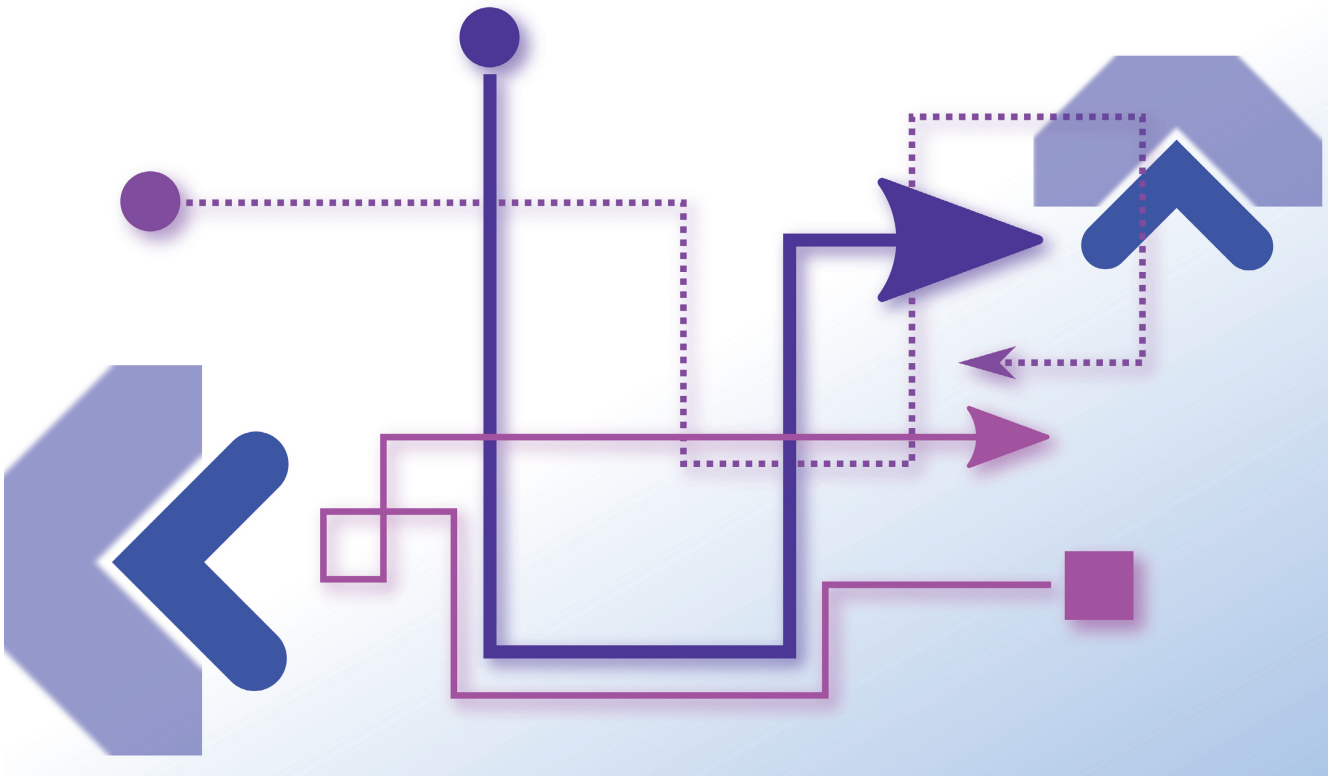


NAVIGIEREN MIT SMART.NAVIGATION .NET

Der Weg durch die Anwendung

Die Bibliothek hilft bei der Navigation innerhalb einer Anwendung.

Software ist ein komplexes Gebilde. Nicht nur bezogen auf Code, Architektur oder eingesetzte Technologie, sondern meist auch aus Sicht eines Benutzers. Die Funktionen einer Software sind in der Regel auf verschiedene Sichten aufgeteilt, und die Navigation zwischen diesen Ansichten kann mitunter in ein komplexes Logik-Gebilde münden.

In Webanwendungen, die mit Angular oder Vue.js umgesetzt sind, gibt es die Router-Funktionalität, die genau hier unter die Arme greift. Sie reduziert die Komplexität einer umfassenden Navigation erheblich.

Aber wie sieht das auf dem klassischen Desktop aus? Gibt es eine ähnliche Funktionalität bei Windows Forms und Windows Presentation Foundation (WPF)?

Was ist Smart.Navigation .NET?

Die Antwort darauf ist ein klares „Jein“. Während unter Windows Forms eine solche vorgefertigte Lösung fehlt, bietet WPF die Klasse *NavigationService* [1] an, mit der eine Navigation realisierbar ist. Verschiedene Lösungen über die diversen Plattformen und Frameworks hinweg machen das Entwicklerleben allerdings nicht einfacher. Über die Jahre sind daher Frameworks und Bibliotheken entstanden, mit denen eine Navigation realisiert werden kann.

Smart.Navigation .NET, im Folgenden Smart.Navigation genannt, ist eine dieser Bibliotheken. Ihr erklärtes Ziel ist es, die Navigation innerhalb einer Anwendung zu vereinfachen und darüber hinaus über verschiedene Plattformen hinweg zu vereinheitlichen.

Dass das Thema Navigation nicht unbedeutend ist, wird bei einer Suche in der NuGet-Galerie deutlich. Über 500 Pakete befassen sich mit der Navigation, darunter auch Smart.Navigation, dessen NuGet-Paket *Usa.Smart.Navigation* heißt. Die Bibliothek bringt Support für Windows Forms, WPF und Xamarin mit. Das Prinzip hinter Smart.Navigation lässt sich am einfachsten mit dem Begriff des Control Switching beschreiben. Es werden gar keine besonders ausgeklügelten Mechanismen oder schmutzigen Tricks genutzt, um einer Anwendung eine konkrete Navigation beizubringen.

Control Switching bedeutet, dass einer Basiskomponente eine beliebige Anzahl von Navigationskomponenten hinzugefügt wird, die dann bei Navigationsereignissen ausgetauscht, sprich entfernt und wieder hinzugefügt werden.

Der Zugriff auf die einzelnen Navigationskomponenten erfolgt über eine View-ID, ein weiteres Merkmal von Smart.Navigation.

Installation

Die Installation läuft ohne Probleme ab. Verpflichtend ist die Kern-Bibliothek mit Namen *Usa.Smart.Navigation*. Je nach Plattform ist dann eines der weiteren Pakete erforderlich. In unserem Beispiel, das unter Windows Forms spielt, ist das *Usa.Smart.Navigation.Windows.Forms*.

Die Namensgebung ist etwas verwirrend, da es noch die Pakete *Usa.Smart.Navigation.Forms* für Xamarin und *Usa.Smart.Navigation.Windows* für WPF gibt.

Zudem machen die vielen fehlenden Buchstaben und Rechtschreibfehler in den Paketbeschreibungen nicht den besten Eindruck. Allerdings muss das keinen Einfluss auf die Qualität und Funktionalität haben.

Das Kernpaket bringt Unterstützung für das .NET Framework 4.6.2 und .NET Standard 2.0 mit. Die Windows-Forms-spezifische Komponente bietet Unterstützung für das erstgenannte Framework. Weitere Abhängigkeiten sind nicht vorhanden, und auch ansonsten gibt es bei der Installation nichts weiter zu beachten.

Die Bibliothek ist Open Source und der Code liegt auf GitHub [2]. Eine Lizenz ist leider nicht angegeben.

Die Features im Überblick

Die Bibliothek Smart.Navigation bringt zahlreiche Funktionen mit. Die Navigation durch Control Switching, die Identifikation der Navigationskomponenten anhand einer View-ID und die Unterstützung mehrerer Plattformen wurden bereits erwähnt. Hinzu kommen noch folgende Funktionen:

- Unterstützung für Parameterübergabe zwischen Navigationszielen,
- verschiedene Arten der Navigation (Stacked, Wizard, …),
- Events zum Lebenszyklus,
- Cancel-Event,
- Integration von Plug-ins und anderen Bibliotheken.

Dieser Artikel fokussiert sich auf eine Auswahl. Wie bereits erwähnt, ist das Beispiel mit Windows Forms realisiert und zeigt zwei Arten zu navigieren.

Das Drumherum im Code

Es gibt nur selten den Fall, dass einfach Methoden aufgerufen werden können, die dann die Arbeit erledigen. Das mag in funktionalen Sprachen gehen, in der objektorientierten Welt ist das des Öfteren ein größerer Akt und eine Architekturfrage. Bei Smart.Navigation ist das nicht anders. Es bietet sich an, den Komponenten, die Teil einer Navigation sind, eine gemeinsame Basisklasse zu verpassen. Hier lassen sich bestimmte Anforderungen der Bibliothek bündeln, um auf diese Weise etwas Code und Aufwand zu sparen.

Listing 1: Die Konfiguration des Navigators

```
navigator = new NavigatorConfig()
  .UseControlNavigationProvider(navigationPanel)
  .UseIdViewMapper(m => m.AutoRegister(
    Assembly.GetExecutingAssembly().ExportedTypes))
  .ToNavigator();
navigator.Exited += OnExited;
navigator.Navigating += OnNavigating;
```

Bei Windows Forms liegt ein User Control als Basisklasse nahe. Dadurch kann direkt etwas Windows-Forms-Funktionalität gebündelt werden. Zusätzlich sind einige Interfaces von Smart.Navigation notwendig, um die Navigation zu ermöglichen. Diese Schnittstellen sind folgende:

- *INavigatorAware*
- *IConfirmRequest*
- *INavigationEventSupport*

Zusätzlich ist es zweckmäßig, durch eine weitere Schnittstelle, zum Beispiel *IApplicationView*, Eigenschaften anzubieten, um durch jede Navigationskomponente einen Titel setzen zu können, ebenso wie die Einstellung und Methode, um zur Ausgangs-View (Home) zu navigieren. Letztgenannte Schnittstelle bietet Methoden an, die bei diversen Events aufgerufen werden, zum Beispiel bei einer Navigation von oder zu einer Komponente. Auf diese Weise lassen sich global Events bündeln beziehungsweise in den einzelnen Navigationskomponenten bei Bedarf überschreiben.

Die sogenannte Main-View wird durch das User Control allerdings nicht erweitert. Die Main-View ist die Komponente, die als Basis des Control Switching dient. Sie definiert den Ort und die Größe der anderen Komponenten und wird bei der Navigation ausgetauscht. Bei Windows Forms reicht dazu ein normales Panel aus.

Bevor es mit der Navigation losgehen kann, muss diese konfiguriert werden. Dafür ist die Klasse *NavigatorConfig* zuständig. Getreu dem Motto „Talk is cheap, show me the code" zeigt Listing 1 die angesprochene Konfiguration. Zuerst wird das Panel zugewiesen, das als Basis für die jeweiligen Navigations-Sichten dient. Der View-Mapper, auf Basis von IDs, kann eine angegebene Menge von Typen automatisch auf die Eigenschaft *[View()]* testen.

Als Parameter nimmt diese Property die *ViewId*, zum Beispiel *ViewId.FirstWizard*, entgegen. Auf diese Weise kann die Navigation über eine einzelne ID erfolgen. Der konfigurierte Navigator weiß, welches Element gemeint ist. Wenn die Registrierung manuell erfolgen muss, steht zusätzlich noch die Methode *Register* zur Verfügung, die eine ID und einen Typ entgegennimmt. Darüber hinaus steht zudem die Methode *UseDirectViewMapper* bereit. Dabei wird kein direktes Mapping vorgenommen. Die Navigation erfolgt nicht über eine ID, sondern über die Angabe des Typs der View, zu der navigiert werden soll: ▶

```
navigator.Forward(typeof(View1));
```

Wer lieber eine Navigation im Sinne eines Routings nutzen möchte, der nutzt die Konfigurationsoption *UsePathViewMapper*.

Listing 2 zeigt die dafür notwendige Konfiguration. Zunächst wird eine Wurzel angegeben, die zur Spezifikation des Namensraums dient, in dem sich die Views befinden. Das Suffix gibt an, mit welcher Zeichenkette der Name der Views enden soll. Auf diese Weise ist, basierend auf der Namensgebung von Views und deren Namensräumen, ein Routing möglich, wie es Listing 2 ebenfalls zeigt. Eine Kind-View könnte als Beispiel wie folgt definiert sein:

```
namespace Example.Views.Children
{
  public class Child1View
  {
  }
}
```

Nach der Konfiguration der Navigation kann es losgehen. Die folgenden drei Zeilen sorgen dafür, dass das Hauptfenster angezeigt wird, zu einer View navigiert wird und die dann gültige View den Fokus bekommt:

```
Show();
navigator.Forward(ViewId.Menu);
((Control)navigator.CurrentView).Focus();
```

Die anderen Zeilen in der gezeigten Navigation (Listing 1) dienen dem Abonnieren von Events, dem Ändern des Textes im Titel der Main-Form und um auf das Schließen der Navigation reagieren zu können.

Listing 2: Konfiguration für ein Path-Routing

```
var navigator = new NavigatorConfig()
  .UseMockFormProvider()
  .UsePathViewMapper(option =>
  {
    option.Root = "Example.Views";
    option.Suffix = "View";
    option.AddAssembly(
      Assembly.GetExecutingAssembly());
  })
  .ToNavigator();

// navigation
navigator.Forward("/Parent");
navigator.Forward("/Children/Child1");
navigator.Forward("Child2");
navigator.Forward("../Parent");
navigator.Forward("Children/Child2");
```

Eine Stack-Navigation

Die Navigation startet, in diesem Fall, mit der ID *ViewId.Menu*. Dieses Menü zeigt zwei einfache Buttons an, um entweder die Stack- oder die Wizard-Navigation zu starten (siehe Bild 1). Um zum Beispiel nach vorne zu navigieren, reicht ein Aufruf wie folgt aus:

```
Navigator.Forward(ViewId.FirstStack);
```

Zusätzlich zu Forward gibt es noch Methoden für *Pop* und *Push* sowie asynchrone Varianten.

Für die Stack-Navigation sind primär die Methoden *Pop* und *Push* zuständig. Sobald mit *Forward* die erste View der Stack-Navigation angezeigt wird, kann diese View zu anderen Views im Stack-Prinzip navigieren. Nach vorne geht es mit einem *Push*, zurück mit einem *Pop*, wieder zum Hauptmenü mit einem Aufruf von *PopAllAndForward*. Wenn mehrere Views auf einmal übersprungen werden sollen, hilft die *Pop*-Methode mit einem Parameter weiter, der den Level der *Pop*-Operation bestimmt.

Das Ergebnis ist eine Navigation wie auf einer Leiter. Von einem bestimmten Level aus geht es nach oben oder unten – mit dem Unterschied, dass in der *Pop*-Navigation auch größere Sprünge möglich sind, ohne dass Lebensgefahr besteht.

Eine Wizard-Navigation

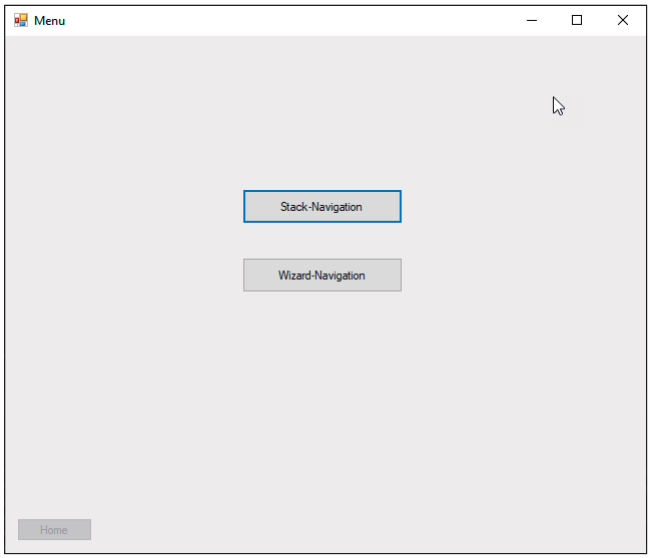
Eine sogenannte Wizard-Navigation ist eine sehr beliebte Art, in einer Anwendung zu navigieren. Der Wizard führt in der Regel durch notwendige Schritte bei einem Arbeitsablauf. In einem solchen Hilfsdialog kann horizontal navigiert werden. Das heißt, eine View wird angezeigt, nach rechts geht es weiter und, wenn im Kontext sinnvoll beziehungsweise erlaubt, nach links wieder zurück.

Im Fall von Smart.Navigation bietet die Methode *Forward* alles an, um eine Wizard-Navigation zu erstellen. Die Navigationselemente werden nicht technisch oder organisatorisch in einer bestimmten Reihenfolge angeordnet. Durch den Aufruf einer bestimmten View und die UI-Elemente darauf wird dem Benutzer suggeriert, dass er in der Navigation entweder nach vorne oder wieder zurück geht. Technisch gesehen wird lediglich die aktuell angezeigte View gegen eine neue ausgetauscht.

Wenn auch von der Implementierung her eine Art Vor-und-Zurück gewünscht ist, kann die eben vorgestellte Stack-Navigation helfen.

Notwendig ist das nur, wenn eine technische Absicherung notwendig ist, dass nach einer View nur eine bestimmte vorhergehende angezeigt werden soll. Die Stack-Navigation kann das über den Stapel sicherstellen.

Das Beispiel in der Test-Solution zeigt zudem, wie ein gemeinsamer Kontext genutzt werden kann, um Daten zwischen verschiedenen Views zu übertragen, beziehungsweise wie ein gemeinsamer Datenspeicher realisierbar ist. Die Klasse *WizardContext* speichert zwei Datenwerte in Form von Zeichenketten. Die implementierte Schnittstelle *IInitializable* bietet zudem die Möglich an, den Datenspeicher zu initialisieren. Innerhalb der Wizard-Views reicht die folgende



Das Menü der Navigation, realisiert als Navigation-View (Bild 1)

Eigenschaft inklusive Annotation für den gemeinsamen Datenspeicher aus:

```
[Scope]
public WizardContext Context { get; set; }
```

Die Property *[Scope]* sorgt dafür, dass die Eigenschaft mit dem entsprechenden Objekt initialisiert wird. Sobald eine View mit einem solchen Scope das erste Mal angezeigt wird, initialisiert Smart.Navigation das Objekt und setzt die Eigenschaft entsprechend. Sobald eine View angezeigt wird, die diesen Scope nicht enthält, wird das dahinterliegende Objekt abgeräumt. Der Begriff Scope der Annotation kann somit wörtlich genommen werden, da die Lebenszeit einem bestimmten Scope von Views entspricht. Auf diese Weise ist es recht simpel, Daten zwischen Views auszutauschen.

Ein Ausblick

Ein Feature von Smart.Navigation, das lediglich am Rande erwähnt werden soll, sind Plug-ins. Die Möglichkeit der Scopes, die als Plug-in realisiert sind, wurde bereits gezeigt. Darüber hinaus gibt es noch das Parameter-Plugin.

Über das Attribut *[Parameter]* können Eigenschaften ausgezeichnet werden, deren Datenwerte über Views hinweg transportiert werden.

Des Weiteren ist die Einbindung von weiteren Bibliotheken möglich, zum Beispiel über die Schnittstellen *IActivator* und *IConverter*. Erstere dient dazu, Objekte zu initialisieren, Letztere dazu, Umwandlungen durchzuführen. Eine Implementierung für *IActivator* wird über die Bibliothek *Usa.Smart.Resolver* [3] angeboten, die vom selben Autor stammt.

Komplex oder doch nicht?

Bei Bibliotheken dieser Art stellt sich immer wieder eine Kernfrage: Ist es einfacher, die Funktionalität selbst zu implementieren, oder lohnt sich die Einarbeitung, um anschließend die Features geschenkt zu bekommen? Diese Frage muss individuell pro Bibliothek beantwortet werden, da sie zu einem großen Teil davon abhängt, wie komplex die Bibliothek ist, und vor allem wie komplex sich die Einbindung gestaltet.

Bei Smart.Navigation ist nur eine geringe Komplexität vorhanden: Bis auf die Konfiguration der Navigation ist nichts Spezielles zu implementieren.

Die einzelnen Views als UserControls zu implementieren ist auch ohne die Bibliothek eine gängige Praxis. Jetzt kommt noch eine *ViewId* hinzu, wenn die Navigation mittels ID genutzt werden soll. Insgesamt hält sich aber der Aufwand stark in Grenzen.

Von Vorteil sind die vielen Plug-ins und sonstigen Schnittstellen. Charakteristisch bei Smart.Navigation ist, dass praktisch jede Funktionalität hinter einem Interface gekapselt ist und eine Möglichkeit existiert, eigene Implementierungen anzugeben. Das macht es nicht viel komplexer, allerdings deutlich anpassungsfähiger und flexibler.

Alles in allem ist Smart.Navigation damit nicht komplexer als eine eigene Implementierung. Wer allerdings nur die Navigation nutzen möchte, ohne weitere Funktionalität der Plug-ins, der hat auf der anderen Seite auch nahezu keine Vorteile durch den Einsatz der Bibliothek.

Fazit

Smart.Navigation .NET ist eine kleine, aber feine Bibliothek zur Bereitstellung von Funktionen zur Navigation in Anwendungen. Die vielen verschiedenen Optionen, eine Navigation zu konfigurieren und zu implementieren, sowie die vielen Plug-ins und Erweiterungsmöglichkeiten machen Smart.Navigation sehr flexibel.

Ob sich der Einsatz tatsächlich lohnt, kann nur projektweise entschieden werden. Hier gilt es, zwischen einer weiteren Abhängigkeit mit etwas Komplexität und einer eigenen Implementierung abzuwägen.

Insgesamt ist Smart.Navigation .NET eine ordentliche Bibliothek. Die angebotenen Features funktionieren und die Doku ist schlicht, aber zu großen Teilen inhaltlich ausreichend. Daher verdient sich Smart.Navigation .NET ein „Gut“ und die Empfehlung, sich die Bibliothek einmal anzuschauen. ■

[1] Microsoft-Dokumentation zum NavigationService, www.dotnetpro.de/SL1903Frameworks1
[2] Smart.Navigation .NET auf GitHub, www.dotnetpro.de/SL1903Frameworks2
[3] Die Bibliothek Usa.Smart.Resolver in der NuGet-Galerie, www.dotnetpro.de/SL1903Frameworks3

Fabian Deitelhoff
promoviert am Graduiertenkolleg „User-Centred Social Media“ im Themenumfeld der Code Comprehension. Zudem ist er Autor, Entwickler, Trainer und Gründer von www.brickobotik.de. Erreichbar über www.fabiandeitelhoff.de oder **@FDeitelhoff**.

dnpCode A1903Frameworks