
RemoteBank bank = new RemoteBank();
```

- Parameter **`-nowp`** erzeugt einen "nonwrapped"-Proxy

```
RemoteBank bank = (RemoteBank) Activator.GetObject(
    typeof(RemoteBank), "http://....soap");
```


3 Metadaten-Assembly: Proxy-Internas

- Wrapped-Proxy

```
public class RemoteBank :
    System.Runtime.Remoting.Services.RemotingClientProxy {
    public RemoteBank() {
        base.ConfigureProxy(this.GetType(),
            "http://localhost:1234/RemoteBank.soap");
    }

    public int deposit(int amount) {
        return ((RemoteBank) _tp).deposit(amount);
    }
}
```

- Nonwrapped-Proxy
  - erzeugt werden nur leere Klassendefinitionen
  - flexibler (Verwendung von Konfigurationsdateien für Channel, Adresse usw.)

3 Metadaten-Assembly: soapsuds auf Clientseite

- Alternative Ausgabeformate
  - C#:
    ```
    Win> soapsuds -url:.../MyURI.soap?wsdl -gc BankProxy.cs
    ```
  - WSDL:
    ```
    Win> soapsuds -url:.../MyURI.soap?wsdl -os bank.wsdl
    ```
- bei clientaktivierten Objekten:
  ```
  Win> soapsuds -url:http://localhost:4711/
         RemoteApplicationMetadata.rem?wsdl
         -oa BankProxy.dll
  ```
  - keine vordefinierte URI
  - **`RemoteApplicationMetadata.rem`** als spezielle Adresse
  - liefert Metadaten für *alle* verwendbaren Objekte inkl. “well-known” Objekte
  - Nachteil: zu den Objekten werden keine Informationen über mögliche Konstruktoren erzeugt (nur Standardkonstruktor möglich).

3 Metadaten-Assembly: soapsuds auf Serverseite

- auf Serverseite:
  ```
  Win> soapsuds -ia Bank -oa BankProxy.dll
  ```
- nur einzelne Typen erzeugen:
  ```
  Win> soapsuds
       -types:BankLibrary.Bank,Bank;BankLibrary.Account,Bank
       -oa BankProxy.dll
  ```
- Proxy mit fest codierter URL möglich:
  ```
  Win> soapsuds
       -types:BankLibrary.Bank,Bank,
          http://localhost:4711/MyURI.soap;
          BankLibrary.Account,Bank
       -oa BankProxy.dll
  ```


4 Proxies

- implementieren die Methoden und Attribute des entfernten Objekts,
- leiten Methodenaufrufe an entfernte Objekte weiter
- werden dynamisch erzeugt!
- dabei Aufteilung des Proxies in zwei Teile
- transparenter Proxy
  - Umsetzung der Methodenaufrufe
  - kann nicht verändert werden
- **`RealProxy`**
  - abstrakte Basisklasse, erweiterbar durch eigene Implementierung
  - default-Implementierung der .NET-Architektur: **`System.Runtime.Remoting.Proxies.RemotingProxy`**

4 Proxies: Transparenter Proxy

- wird beim Client transparent erzeugt, z.B.: durch **`Activator.GetObject`**
- besitzt nach außen dasselbe Interface wie das “richtige” Objekt
- Die Laufzeitumgebung kann die Anzahl und den Typ der Parameter prüfen.
- lokales Objekt: Methodenaufruf
- entferntes Objekt:
  - Argumente in **`IMessage`**-Objekt packen
    - enthält URI des entfernten Objekts, Name und Parameter der aufzurufenden Methode
  - mittels **`Invoke`** an den **`RealProxy`** übergeben

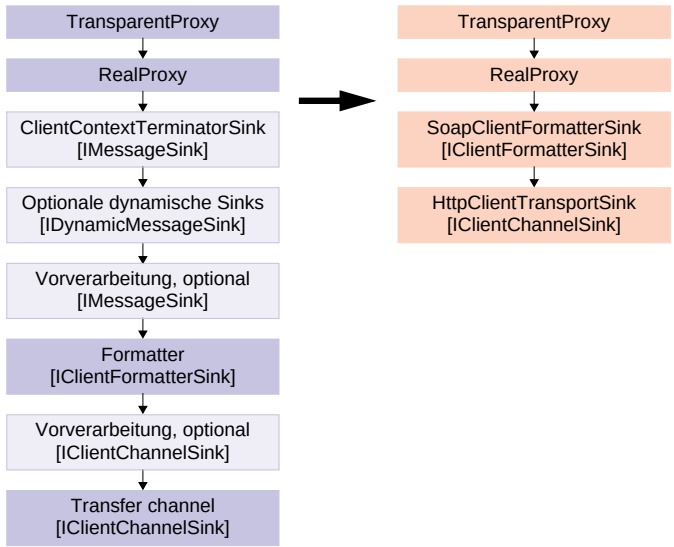
4 Proxies: RealProxy

- Klasse **`RealProxy`**
  - ebenfalls dynamisch erzeugt
  - kann erweitert und verändert werden (über ProxyAttribut, s. unten)

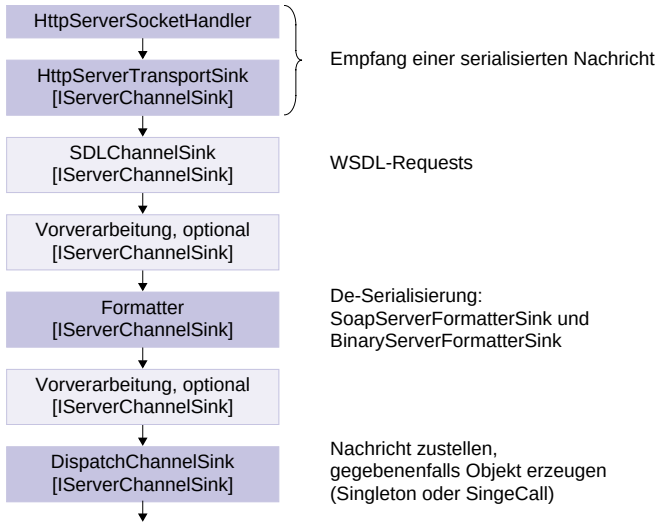
```
public class MyProxy : RealProxy {
    MarshalByRefObject _target;
    public MyProxy(Type type, MarshalByRefObject target) :
        base(type) {
        _target = target;
    }

    public override IMessage Invoke(IMessage msg) {
        // Standardimplementierung: msg an Kanal weitergeben
    }
}
```

- **`Activator.GetObject(...)`**
  - erzeugt einen transparenten Proxy für ein serveraktiviertes Objekt
  - keine Netzwerkinteraktion bei der Erstellung;
    erst wenn eine Methode am Proxy aufgerufen wird

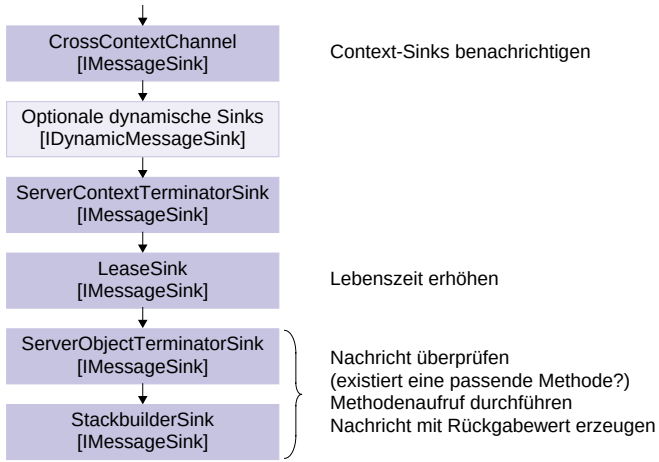
5 Sinks: Überblick Client



5 Sinks: Überblick Server



5 Sinks: Überblick Server-Side Messaging



5 Sinks

- Interface **IMessageSink**

```
public interface IMessageSink {
        IMessageSink NextSink { get; }
        IMessageCtrl AsyncProcessMessage(IMessage msg,
                        IMessageSink replySink);
        IMessage SyncProcessMessage(IMessage msg);
}
```

  - MessageSink empfängt Nachricht über (**A**)**SyncProcessMessage**.
  - verarbeitet die Nachricht
  - schickt die Nachricht an den **IMessageSink** der **NextSink**-Property weiter

- Beispiel: synchrone Nachrichtenübertragung

```
class MySink: IMessageSink {
        IMessageSink _next;
        IMessage SyncProcessMessage(IMessage msg) {
                // Nachricht msg verarbeiten
                IMessage ret = _next.SyncProcessMessage(msg);
                // Antwort ret verarbeiten
                return ret;
        }
}
```


5 Sinks: asynchrone Aufrufe

- Implementierung der Sinkkette
  - **AsyncProcessMessage()** kehrt sofort zurück.
  - Antwort wird über zweite Sinkkette verschickt (**replySink**-Parameter).

```
public class MyReplySink: IMessageSink {
    IMessageSink _nextSink;
    MyReplySink(IMessageSink next) {
        _nextSink = next;
    }
    IMessage SyncProcessMessage(IMessage msg) { ... }
}

public class MySink: IMessageSink {
    IMessageSink _nextSink;
    IMessageCtrl AsyncProcessMessage(IMessage msg,
                    IMessageSink replySink) {
        IMessageSink myRChain = new MyReplySink(replySink);
        return _nextSink.AsyncProcessMessage(msg, myRChain);
    }
}
```

- Die Antwort wird synchron verarbeitet, nur die Erzeugung der Nachricht geschieht asynchron auf dem Server.

5 Sinks: SinkProvider

- Wie werden Sinks und Sinkketten erzeugt?
- SinkProvider liefern bei Bedarf Sinks (**IClientChannelSinkProvider**, **IServerChannelSinkProvider**, **IClientFormatterSinkProvider**, **IServerFormatterSinkProvider**)
- Provider bilden ebenfalls eine Kette.
- In der Konfigurationsdatei wird festgelegt, welche Provider in der Kette vorhanden sein sollen.

5 Sinks: Erzeugung

- Das Channel-Objekt enthält einen Verweis auf die Providerkette.

```
IChannel chnl = ChannelServices.GetChannel("http");
```

- Bei Instantiierung eines serveraktivierten Objekts wird die Methode **CreateMessageSink()** an allen registrierten Channels aufgerufen, bis ein Channel-Objekt die angegebene URL (Parameter) akzeptiert.
  - Beim HTTP-Channel beispielsweise nur URLs, die mit “http:” oder “https” beginnen
- Der Channel ruft **CreateSink()** am ersten Sinkprovider auf, dieser dann (rekursiv) beim jeweils nächsten Provider.
- Im Anschluss erzeugen die Provider einen neuen Sink und liefern diesen zurück.
- Das TransparentProxy/RealProxy-Paar erhält eine Referenz auf die komplette Sinkkette.

6 Asynchrone Aufrufe

- Aufruf über Delegate mit gleicher Signatur wie die Methode, die asynchron aufgerufen werden soll

```
public class AsyncDemo {
    public string TestMethod(int dauer, out int id) {
        Thread.Sleep(dauer);
        id = AppDomain.GetCurrentThreadId();
        return "Dauer des Aufrufs: " + dauer.ToString();
    }
}
public delegate string AsyncDelegate(int dauer, out int id);
```

- Laufzeitumgebung definiert automatisch **`BeginInvoke()`** und **`EndInvoke()`**-Methoden für jeden Delegate.
  - **`BeginInvoke()`**: Aufruf initialisieren, liefert **`IasyncResult`** zurück

```
AsyncDemo ad = new AsyncDemo();
AsyncDelegate dlgt = new AsyncDelegate(ad.TestMethod);
IAsyncResult ar = dlgt.BeginInvoke(3000, out threadId,
    null, null);
```

  - **`EndInvoke()`**: blockiert, bis Aufruf bearbeitet wurde (muss immer aufgerufen werden)

6 Asynchrone Aufrufe

- Wie bekommt man mit, dass der Aufruf zu Ende ist?
  - blockierendes Warten: einfach **`EndInvoke()`** aufrufen
  - Pollen: **`IAsyncResult`** (Rückgabewert **`BeginInvoke()`**) abfragen über **`IsCompleted`**-Property
  - Callback: Delegate für Callback-Methode bei **`BeginInvoke()`** übergeben

```
static void Main(string[] args) {
    AsyncDemo ad = new AsyncDemo();
    AsyncDelegate dlgt = new AsyncDelegate(ad.TestMethod);
    IAsyncResult ar = dlgt.BeginInvoke(3000, out threadId,
        new AsyncCallback(CallbackMethod), dlgt);
    ...
}

static void CallbackMethod(IAsyncResult ar) {
    AsyncDelegate dlgt = (AsyncDelegate)ar.AsyncState;
    dlgt.EndInvoke(out threadId, ar);
}
```


6 Asynchrone Aufrufe

- Asynchrone One-Way Aufrufe
  - Ausführung ist nicht garantiert.
  - keine Rückgabewerte oder out-Argumente
  - über Delegates implementiert; **`EndInvoke()`** kehrt immer sofort zurück
  - Server

```
public class DistributedWhiteboard : MarshalByRefObject {
  [OneWay()]
  public void Refresh() { ... }
  ...
```

  - Client

```
delegate void RefreshDelegate();

DistributedWhiteboard wboard = (DistributedWhiteboard)
  Activator.GetObject(typeof(DistributedWhiteboard), ...);
RefreshDelegate rdlg = new RefreshDelegate(wboard.Refresh);
IAsyncResult svAsyncres = rdlg.BeginInvoke(null, null);
rdlg.EndInvoke(svAsyncres);
```

  - Keine Exceptions, auch wenn Server nicht erreichbar ist

7 Parameterübergabe

- Wie werden Parameter übergeben?
  - Wertparameter (call-by-value)

```
void op(int n) { ... }
```

    bei Fernaufrufen ebenfalls call-by-value (Argumente serialisieren)
  - Variablenparameter (call-by-reference)

```
void op(ref int n) { ... }
```

    bei Fernaufrufen: *call-by-value-result*
    Zunächst Kopie übergeben; das Original wird überschrieben, sobald die Methode zurückkehrt.
  - Variablenparameter, evtl. nicht belegt (call-by-reference)

```
void op(out int n) { ... }
```

    bei Fernaufrufen: *call-by-result*
    Das Original wird überschreiben, sobald die Methode zurückkehrt.

8 Kontext

- Beispiel: Einschränkungen einer Methode
  - Password setzen: minimale Länge 8 Zeichen

```
public void SetPassword(string passwd) {
    if (passwd.Length < 8) throw new Exception("too short");
    ...
```

  - Nachteil: nicht sichtbar in Methodensignatur (Interface!)
  - Besser: Attribut

```
[Check(Length>7)]
public void SetPassword(string passwd) {
```

  - Die Überprüfung muss von einer Instanz zwischen Aufrufer (Client) und (Server-)Objekt durchgeführt werden: **`ContextBoundObject`**
- Anwendungen
  - z.B. Logging, Sicherheit, Transaktionen
  - Synchronisierung, z.B. nur ein Thread darf in diesem Kontext aufgeführt werden, Objekte selbst müssen sich nicht um Synchronisierung kümmern

8 Kontext

- Ableiten von **`System.ContextBoundObject`**
  - Die Laufzeitumgebung prüft vor der Aktivierung eines Kontext-gebundenen Objekts, ob der Kontext gewechselt werden muss: **`IContextAttribute.IsContextOK()`**
  - Falls nicht, wird mit **`IContextAttribute.GetPropertiesForNewContext()`** die erforderliche Umgebung erzeugt.
- Ein Application Domain kann mehrere Kontexte enthalten, mindestens den “default context”.
  - Kapselung innerhalb einer Application Domain, kontrollierte Ausführung, auch bei Fernaufrufen
  - **`ContextBoundObject`** ist von **`MarshalByRefObject`** abgeleitet

8 Kontext

- Beispiel CheckAttribute

```
namespace ContextBound {
    [AttributeUsage(AttributeTargets.Class)]
    public class CheckAttribute : ContextAttribute {
        public override bool IsContextOK(Context ctx,
            IConstructionCallMessage ctor) {
            return false;
        }
        public override void GetPropertiesForNewContext(
            IConstructionCallMessage ctor) {
            ctor.ContextProperties.Add(
                new CheckContextProperty());
        }
}
```

- CheckContextProperty erzeugt beispielsweise neuen Sink
  - Alle Aufrufe laufen über diesen Sink
  - Attribute abfragen mit Reflection (**`GetCustomAttributes()`**, **`GetParameters()`** auf Message-Objekt)

8 Kontext: Dynamische Sinks

- Dynamische Sinks sind mit einem bestimmten Kontext verknüpft.
  - werden aufgerufen, wenn in einen anderen Kontext gewechselt wird

```
public class MyDynamicSinkProvider :
    IDynamicProperty, IContributeDynamicSink {
        public string Name {
            get { return "MyName"; }
        }
        public IDynamicMessageSink GetDynamicSink() {
            return new MyDynamicSink();
        }
    }
}

public class MyDynamicSink : IDynamicMessageSink {
    public void ProcessMessageStart(IMessage reqMsg,
        bool bCliSide, bool bAsync) { ... }
    public void ProcessMessageFinish(IMessage replyMsg,
        bool bCliSide, bool bAsync) { ... }
}
```

  - Registrierung

```
Context ctx = Context.DefaultContext;
IDynamicProperty prp = new MyDynamicSinkProvider();
Context.RegisterDynamicProperty(prp, null, ctx);
```

8 Kontext: ProxyAttribute

- Beispiele: Persistente Objekte, automatische Caches ...

```
[PersistentObject("XYZ-Kopie")]
public class XYZ : ContextBoundObject
{
    ...
}
```

- Proxy-System verwenden
  - Bei der Instantiierung des Objekts wird automatisch ein Proxy erzeugt (auch im lokalen Fall!).
  - Dieser kann Konstruktor- (**`new()`**) und Methodenaufrufe abfangen.

8 Kontext: ProxyAttribute

- Implementierungen des Attributs und des Proxies

```
[AttributeUsage(AttributeTargets.Class)]
public class PersistentObjectAttribute : ProxyAttribute
{
  public PersistentObjectAttribute(string tableName) { ... }
  public override MarshalByRefObject
    CreateInstance(Type srvType) {
    PersistenceProxy proxy = new PersistenceProxy(srvType);
    return (MarshalByRefObject)proxy.GetTransparentProxy();
  }
}

public class PersistenceProxy : RealProxy
{
  public PersistenceProxy(Type type) : base(type) { }
  public override IMessage Invoke(IMessage msg) {
    if (msg is IConstructionCallMessage) {
      // z.B. Objektzustand aus Datei laden
    } else if (msg is IMethodCallMessage) {
      ...
    } else if (msg is IMethodReturnMessage) {
      ...
    }
  }
}
```


9 Lease-based Garbage Collector

- DCOM:
  - Verweiszähler
  - Durch “pings” werden nicht erreichbare Clients entdeckt.
  - Finden sog. “greedy clients”, die absichtlich oder versehentlich eine Referenz auf das Objekt behalten
- .NET: lease-based GC:
  - Jedem *Marshal by Reference*-Objekt wird eine bestimmte Zeit (*Lease*) eingeräumt, während der es nicht gelöscht wird.
  - Durch die Nutzung des Objekts kann die Lease verlängert werden.
  - wenn nach Ablauf der Lease eine Methode an dem Objekt aufgerufen wird:
    - Singleton: neues Objekt wird erzeugt
    - clientaktiviertes Objekt: RemotingException

9 Lease-based Garbage Collector

- Lease-based GC wird für Singleton und clientaktivierte Objekte verwendet.
- **`MarshalByRefObject.GetLifetimeService()`** liefert ein Objekt vom Typ **`ILease`** mit Informationen zur Lebenszeit:

```
public class myClass : MarshalByRefObject {
   public example() {
      ILease myLease = (ILease)this.GetLifetimeService();
   }
}
```

9 Lease-based Garbage Collector

- Eigenschaften des **`ILease`** Interfaces:
  - **`TimeSpan CurrentLeaseTime`**
    (nur lesbare) verbleibende Zeit, bis das Objekt gelöscht werden kann
  - **`TimeSpan InitialLeaseTime`**
    Zeit, die ein Objekt nach der Erzeugung bekommt.
    (Standard: 5 Minuten)
  - **`TimeSpan RenewOnCallTime`**
    Zeit, die ein Objekt nach einem Aufruf mindestens noch am Leben bleibt
    (Standard: 2 Minuten)
  - **`LeaseState CurrentState`**
    Zustand der Lease: **`Active`**, **`Expired`**, **`Initial`**, **`Null`** oder **`Renewing`**
- Methoden:
  - **`void Register(ISponsor)`**
    einen Sponsor registrieren

9 Lease-based Garbage Collector

- Konfiguration der Lease-Zeiten
  - für alle Objekte einer Anwendung: durch Remotekonfigurationsdatei

```
<application>
    <lifetime leaseTime="10s" renewOnCallTime="5s" />
    <service>
        <activated type="BankLibrary.Account, Bank" />
    </service>
    <channels>
        <channel port=4711 type="http" />
    </channels>
</application>
```

  - für einzelne Objekte: **`InitializeLifetimeService`** überschreiben

```
public override object InitializeLifetimeService() {
    ILease leaseInfo =
                (ILease)base.InitializeLifetimeService();
    leaseInfo.InitialLeaseTime = TimeSpan.FromSeconds(7);
    leaseInfo.RenewOnCallTime = TimeSpan.FromSeconds(3);
    // unendliche Laufzeit: InitialLeaseTime=TimeSpan.Zero
    //                oder: return null;
    return leaseInfo;
}
```


9 Lease-based Garbage Collector - Sponsoren

- Ist die Lease abgelaufen, so werden alle Sponsoren gefragt, ob die Lease verlängert werden soll.
- Sponsor implementiert das Interface **`ISponsor`**.
  - **`public TimeSpan Renew(ILease leaseInfo)`**
    Wenn 0 zurückgegeben wird, so wird der Sponsor von der Liste entfernt.
- Registrieren mittels **`ILease.Register(ISponsor)`**
- weitere Einstellungsmöglichkeiten:

```
<application>
    <lifetime leaseTime="10s"
              renewOnCallTime="5s"
              sponsorshipTimeout="5s"
              leaseManagerPollTime="10s"/>
</application>
```

  - **`sponsorshipTimeout`**: Wartezeit auf Antwort eines Sponsors (2 Min.)
    = **`TimeSpan.Zero`**: Die Lease nimmt keine Sponsoren an.
  - **`leaseManagerPollTime`**: Überprüfung der Leases (10 Sekunden)