Das Businessobjekt-Framework CSLA.NET einsetzen

Baukasten fürs Business

Was braucht jede halbwegs ausgewachsene Geschäftsapplikation? Zum Beispiel solche Dinge wie Validierung, Authentifizierung, Autorisierung, einen Undo-Mechanismus, Status-Tracking und manches mehr. Und genau dafür gibt es Frameworks. Eines davon ist CSLA.NET.

Auf einen Blick

Thomas Kehl entwickelt seit mehr als 15 Jahren Businesssoftware auf verschiedenen Systemen, angefangen bei Clipper bis hin zur .NET-Plattform. Er ist Mitinhaber der Firma PP Software AG in der Schweiz und leitet dort die Abteilung Softwareengineering. Sie erreichen ihn unter t.kehl@ppsoftware.ch.

Inhalt

- Die Fünf-Schichten-Architektur von CSLA.NET einsetzen.
- Businessobjekte anlegen.
- Mobile Objekte für die Kommunikation zwischen den Schichten nutzen.

Serie

1. **Businessobjekte mit CSLA.NET erstellen.**
2. Den Data-Access-Layer an CSLA.NET-Businessobjekte anbinden.
3. Businessobjekte mittels WPF und MVVM im Presentation-Layer darstellen.

dnpCode

A1101CSLANET

Diese dreiteilige Artikelserie befasst sich mit dem Businessobjekt-Framework CSLA.NET. Das Kürzel steht für Component-based Scalable Logical Architecture. CSLA.NET wurde vor mehr als zehn Jahren von Rockford Lhotka erfunden und wird seitdem stetig weiterentwickelt [1]. Die .NET-Implementierung startete bereits 1999. Bis dahin fand die Entwicklung in VB 6 statt. Es dient dem Zweck, ein solides Framework für die Entwicklung von Businessobjekten zur Verfügung zu stellen. Es unterstützt die Entwicklung skalierbarer Applikationen für alle Microsoft-UI-Plattformen.

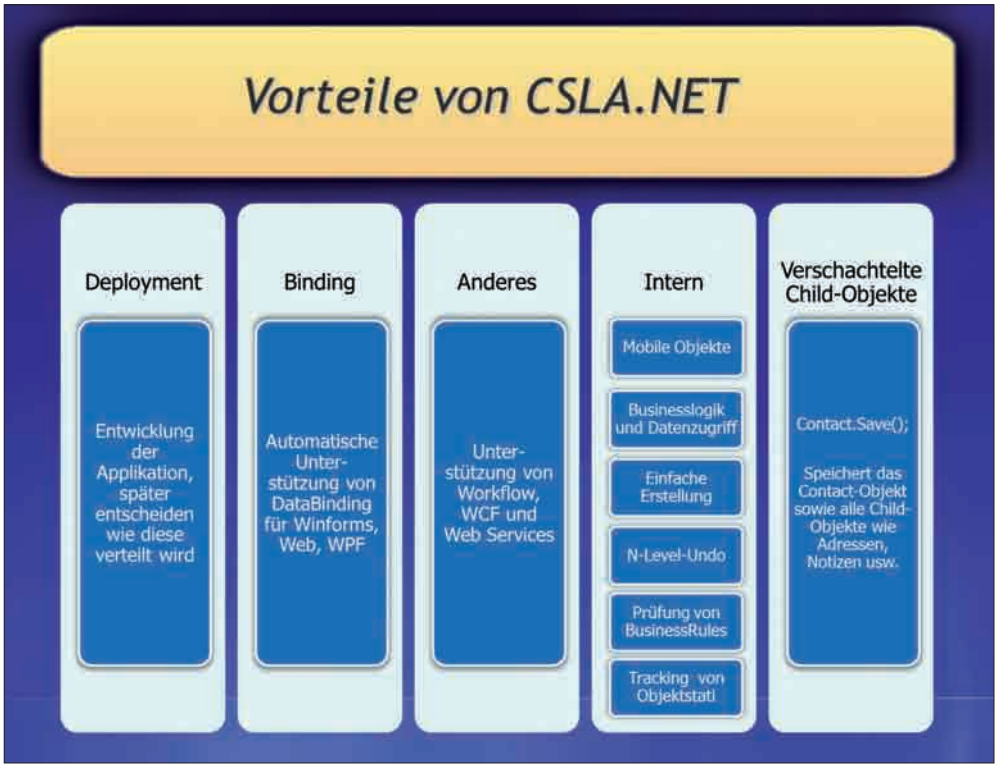
CSLA.NET vereinfacht und standardisiert insbesondere die Umsetzung von Geschäftslogik sowie Validierung und Autorisierung der Objekte. Es bietet zudem volle Unterstützung von DataBinding, N-Level Undo, Authentifizierung, abstrakte Speicherung von Daten sowie mobile Objekte, um Funktionalität über die physischen Grenzen einzelner Rechner hinaus auszuführen.

Diese Serie behandelt folgende Bereiche:

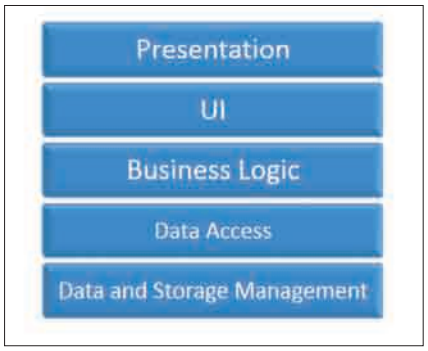
1. Teil: Aufbau und Funktion des Frameworks,
2. Teil: Versorgung der Businessobjekte mit Daten aus beliebigen Datenspeichern,
3. Teil: Anbindung der View über das MVVM-Pattern an die Businessobjekte.

Wozu braucht man ein Businessframework? Die Welt der Businessapplikationen mit ihren zum Teil tausenden Objekten wird immer komplexer. Bei der Entwicklung solcher Applikationen stellen sich bezüglich der Infrastruktur immer wieder die gleichen Fragen und Anforderungen. Die Anwendung soll etwa beliebig skalierbar sein und Mehrschichtenarchitekturen unterstützen. Die Businesslogik soll unabhängig vom Datenspeicher sowie von der Präsentation sein.

Zudem soll sie fehlertolerant sein und doch diverse Sicherheitsaspekte implementieren – insbesondere Business- sowie Validierungsregeln, Benutzerauthentifizierung und integrierte Autori-



[Abb. 1] Vorteile von CSLA.NET.



[Abb. 2] Fünf-Schichten-Architektur.

sierungsregeln. Vielfach werden weitere Mechanismen benötigt wie etwa N-Level Undo oder ein Status-Tracking der Änderung von Businessobjekten. CSLA.NET erfüllt alle diese Anforderungen, siehe Abbildung 1.

Welche ist die richtige Architektur? Wenn man eine Gruppe von Softwarearchitekten versammelt und jeden bittet, seine ideale Architektur zu beschreiben, wird fast jeder ein anderes Bild skizzieren. Für CSLA.NET wird die Verwendung einer Fünf-Schichten-Architektur propagiert, siehe Abbildung 2. Dies stellt jedoch keine Bedingung für die Verwendung von CSLA.NET dar.

Verwaltung der Businesslogik

In der einen oder anderen Art und Weise verwendet oder interagiert jede dieser Schichten mit Applikationsdaten. Am offensichtlichsten ist dies bei den Schichten Data-Access und Data and Storage Management. Die Businessschicht muss die Daten validieren, berechnen und manipulieren. Die UI-Schicht muss die Daten zwischen den Schichten Business Logic und Presentation hin- und hertransportieren. Zusätzlich formatiert der UI-Layer die Daten oder trifft anhand der Daten Entscheidungen bezüglich der Navigation innerhalb der Anwendung. Der Presentation-Layer zeigt dem Benutzer die Daten an und nimmt neue Daten von ihm entgegen.

In einer idealen Welt existiert die komplette Businesslogik in der Businessschicht. Dies ist in der Realität meist nicht möglich. In einer webbasierten Applikation sind beispielsweise oft Validierungen im Presentation-Layer integriert. Aber jede Validierung, die im Webbrowser durchgeführt wird, ist unzuverlässig, da es leicht ist, sie zu umgehen. Deshalb muss jede im Browser durchgeführte Validierung im Business-Layer nochmals überprüft werden.

Analog verlangen die meisten Datenbanken Datenintegrität sowie die Einhaltung weiterer Regeln. Dies stellt auch Businesslogik dar. Zudem enthält der Data-Access-Layer oft Logik, um zu entscheiden, wann und wie Daten in einer Datenquelle gespeichert beziehungsweise geladen werden müssen. Es ist in fast jeder Anwendung der Fall, dass Businesslogik mehr oder weniger über alle Schichten verstreut ist.

Businessobjekte

Die Problematik der über verschiedene Schichten verteilten Businesslogik kann mit dem Einsatz von Businessobjekten gelöst werden. Das primäre Ziel beim Design jeder Art von Softwareobjekten ist die Schaffung einer abstrakten Repräsentation von Einheiten oder Konzepten.

In ADO.NET zum Beispiel repräsentiert die *DataTable* eine tabellarische Menge von Daten. *DataTables* stellen einen abstrakten und konsistenten Mechanismus zur Verfügung, um mit tabellarischen Daten zu arbeiten. Ebenso ist eine Windows-Forms-*TextBox* ein Objekt, welches das Konzept der Darstellung und Eingabe von Daten repräsentiert. Aus der Perspektive der Applikation ist es nicht nötig zu wissen, wie das Control auf dem Bildschirm gezeichnet wird oder wie der Benutzer mit dem Control interagiert. Es ist einfach ein Objekt, das eine Eigenschaft *Text* sowie mehrere interessante Events besitzt.

Der Schlüssel zu erfolgreichem Objektdesign ist die Kapselung. Das heißt, dass ein Objekt eine Black Box ist, die Daten und Logik enthält. Der Benutzer des Objekts muss nicht wissen, welche Daten dieses beinhaltet oder wie die Logik funktioniert.

Wenn Objekte abstrakte Darstellungen von Einheiten oder Konzepten sind, die Daten und damit in Verbindung stehende Logik kapseln, was sind dann Businessobjekte?

Objektorientierte Applikationen werden entwickelt, um die eine oder andere Sorte von Problemen zu lösen. Um dies zu tun, wird häufig eine Vielzahl verschiedener Objekte verwendet. Einige dieser Objekte haben keinen unmittelbaren Zusammenhang mit dem zu lösenden Problem. Zum Beispiel sind die Objekte *DataTable* oder *TextBox* nur eine abstrakte Darstellung von Computerkonzepten.

Andere Objekte hingegen haben eine starke Verbindung zum Geschäftsfeld, für das die Applikation entwickelt wird. Bei diesen Objekten handelt es sich um Businessobjekte. Sie beinhalten das Verhalten beziehungsweise die Logik des abstrahierten Objektes sowie die Daten, die für die Logik benötigt werden. Für die Interaktion stellt das Objekt ein definiertes Interface zur Verfügung.

Mobile Objekte

Objektorientierte Applikationen werden vielfach unter der Annahme konzipiert, dass alle Objekte in einer Anwendung miteinander ohne Performanceverlust interagieren können. Dies gilt insbesondere dann, wenn sämtliche Objekte im selben Prozess auf demselben Computer ausgeführt werden.

Diese Annahme trifft jedoch nicht zu, wenn Objekte in verschiedenen Prozessen oder sogar auf verschiedenen Computern ausgeführt werden. Weiter oben wurde bereits beschrieben, dass CSLA.NET für den Betrieb in einer Fünf-Schichten-Architektur konzipiert ist. Dies gilt insbesondere auch für Architekturen, die auf verschiedenen Rechnern ausgeführt werden. Um für eine hohe Skalierbarkeit zu sorgen, kann das zum Beispiel ein Client, ein Applikationsserver sowie ein Datenserver sein. Bei einer Webclientarchitektur ist meistens zusätzlich noch ein Webserver im Spiel. Auf jeder dieser Maschinen werden Teile der Anwendung ausgeführt. Diese wiederum sollten mit allen anderen Teilen interagieren können.

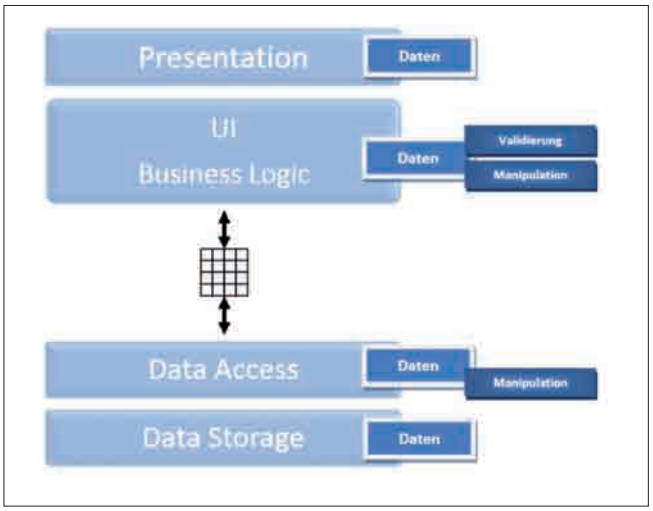
Solche verteilten Architekturen können nicht mehr mit den klassischen objektorientierten Konzepten aufgebaut werden, denn jede Kommunikation zwischen Objekten auf verschiedenen Maschinen erzeugt Netzwerktraffic. Dies führt vielfach zu Performanceproblemen, die nicht einfach ignoriert werden können.

Um diese Probleme zu überwinden, werden die meisten verteilten Applikationen nicht komplett objektorientiert konzipiert. Stattdessen bestehen sie aus mehreren Teilen prozeduralen Codes, der auf jeder Maschine läuft.

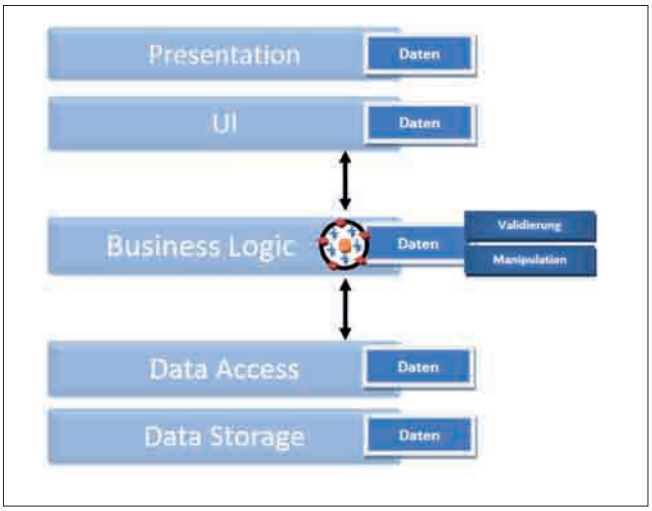
Die Daten befinden sich dabei in *DataSets*, Arrays oder XML-Dokumenten, die

Mobilität der Objekte

Businessobjekte von CSLA.NET lassen sich in der Regel effizienter über das Netzwerk übertragen als *DataSets*. CSLA.NET verwendet für die Serialisierung ein binäres System, das nur etwa 30 % der Kapazität benötigt, die ansonsten für *DataSets* erforderlich ist. Außerdem enthalten die Businessobjekte weniger Metadaten als ein *DataSet*, was die Menge der Bytes, die über das Netzwerk übertragen werden, weiter reduziert.



[Abb. 3] Zwischen Business- und Data-Layer wird ein DataSet übertragen.



[Abb. 4] Businesslogik, zentralisiert in einem Businessobjekt.

von Maschine zu Maschine gereicht werden.

Das soll nicht bedeuten, dass objektorientierte Designs in verteilten Umgebungen nicht verwendet werden – sie komplizieren die Anforderungen aber meistens. Um die Komplexität zu minimieren, sind die meisten Anwendungen innerhalb einer Schicht objektorientiert aufgebaut. Zwischen den Schichten benutzen sie zumeist prozedurale oder serviceorientierte Ansätze. Das Endergebnis ist, dass solche Applikationen als Ganzes weder objektorientiert noch prozedural sind, sondern eine Mischung aus beiden Welten.

Abbildung 3 zeigt, wie die Architektur für solche Anwendungen vielleicht am häufigsten aufgebaut ist. Es gibt einen Data-Access-Layer, der die Daten von der Datenbank in ein *DataSet* liest. Das *DataSet* wird zum Client (oder Webserver) zurückgegeben. Die Präsentation agiert direkt mit dem *DataSet*.

Dieser Ansatz hat große Mängel bezüglich Wartung und Wiederverwendung von Code. Aber die Performance ist in den meisten Fällen sehr gut. Zudem sind die meisten Softwareentwickler recht vertraut mit dem Schreiben von Code, der *DataSets* verwendet und manipuliert. Das bedeutet, dass der Code gut zu verstehen ist, und das beschleunigt die Entwicklung.

Die Entscheidung für einen solchen Ansatz sollte mit Vorsicht getroffen werden. Bei diesem Ansatz ist es sehr einfach, das objektorientierte Design zu kompromittieren, indem Daten aus einem Objekt, das auf Maschine 1 läuft, über das Netzwerk an ein anderes Objekt gesendet werden. Damit wird es möglich, dass andere Objekte außerhalb des Kontexts der Businesslogik Daten verwenden und ändern können. Das bricht aber die Kapselung des logischen Business-Layers auf.

Im Gegensatz dazu stehen Konzepte, die mobile Objekte von einer Maschine zur anderen statt nur Rohdaten versenden. Diese Objekte kapseln, zusammengenommen, den kompletten Business-Layer. Durch die Technologien von WCF, Serialisierung und Deployment beinhaltet das .NET Framework direkte Unterstützung für das Konzept mobiler Objekte. Dank dieser Möglichkeiten kann der Data-Access-Layer (der auf einem Applikationsserver läuft) Businessobjekte erstellen und sie mit Daten aus der Datenbank laden. Diese Objekte können dann zum Client (oder Webserver) gesendet werden, wo sie vom Presentation-Layer verwendet werden können. Abbildung 4 verdeutlicht dieses Konzept.

In dieser Architektur werden intelligente Daten in Form von Businessobjekten anstatt Rohdaten zum Client gesendet. Somit kann die Präsentation die gleiche Businesslogik verwenden wie der Data-Access. Dies reduziert den Aufwand für die Wartung der Applikation, da nicht ein Teil der Businesslogik im Data-Access-Layer und ein anderer Teil im Presentation-Layer vorhanden sind. Stattdessen wird die Businesslogik in einem separaten Layer konsolidiert. Diese Businessobjekte können sich nun über das Netzwerk bewegen. Im Gegensatz zum *DataSet*-Ansatz beinhalten diese Objekte die Daten und die zugehörige Businesslogik – etwas, das ein *DataSet* nicht so einfach bieten kann.

In der Praxis wird der Business-Layer meist aufgeteilt zwischen der Maschine, auf der der Data-Access-Layer läuft, und derjenigen, auf welcher der Presentation-Layer läuft. Mit der Unterstützung mobiler Objekte ist das eine ideale Lösung: Sie forciert die Wiederverwendung von Code, erzeugt geringe Wartungskosten und eine hohe Leistungsfähigkeit.

Stereotypen

CSLA.NET stellt unter anderem Basisimplementationen für folgende Stereotypen zur Verfügung, siehe Abbildung 5:

- *BusinessBase* – Editable root-/child-Objekt,
- *ReadOnlyBase* – Read only root-/child-Objekt,
- *BusinessListBase* – Editable root-/child-Liste,
- *ReadOnlyListBase* – Read only root-/child-Liste,
- *NameValueListBase* – Name/value-Liste,
- *CommandBase* – Command.

Businessobjekt im Detail

Businessobjekt-Stereotypen sind das Herz jeder CSLA.NET-Anwendung – ob Silverlight- oder .NET-basiert. Jedes Businessobjekt hat Abschnitte, die Businessfunktionalität implementieren, wie etwa:

- Businesslogik,
- Properties,
- Validation-Rules,
- Authorization-Rules,
- Factory-Methoden,
- Data-Access-Methoden.

Listing 1 zeigt eine Standarddeklaration eines CSLA-Businessobjekts. Grundsätzlich muss immer von einer CSLA-Basis-

Listing 1

Ein Businessobjekt deklarieren.

```
[Serializable]
public class Company : BusinessBase<Company>
```

klasse geerbt werden, damit die Funktionalität von CSLA.NET zur Verfügung steht. Zudem müssen alle Objekte mit dem *Serializable*-Attribut markiert werden, um die Funktionalität für mobile Objekte zu ermöglichen.

Listing 2 zeigt die Definition einer Property in einem Businessobjekt. Wie man sieht, vereinfacht CSLA.NET die Deklaration von Properties. Das *RegisterProperty*-Pattern sollte jedem vertraut sein, der bereits *CustomControls* in WPF erstellt hat. Es funktioniert ähnlich wie die *Dependency-Properties* von WPF. Das *RegisterProperty*-Pattern in CSLA.NET bietet die Möglichkeit, Properties zu verwalten. Dies ergibt viele Vorteile einschließlich des Transfers von Daten durch das DataPortal mittels mobiler Objekte. Trotz seiner scheinbar einfachen Syntax führt *SetProperty()* eine Reihe von Funktionen für den Entwickler aus wie:

- Ausführung von Business-Rules sowie Validation-Rules,
- Prüfung von Zugriffsberechtigungen, basierend auf Authorization-Rules,
- Verwaltung des Objektstatus, zum Beispiel der *dirty*-Markierung, wenn sich eine Property ändert,
- Auslösung des *PropertyChanged*-Events für das DataBinding.

Alle diese Funktionen stellt CSLA.NET „out of the box" zur Verfügung.

Business- und Validation-Rules

Ab CSLA.NET 4 ist neu, dass Business- und Validation-Rules als Klassen implementiert sind. Bisher waren sie als Methoden implementiert. Eine Business-Rule-Klasse muss das Interface *IBusinessRule* implementieren. Es ist jedoch einfacher, von der Klasse *BusinessRule*, welche bereits ein Spektrum von Grundfunktionalität im Zusammenhang mit Business-Rules zur Verfügung stellt, zu erben. Diese Rules-Objekte werden dann mit Properties des Businessobjektes verknüpft.

Die Klasse *BusinessBase* enthält Code, der die Regeln an den entsprechenden Stellen prüft. Zum Beispiel werden Regeln für eine Property geprüft, wenn sich die Property ändert oder wenn sich eine abhängige Property ändert. Sie werden auch geprüft, wenn eine neue Instanz eines Businessobjekts erstellt wird. Zudem kann die Prüfung auch explizit im Code aufgerufen werden.

Normalerweise werden Rules mit Businessobjekten nach Typ verknüpft. Dies bedeutet, dass sämtliche Instanzen eines bestimmten Businessobjekttyps die gleichen Rules mit den gleichen Properties verknüpft haben. Dies ist optimal, weil damit der Aufwand für die Verknüpfung der Regeln nur einmal vorgenommen werden muss und diese Informationen über sämtliche Instanzen hinweg verwendet werden. Das spart sowohl Prozessorzeit als auch Arbeitsspeicher.

Als Alternative kann man eine Rule nur mit einzelnen Instanzen verknüpfen. Das bedeutet, dass jede Instanz eines Businessobjekttyps ihre eigenen Rules besitzt, die unterschiedlich sein können. Die Verknüpfungen werden beim Erzeugen des Objektes erstellt und im Speicher gehalten. Dies kann sich jedoch negativ auf die Perfor-

Listing 2

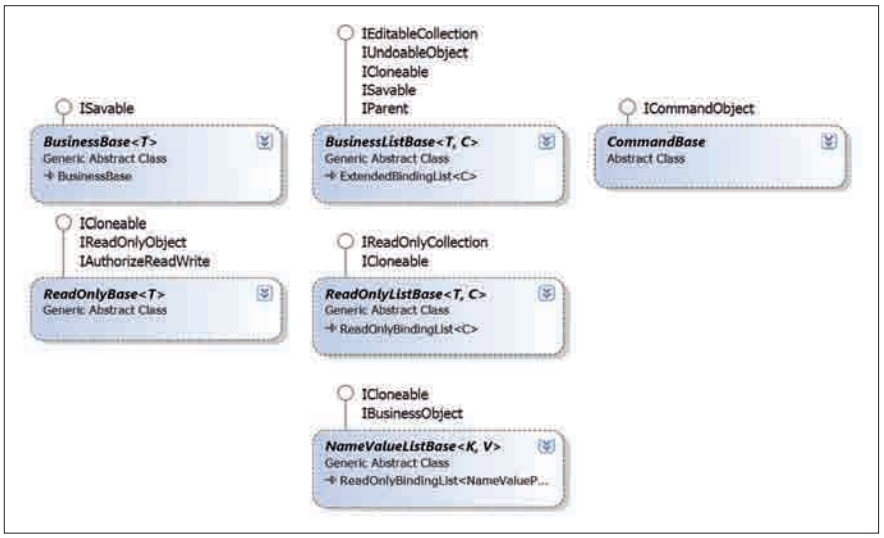
Eine Property für ein Businessobjekt anlegen.

```
private static PropertyInfo<string> CompanyNameProperty = RegisterProperty<string>(
    new PropertyInfo<string>(
    "CompanyName",
    "Company Name",
    string.Empty));

public string CompanyName {
  get { return GetProperty(CompanyNameProperty); }
  set { SetProperty(CompanyNameProperty, value); }
}
```

Listing 3

Eine Rule definieren.

```
private class MyRule : Csla.Rules.BusinessRule {
  protected override void Execute(RuleContext context) {
    var target = (Customer)context.Target;
    if (target.Sales < 10) {
      context.AddInformationResult("Customer has low sales");
    }
  }
}

protected override void AddBusinessRules() {
  base.AddBusinessRules();
  BusinessRules.AddRule(new MyRule { PrimaryProperty = SalesProperty });
}
```



[Abb. 5] Übersicht über die wichtigsten Stereotypen von CSLA.NET.

Listing 4

Authorization-Rules definieren.

```
protected override void AddAuthorizationRules() {
  string[] canWrite = { "AdminUser", "RegularUser" };
  string[] canRead = { "AdminUser", "RegularUser", "ReadOnlyUser" };
  string[] admin = { "AdminUser" };
  AuthorizationRules.AllowCreate(typeof(Company), admin);
  AuthorizationRules.AllowDelete(typeof(Company), admin);
  AuthorizationRules.AllowEdit(typeof(Company), canWrite);
  AuthorizationRules.AllowGet(typeof(Company), canRead);
  AuthorizationRules.AllowWrite(CompanyNameProperty, canWrite);
  AuthorizationRules.AllowWrite(CompanyIdProperty, canWrite);
  AuthorizationRules.AllowWrite(DateAddedProperty, canWrite);
  AuthorizationRules.AllowRead(CompanyNameProperty, canRead);
  AuthorizationRules.AllowRead(CompanyIdProperty, canRead);
  AuthorizationRules.AllowRead(DateAddedProperty, canRead);
}

[Serializable]
public class RolodexPrincipal : BusinessPrincipalBase
{ /* ... */ }

private static void SetPrincipal(Csla.Security.CslaIdentity identity)
{
  RolodexPrincipal principal = new RolodexPrincipal(identity);
  Csla.ApplicationContext.User = principal;
}
```

Listing 5

Ein Businessobjekt laden.

```
public static Company GetCompany(int Id) {
  return DataPortal.Fetch<Company>(new SingleCriteria<Company, int>(Id));
}

protected void DataPortal_Fetch(SingleCriteria<Company, int> criteria) {
  using (SqlConnection connection = new SqlConnection(DataConnection.ConnectionString))
{
    connection.Open();
    /* Data-Access Here */
    LoadProperty<int>(CompanyIdProperty, reader.GetInt32("CompanyID"));
    LoadProperty<string>(CompanyNameProperty, reader.GetString("CompanyName"));
    LoadProperty<SmartDate>(DateAddedProperty, reader.GetSmartDate("DateAdded"));
    /* Load Children Here */
  }
}
```

mance auswirken und zu einem hohen Speicherverbrauch führen. Dies gilt vor allem dann, wenn es von einem bestimmten Typ viele Instanzen gibt.

Rules, die das aktuelle Objekt als ungültig kennzeichnen, werden zur *BrokenRuleList* hinzugefügt. Sobald die jeweilige Regel erfüllt ist, wird die Rule aus dieser Liste entfernt. Jedes editierbare Businessobjekt hat immer eine Liste mit aktuell ungültigen Rules. Wenn diese Liste leer ist, so ist die Property *IsSelfValid* des Businessobjekts (geerbt von *BusinessBase*) gleich *true*, andernfalls gleich *false*. Diese Liste wird auch für die Implementierung von *IDataErrorInfo* für das DataBinding verwendet. Die Liste der Rules, die angeben, ob ein Objekt ungültig ist, ist als Property des Businessobjektes verfügbar. Somit kann das Objekt selbst für DataBinding verwendet und dem Benutzer der Applikation verfügbar gemacht werden. Eine solche Rule definiert auch, ob es sich um einen Fehler, eine Warnung oder eine Information handelt. Somit können die Hinweise dem Benutzer dementsprechend angezeigt werden, und die Applikation kann richtig reagieren.

CSLA.NET stellt ein Set an Standard-Rules zur Verfügung, die mit Properties verbunden werden können. Dieses beinhaltet zum Beispiel:

- *MaxLength/MinLength*,
- *MaxValue/MinValue*,
- *Required*,
- *RegExMatch*.

Für sehr aufwendige Vorgänge – insbesondere wenn Servercalls beteiligt sind – bietet CSLA.NET auch die Möglichkeit, Rules asynchron zu prüfen. Somit wird der Arbeitsfluss des Benutzers nicht behindert.

Listing 3 zeigt beispielhaft, wie Sie eine Rule in CSLA.NET definieren können. Zudem zeigt das Beispiel, wie Sie die Rule in einem Businessobjekt mit einer Property verbinden können. Dazu müssen Sie im Businessobjekt, das von *BusinessBase* erbt, die Methode *AddBusinessRules()* überschreiben.

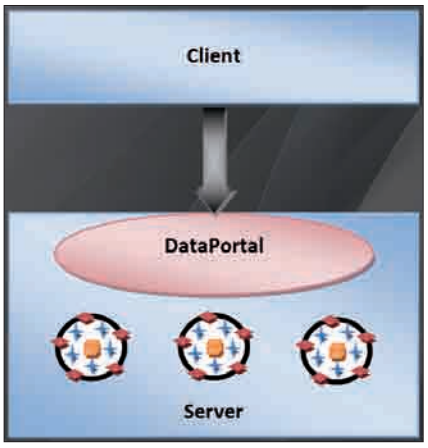
Authorization-Rules

Die Definition von Authorization-Rules in CSLA.NET ist einfach. Definieren Sie dazu, welche Rolle welche Operationen ein bestimmtes Businessobjekt ausführen darf, siehe Listing 4. Authorization wird gegen den aktuellen *Principal*, der in *ApplicationContext.User* gesetzt wurde, geprüft. Um einen *Principal* für CSLA.NET zu erstellen, muss ein *Principal*-Businessobjekt erstellt werden.

Wenn sich dann ein Benutzer einloggt, kann der *Principal* wie jedes andere Businessobjekt ermittelt und als aktueller Benutzer im *ApplicationContext* gesetzt werden.

DataPortal

Die Unterstützung von Objektpersistenz kann sehr komplex sein. Die Businessobjekte in CSLA.NET kapseln die Datenzugriffslogik in sich selbst, oder sie delegieren das Datenzugriffsverhalten an ein Persistenzobjekt. Gleichzeitig möchte man nicht das Businessobjekt abändern müssen, falls sich die physische Architektur ändert. CSLA.NET bietet zudem die Möglichkeit, leicht umzuschalten, ob der Data-Access-Layer auf dem Client oder dem Applikationsserver läuft. In CSLA.NET ist dazu lediglich eine Einstellung in einer *app.config*-Datei nötig. Wenn ein Applikationsserver eingesetzt wird, sollte nicht jedes Busi-



[Abb. 6] Das DataPortal als „single point of entry".

nessobjekt direkt auf den Server zugreifen. Dies wäre ein Wartungs- und Konfigurationsalbtraum. Es würde jedes Mal, wenn ein Businessobjekt hinzugefügt oder geändert wird, auf allen Clientrechnern eine Aktualisierung der Konfigurationsdaten bedeuten. Stattdessen wäre es ideal, wenn es eine einheitliche Eintrittsstelle zum Applikationsserver gäbe, sodass jeder Client einfach konfiguriert werden kann.

Wie in Abbildung 6 dargestellt, unterstützt das DataPortal von CSLA.NET diesen „single point of entry" und die Konfiguration für den Server. Es verwaltet die Kommunikation mit Businessobjekten, damit der Data-Access-Layer auf dem Server läuft.

Außerdem bietet das DataPortal-Konzept noch weitere Vorteile:

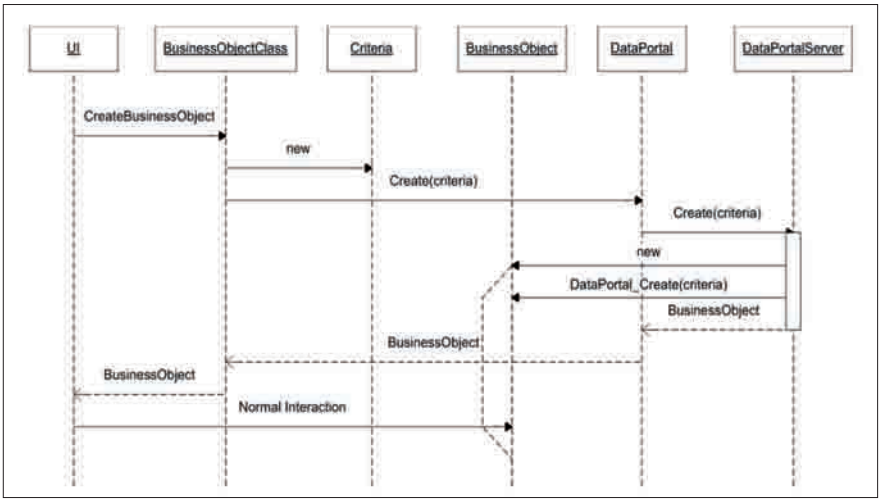
- Zentralisierte Sicherheit bei Aufrufen auf dem Applikationsserver.
- Einen einheitlichen Objektpersistenzmechanismus.
- Abstraktion des Netzwerktransports zwischen Client und Server (Unterstützung von Remoting, Webservices, Enterprise Services, WCF und zukünftigen Protokollen.)
- Einen Ort, an dem eingestellt wird, ob der Data-Access-Code lokal oder zum Beispiel über Remoting läuft.

Das DataPortal von CSLA.NET verwendet Factory-Methoden, die in den Businessobjekten implementiert werden müssen. Das Factory-Pattern ist ein Standardmuster, mit dem Konsumenten Objekte erstellen können. Um Daten in ein Businessobjekt laden zu können, müssen die verschiedenen *DataPortal_XYZ()*-Methoden im Businessobjekt implementiert werden. Um zum Beispiel ein *Company*-Objekt zu laden, muss zuerst eine Methode *DataPortal_Fetch()* erstellt werden. Diese kann ent-

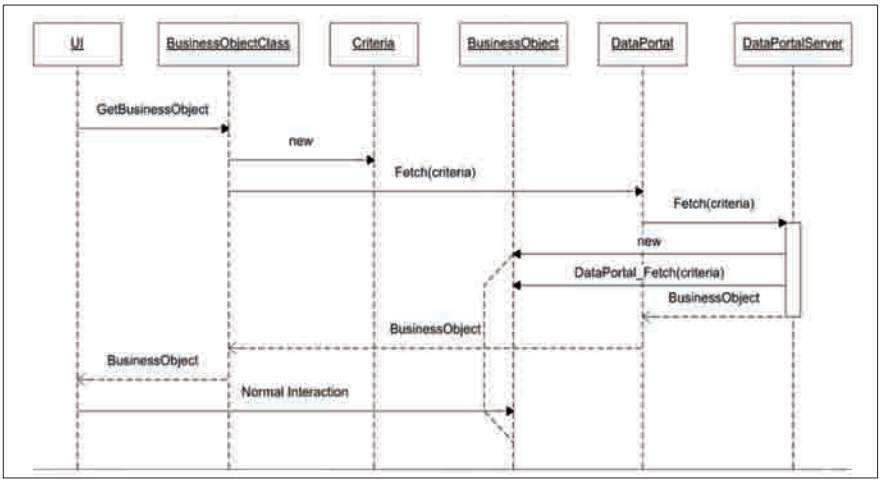
Listing 6

DataBinding in WPF.

```
<Grid x:Name="LayoutRoot" DataContext="{StaticResource CompanyData}">
    <Grid.RowDefinitions>
      <RowDefinition Height="35"/>
      <RowDefinition Height="35"/>
      <RowDefinition Height="35"/>
      <RowDefinition Height="35"/>
      <RowDefinition Height="*"/>
      <RowDefinition Height="35"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="150"/>
      <ColumnDefinition Width="200"/>
      <ColumnDefinition Width="20"/>
    </Grid.ColumnDefinitions>
    <TextBlock Text="Company Name:" TextAlignment="Right"
     HorizontalAlignment="Right" Grid.Column="0" Grid.Row="0" Margin="6,6,6,6"/>
    <TextBox x:Name="CompanyNameTextbox" Grid.Column="1" Grid.Row="0"
     HorizontalAlignment="Stretch" Margin="6,6,6,6" Text="{Binding Path =
                          Data.CompanyName, Mode=TwoWay}" IsReadOnly="True"/>
    <csla:PropertyStatus RelativeTargetName="CompanyNameTextbox"
     Property="CompanyName" Source="{Binding Path=Data}" Grid.Column="2"
     Grid.Row="0" HorizontalAlignment="Left"/>
```



[Abb. 7] Ablauf für DataPortal_Create().



[Abb. 8] Ablauf für DataPortal_Fetch().

Listing 7

Ein UI-Wrapper für Businessobjekte.

```
<csla:CslaDataProvider
 x:Key="CompanyData"
 ManageObjectLifetime="True"
 IsInitialLoadEnabled="False"
 ObjectType="Company, Library ..."
 PropertyChanged=
    "CslaDataProvider_PropertyChanged"
 FetchFactoryMethod="GetCompany"
 CreateFactoryMethod="CreateCompany"
 DataChanged=
    "CslaDataProvider_DataChanged"
/>
```

weder von der Basisklasse überschrieben werden, oder es kann eine eigene *DataPortal_Fetch()*-Methode erstellt werden, die exakt das gleiche *Criteria*-Objekt erwartet, das dem DataPortal in der Factory-Methode übergeben wurde. Ist dies der Fall, ruft CSLA.NET die korrekte Überladung der *DataPortal_Fetch()*-Methode auf.

Das Laden eines Businessobjekts vom Typ *Company* zeigt Listing 5. Die Abbildungen 7 und 8 zeigen exemplarisch den Ablauf für *DataPortal_Create()* beziehungsweise *DataPortal_Fetch()*.

Die verschiedenen Möglichkeiten zur Anbindung eines Data-Access-Layers werden im nächsten Teil der Serie betrachtet.

Der Presentation-Layer

Grundsätzlich macht CSLA-NET beim Presentation-Layer keine Vorgaben. Ein Vorteil von CSLA.NET ist aber die integrierte Unterstützung für DataBinding – etwa für WPF.

Wenn zum Beispiel eine *TextBox* an ein Property *CompanyName* gebunden werden soll, kann die *Text*-Property der *TextBox* direkt an die zugrunde liegende Datenquelle, die durch ein CSLA.NET-Businessobjekt repräsentiert wird, gebunden werden. Dies ist deswegen sehr einfach, da die Stereotypen von CSLA.NET die Mechanismen für Data-Binding von Haus aus implementieren. Veranschaulicht wird dies in Listing 6.

Alle Controls werden an die statische Ressource *CompanyData* gebunden. Konkret wird das *Grid* an diese Ressource gebunden, und alle Controls spezifizieren nur noch den Namen der Property, an die sie gebunden werden sollen. Listing 7 zeigt die dazugehörende Ressource.

CslaDataProvider ist ein Control aus dem CSLA.NET-Framework, das als UI-Wrapper für Businessobjekte agiert. Es ermöglicht die vereinfachte Syntax von XAML und reduziert erheblich die Notwendigkeit von Code-behind. Zudem kann dieses Control automatisch Daten ins Objekt laden. Das Objekt, das verwendet werden soll, kann auch angegeben werden. Dafür muss ein qualifizierter Assemblyname verwendet werden. Die Methode, die für das Laden oder die Erstellung des Objekts verwendet werden soll, kann ebenfalls direkt im Control spezifiziert werden (Factory-Methode für *Fetch* beziehungsweise für *Create*). Zusätzlich können Events registriert werden, die das Objekt auslöst. Der *DataProvider* weiß, wie er Änderungen mittels *Save()* oder *Cancel()* speichern oder verwerfen kann.

Wie Listing 8 zeigt, kann der *DataProvider* auch mit dem *InvokeMethod*-Objekt zusammenarbeiten.

InvokeMethod kann einem Control hinzugefügt werden, um bestimmte Control-Events abzuhören. Abhängig von diesem Event kann *InvokeMethod* spezifische Methoden eines anderen Objekts aufrufen. Im Beispiel von Listing 8 wird *InvokeMethod* zu einem *Save*-Button hinzugefügt. Wenn der Benutzer den Button anwählt, wird die *Save()*-Methode von *CompanyData*, dem *CslaDataProvider*-Objekt, aufgerufen. *InvokeMethod* kann natürlich auch für andere Zwecke beziehungsweise für den Aufruf von beliebigen Methoden zur Laufzeit verwendet werden. Allerdings agiert *InvokeMethod* besonders gut mit dem *CslaDataProvider*-Objekt, insbesondere um Controls abhängig von der Datenquelle zu aktivieren oder zu deaktivieren. Im Beispiel wird nur aktiviert, wenn der Benutzer Änderungen an den Daten vorgenommen hat. Zusätzlich wird *IsEnabled* an die Property *CanSave* des *CslaDataProviders* gebunden. Dies bewirkt, dass der Button nur dann aktiv ist, wenn das Objekt *CompanyData* gültig ist sowie Änderungen vorgenommen wurden, die gespeichert werden müssen.

Listing 8

InvokeMethod einem Save-Button hinzufügen.

```
<Button
 x:Name="SaveButton"
 Content="Save"
 csla:InvokeMethod.MethodName="Save"
 csla:InvokeMethod.Resource="{StaticResource CompanyData}"
 csla:InvokeMethod.TriggerEvent="Click"
 IsEnabled="{Binding Source={StaticResource CompanyData}, Path=CanSave, Mode=OneWay}" />
```

Listing 9

Hinweise visualisieren.

```
<csla:PropertyStatus
 RelativeTargetName="CompanyNameTextbox"
 Property="CompanyName"
 Source="{Binding Path=Data}"
/>
```

Abschließend sei noch das Control *PropertyStatus* erwähnt. Dieses Control ermöglicht die Visualisierung von Hinweisen im Zusammenhang mit bestimmten Eigenschaften des Businessobjekts. Konkret können zum Beispiel ToolTips mit Hinweisen zu nicht erfüllten Business-Rules angezeigt werden. Oder es können Animationen für asynchrone Vorgänge visualisiert werden. Listing 9 veranschaulicht dieses Control an einem Beispiel.

Ausblick

Dieser erste Teil der dreiteiligen Serie gab einen groben Überblick über die Architektur von CSLA.NET. Im nächsten Teil der Serie wird das Augenmerk auf die verschiedenen Möglichkeiten, die CSLA.NET zur Anbindung eines Data-Access-Layers bietet, gelegt. Sie werden sehen, welche Möglichkeiten mit dem DataPortal von CSLA. NET gegeben sind. Es ist sehr einfach, ein beliebiges Persistenzsystem anzubinden – beziehungsweise ein solches auszuwechseln.

Im dritten Teil der Serie wird das Hauptaugenmerk auf der Anbindung des Presentation-Layers an die Businessobjekte liegen. Insbesondere wird dort WPF in Kombination mit dem MVVM-Pattern zum Einsatz kommen. **[ml]**

[1] Rockford Lhotka – Entwickler von CSLA.NET, www.lhotka.net