AUTOMATISIERUNG VON ANWENDUNGEN

Potenziale und Missverständnisse

Eine Anwendung mit Skriptmöglichkeit zu versehen will gut überlegt sein. dotnetpro zeigt, worauf es dabei ankommt.

Die ersten einem breiten Publikum verfügbaren Anwendungen, die ein Nutzer mithilfe einer Skriptsprache an seine Bedürfnisse anpassen konnte, waren Microsofts Büroprogramme Word und Excel. Mit Visual Basic for Applications (VBA) war es möglich, tief in die Prozesse für das Erstellen und Manipulieren von Dokumenten einzugreifen, sodass sich der ursprüngliche Leistungsumfang dieser Anwendungen erweitern und individualisieren ließ. Es dürfte kaum ein Unternehmen geben, das in seinen internen organisatorischen Prozessen mit der Verarbeitung von Dokumenten zu tun hat und nicht eine der beiden Anwendungen verwendet, um sie an seine individuellen Bedürfnisse anzupassen.

Das Spektrum reichte hier von einfachen Berechnungen und spezialisierten Formeln in der Tabellenkalkulation bis hin zur Anbindung an die Unternehmensdatenbanken einschließlich der Aufbereitung von Daten für die Serienbrieffunktionen von Word. Die starke, bis heute bestehende Kundenbindung wurde auch dadurch erreicht, dass man beide Anwendungen bis zur Unkenntlichkeit an die eigenen Bedürfnisse anpassen konnte.

Word und Excel spielen aufgrund ihrer zentralen Bedeutung als Werkzeuge zum Bearbeiten von Dokumenten und Tabellen für den organisatorischen Prozess eines Unternehmens natürlich eine Sonderrolle. Wenn heute Anwendungen für Unternehmen entwickelt werden, können sie selten von einer so großen Reichweite ausgehen. Dennoch kann es sinnvoll sein, über die Frage der Automation nachzudenken, also die Möglichkeit, individuelle Bedürfnisse mit einer leicht anpassbaren Skriptsprache abzudecken.

Für Geschäftsanwendungen stehen zwei grundlegende Strategien zur Auswahl, wie an die Umsetzung funktionaler Anforderungen herangegangen werden soll. Auf der einen Seite steht die Prämisse, dass jedes gewünschte Merkmal genau so umgesetzt wird, wie es ein Kunde wünscht. Dieser direkte Weg erlaubt es, Aufwand und Nutzen unmittelbar gegenüberzustellen. Hier kann sich der Kunde wiederfinden, denn er hat das, was er möchte, direkt vor Augen.

Auf der anderen Seite steht der Ansatz, dass die Anwendung ein Set von Basisfunktionen bietet, das sich durch die Möglichkeit zur Automation ergänzen lässt. Das ist dann sinnvoll, ▶

wenn die Notwendigkeit oder der Wunsch nach hoher Anpassbarkeit besteht – wenn beispielsweise Fachabteilungen spezielle Anforderungen auf der Basis von allgemeinen Anforderungen des Unternehmens haben und diese sehr spezifisch sind und sie zu realisieren viel Aufwand nach sich ziehen würde.

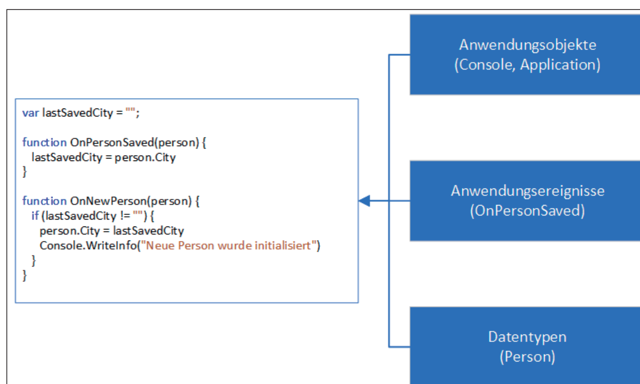
Nicht selten führen eine hohe Dynamik und Segmentierung in den Anforderungen zu einem Qualitätsverlust der Anwendung. Automation könnte hier ein Kompromiss sein, der einerseits einen grundlegenden Funktionsumfang liefert und darüber hinaus die Möglichkeit zu individuellen Anpassungen bietet.

Arten der Automation

Doch zunächst müssen zwei Arten von Automatisierung unterschieden werden. Bei der ersten und umfassenderen Art handelt es sich um die wiederholbare Abfolge von Nutzeraktionen. Eine typische Form der Spezifikation ist hier das Makro, das – wie beispielsweise bei Word oder Excel –, in Form einer Skriptsprache definiert wird. Es gibt jedoch auch Formen, die eine stärkere grafische Interaktion mit dem Anwender erlauben. Dann werden die einzelnen Befehle als Listeneinträge oder als Bestandteil eines Flowcharts angesehen, die dann dem Zweck entsprechend hierarchisiert, positioniert und konfiguriert werden können.

Eine andere Art der Automatisierung ist der Eingriff in Arbeitsabläufe der Anwendung. Dabei geht der Impuls nicht von der Ausführung eines Makros aus, sondern vom Anwender, der eine bestimmte Aktion innerhalb der Anwendung ausführt. Innerhalb dieser Aktion gibt es einen oder mehrere Erweiterungspunkte, die eine Manipulation des Arbeitsablaufs mithilfe eines Skripts erlauben, das an dieser Stelle mit den jeweils passenden Parametern ausgeführt wird. Wie weit eine Anwendung die Manipulation ihrer Arbeitsabläufe ermöglicht, spiegelt sich in der Definition dieser Erweiterungspunkte wider.

Hier ergibt sich ein sehr breites Spektrum, das vom Setzen von Eigenschaften bis hin zur komplexen Manipulation der Gesamtdaten einer Anwendung reicht. Dieser Artikel befasst sich ausschließlich mit der zweiten Variante der Automation. Die Integration von Erweiterungspunkten, die mithilfe einer Skriptsprache Arbeitsabläufe manipulieren, ist der erste und einfachere Ansatz.



Die Elemente beim Gestalten eines API (Bild 1)

Was dagegen spricht

Gerade weil der Einsatz einer Skriptsprache für Entwickler so verlockend ist, ist es wichtig, sich auch die Argumente vor Augen zu führen, die dagegen sprechen. Dabei ist zunächst einmal die Zielgruppe der Anwendung zu betrachten. Jede Skriptsprache – und sei sie noch so einfach – erfordert ein gewisses Maß an Einarbeitung. Die Lust dazu haben in der Regel nur sehr wenige Anwender, die ansonsten mit Dokumenten, Tabellen und Geschäftsprozessen zu tun haben. Es muss also sichergestellt werden, dass die Anwendung und deren Funktionen auch dann funktionieren, wenn kein Skript erstellt wird. Eine Anwendung, die einen Mechanismus zur Automation voraussetzt, wird sich nicht etablieren, da sich die Zielgruppe dann automatisch auf diejenigen reduziert, die ein gewisses Maß an Affinität für Technik mitbringen.

Ein weiterer Aspekt ist die Frage, ob sich eine individuelle Funktionalität nicht als Nutzerkonfiguration dialogbasiert erreichen lässt. Das folgende Gedankenspiel soll die dahinterliegende Abwägung veranschaulichen: Angenommen, es soll ein Datensatz bearbeitet werden. Wenn ein Feld aktiv ist und der Wert bearbeitet wird, soll der aktuelle und noch nicht bearbeitete Wert so ausgegeben werden, dass der Anwender ihn in der Zeit, in der er den Wert bearbeitet, sehen kann. Damit hat er die Möglichkeit, sowohl den neuen Wert als auch den veränderten Wert betrachten zu können. Da für einige Anwender diese Funktionalität hilfreich ist, während sie andere stört, wird sie so implementiert, dass der Anwender sie aktivieren oder deaktivieren kann.

Es könnte sich jedoch herausstellen, dass dies einigen Anwendern noch zu wenig ist und sie die Ausgabe des vorherigen Wertes wünschen, solange der Datensatz nicht gespeichert wird. Dann müsste der vorherige Wert immer dann angezeigt werden, wenn ein Feld aktiv ist, und dieser vorherige Wert so zwischengespeichert werden, wenn andere Felder aktiv sind. Es könnten sich auch Aspekte ergeben, die es erforderlich machen, dass diese Funktion in Abhängigkeit von den Daten erfolgen soll. Wenn der Datensatz aufgrund spezieller Werte eine Priorität hat, die sich aus betriebswirtschaftlichen Aspekten ergibt und die ein unterschiedliches Verhalten hinsichtlich der Anzeige der vorherigen, noch nicht gespeicherten Daten erfordert, wird sich das kaum noch durch einen Konfigurationsdialog abbilden lassen, wenn diese Einstellungen nutzerspezifisch sein sollen.

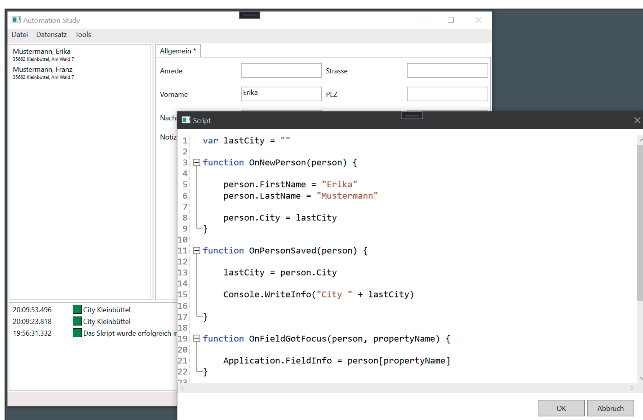
Je mehr Parameter eine Option in einer Anwendung hat, desto schwieriger ist deren Visualisierung. Eine Skriptsprache könnte dabei eine Alternative darstellen, weil sie die Komplexität der Bedingungen hinter den Methoden und Eigenschaften der dabei verwendeten Objekte verbirgt. Daraus lassen sich gut Anforderungen an die Gestaltung der Elemente der Skriptsprache ableiten, denn der Aufwand, die hier als Beispiel beschriebene Funktionalität umzusetzen, darf niemals den notwendigen Aufwand überschreiten, eine komplexe Oberfläche zu verstehen.

Der dritte und letzte Punkt in der Reihe der Gegenargumente ist weniger ein Gegenargument als eine Herausforderung, die sich aus der Idee ergibt, die Anwendung mit einer Möglichkeit zur Automation zu versehen. Die Verwendung

einer Programmiersprache zum Entwickeln einer Anwendung oder von Teilen daraus erfordert eine sehr spezielle Art des Denkens. Ein Verständnis für Begriffe wie Steuerstrukturen, Funktionen und Variablen ist dafür ebenso erforderlich wie ein strukturiertes Arbeitsgedächtnis, mit dem sich eine Abfolge von Befehlen so ordnen lässt, dass am Ende die gewünschte Funktionalität entsteht.

Für Softwareentwickler ist diese Art des Denkens selbstverständlich. Personen, die Projekte verantworten, haben dieses Verständnis in der Regel nicht. Das müssen sie auch nicht, da ihre Verantwortung bei der übergeordneten Koordination und der daraus folgenden Steuerkompetenz liegt. Hier zählt nur das Konkrete, das, was der Anwender sieht und nutzen kann. Anwendungsautomation steht dem entgegen, denn es ist zunächst einmal nicht mehr als ein Versprechen auf einen Werkzeugkasten, mit dem der Nutzer später seine Probleme lösen kann.

Das bedeutet, dass hier in etwas investiert wird, das weder konkret noch durch den Anwender unmittelbar nutzbar ist, da dazwischen immer noch Schulungen mit Einführungen in die Technik stehen. Wenn der Impuls, Anwendungsautomation zu verwenden, von der Entwicklung ausgeht, sind Aspekte wie diese unbedingt zu beachten. Es gibt jedoch zwei Argumente, welche die Automation auch für Verantwortliche verlockend erscheinen lassen. Zum einen lässt sich damit Zeit sparen, und zum anderen gewinnt die Anwendung deutlich an Flexibilität.



Die Beispielanwendung ClearScriptAppStudy (Bild 2)

Eines der wesentlichen Probleme ist in jedem Projekt die Frage, ob die umgesetzten Funktionalitäten dem entsprechen, was der Anwender will. Häufig entzündet sich die Frage an Details, deren Klärung viel Zeit kostet. Bietet hingegen eine Anwendung die Möglichkeit zur Automation, muss gar nicht mehr jeder Kundenwunsch im Kern erfüllt werden. Stattdessen lässt sich seine Erfüllung auf die lokale Ausführung eines Skripts verlagern, das sich später umsetzen lässt.

In diesem Zusammenhang spielt auch die Frage der bereitgestellten Programmiersprache eine Rolle. Die für die Beispielanwendung verwendete Komponente ClearScript [1] unterstützt deren drei: JScript, VBScript und JavaScript. Da JavaScript über die von Google federführend entwickelte Engine V8 unterstützt wird, gibt es kaum eine bessere und zukunftsweisendere Basis für die Möglichkeit, eine Anwendung mit Automatisierung auszustatten. Während VBScript und JScript und der dahinterliegende Windows Scripting Host vor allem Anwender adressieren, die administrative Vorgänge vereinfachen wollen, ist der Basisumfang von JavaScript zunächst einmal nur die Programmiersprache.

Über VBScript und JScript kann man einfach auf Systemobjekte wie *FileObject* oder die WMI zugreifen, bei JavaScript müssen diese Objekte explizit hinzugefügt werden, sodass die Ausführungsumgebung selbst darüber entscheidet, wie weit sich das System öffnet. Das ist ein wichtiger Aspekt, damit die Automatisierung einer Anwendung nicht zum Einfallstor für Sicherheitsvorfälle wird.

API-Gestaltung

Voraussetzung für die Akzeptanz sowohl auf der Seite der zukünftigen Anwender als auch auf der Seite der Verantwortlichen ist ein hochwertiges und leicht zu vermittelndes API. Hochwertig bedeutet in diesem Zusammenhang, dass alle Bestandteile über einen leicht verständlichen Weg miteinander in Verbindung gebracht werden können. Bild 1 zeigt die Elemente dieses Zusammenspiels. An erster Stelle steht die Anwendung selbst. Das ist der Rahmen, den der Anwender sieht und in dessen Kontext er sich bewegt. Ihre Eigenschaften eröffnen den Zugang zu den unterschiedlichen Elementen der Anwendung. Beispielsweise kann man Text auf der Statuszeile ausgeben oder das Erscheinungsbild der verschiedenen Oberflächenelemente ändern.

Welche Elemente hier als steuerbar angeboten werden, bestimmt die Flexibilität, mit der sich mit einem Skript Erscheinungsbild und Verhalten der Anwendung manipulieren lassen. Steht beispielsweise lediglich die Möglichkeit zur Verfügung, Text in der Statuszeile auszugeben, wird der Anwender nicht viel damit machen können. Gibt es darüber hinaus jedoch die Möglichkeit, mit der Farbe von Texten oder Hintergründen bestimmte Oberflächenelemente hervorzuheben, kann der Anwender variantenreicher und individueller auf bestimmte Anwendungsereignisse reagieren. Hier gilt es ein vernünftiges Maß zu finden.

Auf der einen Seite steht die Verbindlichkeit der Anwendung in Bezug auf Verhalten und Erscheinungsbild, auf der anderen Seite das ausgeprägte Interesse des Anwenders, seine Arbeitsumgebung entsprechend seiner Bedürfnisse manipulieren zu können.

Ein weiterer Aspekt der Gestaltung der Kommunikation zwischen Anwendung und Skript betrifft die Frage, wie bestimmte Zustände aus der Anwendung heraus an das Skript gegeben werden. Die direkte – und hier in den Beispielen verwendete – Variante basiert auf Konventionen für Funktionsnamen. Das bedeutet, es gibt eine Funktion im Skript *OnNewPerson()*, die beim Erstellen eines neuen Datensatzes aufgerufen wird, sofern sie existiert.

Der Vorteil ist, dass diese Variante leicht umzusetzen ist, schließlich bedarf es nicht mehr als einer Reihe von Namen für die jeweiligen Zustände. Der Nachteil ist, dass der Anwender diese Namen kennen muss, das Skript entsprechend ▶

Listing 1: Einbindung der Komponente ScriptService

```
public partial class App : PrismApplication
{
  protected override void RegisterTypes(
      IContainerRegistry containerRegistry)
  {
    containerRegistry
      .Register(typeof(MainWindow))
      .Register(typeof(MainWindowViewModel));

    // Dienste registrieren
    containerRegistry
      .RegisterSingleton(typeof(ScriptService))
      .Register<IScriptDialogs>(container =>
        container.Resolve<ScriptService>())
      .Register<IPersonMethods>(container =>
        container.Resolve<ScriptService>())
      .Register<IFieldMethods>(container =>
        container.Resolve<ScriptService>());
    // Dialoge registrieren
    containerRegistry.RegisterDialog
      <ScriptDialogView, ScriptDialogViewModel>();
    containerRegistry.RegisterDialogWindow
      <DialogWindow>();
  }

  protected override void InitializeShell(
      Window shell)
  {
    var settingsScript = new SettingsManager
      <ApplicationScript>(
      "ClearScriptAppStudy.json");

    this.Container.Resolve<IScriptDialogs>().
      Script = settingsScript.LoadSettings();

    base.InitializeShell(shell);

    App.Current.MainWindow = shell;
    App.Current.MainWindow.Show();
  }
}

public class MainWindowViewModel : BindableBase
{
  private readonly IScriptDialogs scriptDialogs;
  private ICommand showScriptDialogCommand;

  public MainWindowViewModel(
      IScriptDialogs scriptDialogs)
  {
    this.scriptDialogs = scriptDialogs;
  }

  public ICommand ShowScriptDialogCommand =>
    showScriptDialogCommand ??=
    new DelegateCommand(OnShowScriptDialog);

  private void OnShowScriptDialog()
  {
    scriptDialogs.ShowScriptDialog();
  }
}
```

vorinitialisiert wird oder es eine Form gibt, mit der der Anwender diese Namen auswählen kann.

Der dritte und letzte Bestandteil sind die bereitgestellten Datentypen. Mit der Entscheidung für JavaScript als Programmiersprache und ClearScript als ausführenden Container steht der volle Spracheumfang von JavaScript zur Verfügung, ergänzt um einige Konstruktionen, die ClearScript selbst umsetzt. Dabei sind weniger Steuerstrukturen wie Iterationen oder bedingte Ausführung der kritische Punkt, als vielmehr die Datentypen zur Verwaltung lokaler Daten. Beispielsweise ist das Erstellen eines Dictionarys mit JavaScript zwar möglich, für den Laien jedoch schwer zu lesen und zu verstehen. Einfacher wäre es, einen .NET-Datentyp zur Verfügung zu stellen, der eine fehlertolerante, Thread-sichere und einfache Form eines Dictionarys zur Verfügung stellt. Ob dabei .NET-eigene Datentypen oder anwendungsspezifische verwendet werden, spielt keine Rolle. Natürlich wäre es immer besser, für typische Anwendungsszenarien Datentypen bereitzustellen, die das Skript einfacher und damit leserlicher machen, ohne dabei an Funktionalität einzubüßen.

Integration in die Anwendung

Die Codefragmente zu diesem Artikel sind einer Beispielanwendung entnommen (Bild 2), welche die Verwendung von Automation in einer Anwendung demonstriert [1]. Sie ist eine auf WPF basierende Desktop-Anwendung, für die Prism als MVVM-Framework und Unity als Dependency-Container verwendet wurden. Zusätzlich schlug ClearScript in der Version 7.0 die Brücke zur JavaScript-Engine V8. Diese Version unterscheidet sich von ihrer Vorgängerin dadurch, dass sie die Verteilung stark vereinfacht. Alle Komponenten wurden als NuGet-Pakete eingebunden, sodass die Ausführung leicht zu realisieren sein sollte.

Das Beispiel ist eine typische WPF-Anwendung, die mit den Bestandteilen View, ViewModel und Dienste modularisiert ist [2]. Ob dabei letztlich wie in diesem Fall Prism als MVVM-Framework und Unity als Dependency-Container verwendet werden, sollte für das Verständnis der Automation keine Rolle spielen. Da die Funktionsweise beider Komponenten ähnlich der anderer Frameworks ist, werden sich die Beispiele leicht übertragen lassen. ▶

Listing 2: Anbindung der Ereignisse zum Anlegen und Speichern von Daten, Teil 1

```
public class MainWindowViewModel : BindableBase
{
  private readonly IPersonMethods
    personScriptMethods;
  private ICommand newPersonCommand;
  private ICommand savePersonCommand;
  private Person selectedPerson;
  private Person editablePerson;
  private ObservableCollection<Person> persons;

  public MainWindowViewModel(
      IPersonMethods personMethods)
  {
    this.personScriptMethods = personMethods;
  }

  public ICommand NewPersonCommand =>
    newPersonCommand ??=
    new DelegateCommand(OnNewPerson);

  public ICommand SavePersonCommand =>
    savePersonCommand ??=
    new DelegateCommand(OnSavePerson);

  public ObservableCollection<Person> Persons
  {
    get => persons;
    set => SetProperty(ref persons, value);
  }

  public Person SelectedPerson
  {
    get => selectedPerson;
    set => SetProperty(ref selectedPerson, value);
  }

  public Person EditablePerson
  {
    get => editablePerson;
    set => SetProperty(ref editablePerson, value);
  }

  private async void OnNewPerson()
  {
    var person = new Person();

    // Skript ausführen
    await personScriptMethods.OnNewPerson(person);

    SelectedPerson = null;

    // hinzufügen und auswählen
    EditablePerson = person;
  }

  private async void OnSavePerson()
  {
    if (EditablePerson != null)
    {
      if (EditablePerson.Id.Equals(Guid.Empty))
      {
        EditablePerson.Id = Guid.NewGuid();
        Persons.Add(EditablePerson);
        SelectedPerson = EditablePerson;
      }
      // Skript ausführen
      await personScriptMethods.OnPersonSaved(
        EditablePerson);
    }
  }
}

public class ScriptService : BindableBase,
    IDisposable, IScriptDialogs, IPersonMethods,
    IFieldMethods
{
  private readonly IDialogService dialogService;
  private ApplicationScript
    applicationScript = null;
  private V8ScriptEngine scriptEngine = null;
  private ObservableCollection<OutputLine> outputs =
    new ObservableCollection<OutputLine>();
  private readonly object lockOutputs =
    new object();
  private bool disposedValue;
  private List<string> listOfPropertyNames;

  public ScriptService()
  {
    BindingOperations.
      EnableCollectionSynchronization(outputs,
      lockOutputs);
  }

  public ObservableCollection<OutputLine> Outputs
  {
    get => outputs;
    set => SetProperty(ref outputs, value);
  }

  public ApplicationScript Script
  {
    get => applicationScript;
    set
    {
      applicationScript = value;
```

Listing 2: Anbindung der Ereignisse zum Anlegen und Speichern von Daten, Teil 2

```
    InitScript(applicationScript);
  }
}

private void InitScript(ApplicationScript
    scriptSettings)
{
  scriptEngine = new V8ScriptEngine();
  listOfPropertyNames = new List<string>();

  if (!String.IsNullOrEmpty(
      scriptSettings.Script))
  {
    try
    {
      Outputs.Clear();

      scriptEngine.AddHostObject("Console",
        new ScriptObjects.Console(
        OnWriteLine));
      scriptEngine.AddHostObject(
        "Application",
        new ScriptObjects.Application());
    };

    scriptEngine.AddHostObject("Field",
      field);
    scriptEngine.Execute(
      scriptSettings.Script);

    listOfPropertyNames.AddRange(
      scriptEngine.Script.PropertyNames);
    }
    catch(Exception ex)
    {
      OnWriteLine($"Fehler bei der "
        + "Initialisierung {ex.Message}",
        OutputTypes.Error);
    }
  }
}

private void OnWriteLine(string message,
    OutputTypes outputType = OutputTypes.Info)
  {
    var outputLine = new OutputLine(message)
      {OutputType = outputType};
    if (outputs.Any())
    {
      outputs.Insert(0, outputLine);
    }
    else
    {
      outputs.Add(outputLine);
    }
  }

  async Task IPersonMethods.OnNewPerson(
      Person person)
  {
    if (scriptEngine != null &&
      listOfPropertyNames.Contains(
      nameof(IPersonMethods.OnNewPerson)))
    {
      await Task.Run(() =>
      {
        scriptEngine.Script.OnNewPerson(
          person);
      });
    }
  }

  async Task IPersonMethods.OnPersonSaved(
      Person person)
  {
    if (scriptEngine != null &&
      listOfPropertyNames.Contains(nameof(
      IPersonMethods.OnPersonSaved)))
    {
      await Task.Run(() =>
      {
        scriptEngine.Script.OnPersonSaved(
          person);
      });
    }
  }
}
```

Die Implementierung der Automatisierung wurde in einer Komponente namens *ScriptService* zusammengefasst, die alle Aspekte von der Bearbeitung des Skripts bis hin zur Ausführung und Kommunikation von Fehlern und Statusinformationen enthält. Dieser Dienst wird als Singleton in der Anwendung registriert, sodass es nur eine Instanz davon gibt, die global in der Anwendung zur Verfügung steht.

Die Verwendung als Singleton bietet hier einen Geschwindigkeitsvorteil in der Ausführung, da der JavaScript-Code nur beim Start der Anwendung ausgeführt wird und dann, wenn der Nutzer ihn verändert hat. Der Code liegt als Zeichenkette vor, die dann in *ScriptService* ausgeführt und vorgehalten wird. Das bedeutet, dass dort lokale Variablen verwendet werden können, die so lange gehalten werden, bis

Listing 3: Der Weg vom Ereignis zur Ausgabe durch das Skript, Teil 1

```
public class GotFocusToScriptAction<T> : EventAction
    where T: class
{
  private readonly IFieldMethods scriptService;

  public GotFocusToScriptAction(
      IFieldMethods scriptService)
  {
    this.scriptService = scriptService;
  }

  public override async void GotFocus(
      UIElement element)
  {
    object dataContext = null;
    string propertyName = null;

    if (DataHelper.TryToGetBindingProperties(
        element, ref dataContext,
        ref propertyName))
    {
      T typedContext = dataContext as T;
      await scriptService.OnFieldGotFocus(
        typedContext, propertyName);
    }
  }

  public override async void LostFocus(
      UIElement element)
  {
    object dataContext = null;
    string propertyName = null;

    if (DataHelper.TryToGetBindingProperties(
      element, ref dataContext,
      ref propertyName))
    {
      T typedContext = dataContext as T;
      await scriptService.OnFieldLostFocus(
        typedContext, propertyName);
    }
  }
}

public static class DataHelper
{
  public static bool TryToGetBindingProperties(
    UIElement element, ref object dataContext,
    ref string propertyName)
  {
```

der Code neu initialisiert werden muss. Die einzelnen Anwendungszustände, wie beispielsweise Datensatz anlegen und speichern, werden dann als Funktionen mit festgelegten Namen in der ausgeführten Instanz aufgerufen.

Listing 1 zeigt die Registrierung des Dienstes, das Laden des Skripts beim Start der Anwendung sowie – mit dem Aufruf des Dialogs zur Konfiguration des Skripts – ein Beispiel der Verwendung. Die Klasse *ScriptService* implementiert mehrere Schnittstellen, die – gebündelt nach Funktionsbereichen – die jeweiligen Methoden zur Verfügung stellen. Der Container und die Klasse wurden so konfiguriert, dass ausschließlich über die Schnittstellen auf die Funktionalitäten zugegriffen werden kann. Diese Unterscheidung ist notwendig, damit die einzelnen Bestandteile der Anwendung immer nur auf die für sie relevanten Funktionsbereiche zugreifen können.

In dem Beispiel enthalten die Schnittstellen *IServiceDialogs*, *IPersonMethods* und *IFieldMethods* nur sehr wenige Methoden. Das ist dem Umstand geschuldet, dass die Beispielanwendung nur sehr klein ist und einen geringen Funktionsumfang hat. Da Anwendungen in der Regel mehr können, ist es sinnvoll, von Beginn an zwischen den einzelnen Bereichen zu unterscheiden und Schnittstellen zur Differenzierung zu verwenden. Die Klasse *MainWindowViewModel* gibt dann im Konstruktor die Schnittstellen an, deren Instanzen sie benötigt. Dabei handelt es sich dann immer um die gleiche Instanz der Klasse *ScriptService* und somit um die einzige Instanz der Ausführungsumgebung, sodass die einzelnen Interaktionspunkte zwischen Anwendung und Skript Daten untereinander austauschen können.

Die Initialisierung der JavaScript-Ausführungsumgebung erfolgt in dem Moment, in dem ihr ein neues Skript zugeordnet wird. Das passiert zum einen beim Start der Anwendung vor der Anzeige des Hauptfensters und zum anderen dann, wenn der Anwender das Skript bearbeitet und speichert.

Zur Initialisierung gehört die Festlegung, mit welchen Objekten im Skript die Verbindung zur Anwendung hergestellt werden kann. Die Beispielanwendung verwendet dafür die beiden Objekte *Application* und *Console*. Die Ausführung wurde so konfiguriert, dass der Name, unter dem die Objekte im Skript angesprochen werden können, mit den Typenbezeichnungen identisch ist, sodass es sich hier tatsächlich um Instanzen mit dieser Bezeichnung handelt. Das Objekt *Application* kann dazu verwendet werden, mit den Eigenschaften *StatusInfo* und *FieldInfo* Texte in der Statuszeile des Hauptfensters auszugeben. Diese Informationen sind insofern flüchtig, als dass sie nach einigen Sekunden wieder verschwinden oder durch eine nachfolgende Aktion überschrieben werden.

Das Objekt *Console* hingegen ist dazu gedacht, im Skript Ausgaben zu ermöglichen, die erhalten bleiben und die der Anwender für die gesamte Laufzeit der Anwendung einsehen kann. Im .NET-Kontext ist das am besten mit Debug-Ausgaben vergleichbar. Die Methoden *WriteInfo()*, *WriteWar-* ▶

Listing 3: Der Weg vom Ereignis zur Ausgabe durch das Skript , Teil 2

```
if (element is TextBox textBox)
{
  var textBinding = BindingOperations.
    GetBinding(textBox,
    TextBox.TextProperty);

  if (textBinding != null)
  {
    object dataObject = textBox.DataContext;
    string bindingPath =
      textBinding.Path.Path;

    if (dataObject == null ||
      string.IsNullOrEmpty(bindingPath))
    return false;

    var pathSegments = bindingPath.Split(".");

    for (int index = 0; index < pathSegments.
        Length; index++)
    {
      var segmentName = pathSegments[index];

      if (index == pathSegments.Length -
        1) // das letzte benötigen wir
        {
          dataContext = dataObject;
          propertyName = segmentName;
          return true;
        }
        else
        {
          var property = dataObject.GetType().
            GetProperty(segmentName);

          if (property == null)
            break;

          dataObject = property.GetValue(
            dataObject);

          if (dataObject == null)
            break;
        }
      }
    }
  }
  return false;
 }
}
```

ning() und *WriteError*() erlauben eine differenzierte Ausgabe der gewünschten Angaben mit einer farblichen Markierung entsprechend der Art der Ausgabe. Aus Gründen der Gewöhnung hat das Objekt *Console* zusätzlich eine Methode *log()*, die sich so verhält, wie man es aus der klassischen Verwendung von JavaScript im Browser erwartet. Diese ist technisch zwar nicht notwendig, für JavaScript-erfahrene Anwender jedoch sehr hilfreich.

Anlegen und speichern

Der erste Anwendungsfall demonstriert das Zusammenspiel zwischen den Ereignissen „Daten erstellen“ und „Daten speichern“. Anwender, die oft mit diesen beiden Funktionalitäten zu tun haben, wünschen sich häufig eine starke Vereinfachung ihrer Eingabemöglichkeiten, wie sie beispielsweise die Vorbelegung einzelner Felder sein kann. Das hier gezeigte Beispiel belegt die Daten mit den Werten vor, die zuletzt gespeichert wurden. Notwendig sind dazu lediglich zwei Ereignisse. Listing 2 zeigt die wesentlichen Ausschnitte aus den beteiligten Komponenten.

Die Anbindung der beiden Ereignisse erfolgt nach dem üblichen MVVM-Muster über Commands im ViewModel. In deren Ausführungsmethoden wird dann der Dienst aufgerufen. Die dabei verwendeten Methoden entsprechen der Einfachheit halber den Namen der Methoden im Skript. Die verwendeten Parameter vom Typ *Person* lassen sich problemlos übergeben und im Skript verwenden, sodass das folgende JavaScript möglich ist:

```
var lastSavedCity = "";

function OnPersonSaved(person) {
  lastSavedCity = person.City
}

function OnNewPerson(person) {
  if (lastSavedCity != "") {
    person.City = lastSavedCity
    Console.WriteInfo("Neue Person wurde initialisiert")
  }
}
```

Beim Speichern eines Datensatzes wird der Wert im Feld *City* festgehalten, mit dem beim Erstellen eines neuen Datensatzes der Wert wieder initialisiert wird. Weitere Funktionen könnten hier mehrere Felder berücksichtigen oder auch das Nachschlagen einer Postleitzahl ermöglichen.

Verbindungen aufbauen

Der zweite Anwendungsfall soll die Frage in den Mittelpunkt stellen, ob im Skript Oberflächenelemente der WPF-Anwendung verwendet werden sollen. Auf der einen Seite ist das

naheliegend, denn Steuerelemente stellen die finale Form der Ausgabe von Daten dar und sind damit für Anwender ein gut zu verstehendes Element der Oberfläche. Es gibt jedoch zwei Aspekte, die klar dagegen sprechen. Der erste Aspekt ist, dass es mit MVVM als Basisarchitektur der Anwendung keine direkte und damit einfach zu verwendende Verbindung zwischen Daten und Oberfläche gibt. Die Beziehung wird ausschließlich über ein Binding hergestellt, das gegebenenfalls um Konverter ergänzt wird. Es ist Sinn und Zweck einer MVVM-Architektur, dass die Daten nichts von ihrer Darstellung auf der Oberfläche wissen. Daher lässt sich diese Verbindung, obwohl sie ja existiert, nicht ohne Weiteres rekonstruieren und für andere Anforderungen nutzen.

Der zweite Aspekt ist, dass der Kontext des Skripts eher datenbasiert ist. Im ersten Anwendungsfall dieses Artikels bestand die Möglichkeit, Daten während ihrer Initialisierung zu beeinflussen. Dieser Ansatz ist für den Anwender leichter zu verstehen als die alternative Manipulation von Textfeldern. Legt man Integrität und Konsistenz als die wichtigste Prämisse einer Schnittstelle zwischen Anwender und Skript zugrunde, ist es notwendig, auch bei anderen Szenarien bei diesem Vorgehen zu bleiben.

Funktional geht es bei diesem Beispiel um die Vermeidung eines Fehlers, der Anwendern häufig bei der Eingabe komplexer Daten passiert: Werden neue Daten in einem Textfeld eingegeben, gehen damit gleichzeitig die vorherigen Daten verloren. Mitunter ist es jedoch wünschenswert, dass diese vorherigen Daten zumindest solange dargestellt werden, bis der Anwender durch Verlassen des Feldes die Eingabe abgeschlossen hat. Gerade bei Daten, bei denen sich aufgrund ihrer Form schnell Fehler einschleichen, beispielsweise einer ISBN, kann es durchaus sinnvoll sein, eine optische Unterstützung anzubieten.

Listing 3 zeigt die wesentlichen Bestandteile einer solchen Möglichkeit. Wird ein Feld aktiv, so wird der aktuelle Wert in der Statuszeile angezeigt und der Anwender kann neue Daten eingeben, ohne die alten aus den Augen zu verlieren. Die Anzeige der alten Daten verschwindet erst, wenn der Anwender das Feld verlässt. Das Skript verbindet das Ereignis mit der Ausgabe in einem Bereich der Statuszeile:

```
function OnFieldGotFocus(person, propertyName) {
  Application.FieldInfo = person[propertyName]
}

function OnFieldLostFocus(person, propertyName) {
  Application.FieldInfo = ""
}
```

Theoretisch wären hier auch noch andere Funktionalitäten denkbar. Beispielsweise ließen sich eine bedingte Undo-Funktion oder eine individuelle Eingabehistorie implementieren, die anzeigt, welche Werte bislang über alle Datensätze hinweg im betreffenden Feld eingetragen worden sind.

Aus technischer Sicht stellt dieses Szenario eine besondere Herausforderung dar, denn hier muss die Perspektive ausgehend von Steuerelementen und Ereignissen in eine funktionale übersetzt werden, die sich eher durch Daten darstellt. Der Weg vom Ereignis *GotFocus* hin zum Aufruf der entsprechenden Methode im Skript führt über mehrere Schritte. Dabei werden die Informationen aus dem Steuerelement zu Informationen die Instanz der Daten betreffend überführt.

Der erste Schritt besteht in einer Attached Property vom Typ *EventAction*, welche das Ereignis abfängt und an die Methode *GotFocus()* weiterleitet. Die Instanz dieser Eigenschaft kommt dann vom ViewModel. Hier wurde die Klasse *GotFocusToScriptAction* als Ableitung von *EventAction* erstellt, um ein spezifisches Verhalten umzusetzen. Innerhalb dieser Methode wird dann ausgehend vom Oberflächenelement dessen *DataContext* und anhand der Bindung der Feldname ermittelt.

Damit sind alle notwendigen Angaben vorhanden, da der Datenkontext die jeweilige Instanz der Entität und der Feldname dessen Eigenschaft darstellt. Der Aufruf lässt sich somit an das Skript weiterleiten und es ist möglich, individuell zu reagieren.

Fazit

Die Verwendung einer Skriptsprache zum Automatisieren der Anwendung stellt sowohl konzeptionell als auch organisatorisch eine Herausforderung dar. Konzeptionell besteht sie darin, eine Technik umzusetzen, die sich nicht nach Technik anfühlt und die sich anhand der angebotenen Komponenten und Verfahren gut in das Verständnis der anvisierten Zielgruppe einfügt oder sich ihr gut vermitteln lässt. Die Kombination von ClearScript als Ausführungsumgebung einer JavaScript-Engine mit WPF und Prism stellt hier eine gute Voraussetzung dar, da deren Bestandteile lose gebunden, leicht ausbaubar und damit innerhalb der Umsetzung gut zu verwenden sind.

Weitaus schwieriger zu lösen sind die organisatorischen Herausforderungen, denn die Möglichkeit zur Automation einer Anwendung lässt sich nicht ohne ein fundiertes Verständnis des Themas und der Vorgehensweise bei allen Beteiligten umsetzen. Dieses Verständnis gilt es schrittweise aufzubauen, indem Beispiele Sinn und Zweck zeigen und Alternativen in ihren Vor- und Nachteilen verdeutlicht werden.

Fällt am Ende eine Entscheidung für Automation, bedeutet das für die Entwickler ein spannendes Thema und für die Nutzer der Anwendung reichhaltige Möglichkeiten, ihre ganz persönlichen Anforderungen abgedeckt zu sehen. ■

[1] ClearScript 7, www.dotnetpro.de/SL2108Clearscript1

[2] Beispielanwendung ClearScriptAppStudy, www.dotnetpro.de/SL2108Clearscript2

Torsten Zimmermann
entwickelte bereits mit Visual Basic 1.0. Seitdem er sich mehr konzeptionellen Fragen zugewandt hat, liegt sein Schwerpunkt im Bereich des Entwurfs großer und verteilter Anwendungen sowie der Anwendungsautomation.
zimmy@web.de

dnpCode A2108Clearscript