Warum sich Unit-Tests und TDD auch für WPF lohnen

Schrumpfkur für die Bug-Quote

Auch in der WPF lassen sich rund 80 Prozent des Codes mit automatisierten Unit-Tests validieren. Wie und warum Test-Driven Development (TDD) dafür eine probate Methode ist, zeigt dieser Artikel.

Auf einen Blick

Dr. Andreas Arnold ist Softwareentwickler für Java und .NET bei der andrena objects ag in Karlsruhe. Erfahrungen hat er besonders mit der Migration alter Anwendungen auf neue Technologien. Er interessiert sich für die automatisierte Messung von Softwarequalität.

Inhalt

dnpCode
A1408TDDWPF

Automatisierte Tests gelten auch bei der Windows Presentation Foundation (WPF) als Mittel der Wahl, um langfristig eine geringe Bug-Quote zu sichern. Allerdings handelt es sich bei diesen Tests oft um GUI- oder Integrationstests. Bei näherem Hinsehen gilt nur für eine winzige Minderheit der Projekte, dass der Großteil des Codes durch Unit-Tests abgedeckt ist, ein kleinerer Teil durch Integrationstests und ein verschwindender Teil durch manuelle Tests, wie es die Testpyramide fordert. Die Gründe dafür sind vielfältig und sollen hier bis auf eine Ausnahme nicht weiter betrachtet werden. Diese Ausnahme bildet die häufig zu hörende Überzeugung, die gesamte Präsentationsschicht könne sowieso nicht mit Unit-Tests getestet werden. Hier müssten GUI-Tests her.

Tatsächlich erscheinen GUI-Tests verlockend, denn oft ist es aufgrund einer ungeschickt gewählten Architektur wirklich schwer, einen Unit-Test zu schreiben. Außerdem gibt es viele Tools, die das Schreiben von GUI-Tests erleichtern. Wenn es also leichter ist, einen GUI-Test zu schreiben als einen Unit-Test, warum sollte man das dann nicht einfach tun? Weil das Problem nicht im Schreiben besteht. Es taucht dann auf, wenn der Test einmal rot läuft, oder auch dann, wenn die Anzahl der Tests gewachsen ist. Unit-Tests können auch in sehr großer Anzahl schnell ablaufen, selbst 10000 Tests sind heutzutage kein Problem. Wenn ein Unit-Test rot ist, dann kann man aufgrund der kleinen getesteten Einheit schnell darauf schließen, wo der Fehler liegt. Bei GUI-Tests ist das anders: Erstens brauchen sie sehr lange, und zweitens sind sie teilweise auch dann rot, wenn beispielsweise der Bildschirmschoner läuft. In vielen Fällen sind GUI-Tests zwar sehr hilfreich, um Bugs in der Benutzeroberfläche zu vermeiden, aber für ein ausgiebiges Testen der Abläufe innerhalb einer Anwendung sind sie nicht geeignet. Hier ist eine hohe Testabdeckung durch Unit-Tests unabdingbar.

TDD ist eine sehr einfache Möglichkeit, diese hohe Testabdeckung zu garantieren. Natürlich ist es nicht immer sinnvoll, für jede Zeile Code einen Unit-Test zu haben, aber zumindest sollte man in der Lage sein, jede Zeile Code zu testen. Denn dann kann man bereits aufgetretene Bugs zukünftig mit einem Unit-Test verhindern. Das Anwenden von TDD zwingt dazu, testbaren Code zu schreiben. Das hat obendrein den Vorteil, dass sofort zu sehen ist, ob der geschriebene Test auch wirklich den Code testet, den man gerade programmiert. Ansonsten passiert es schnell, dass man zwar einen grünen Test programmiert, der aber den zu testenden Code gar nicht durchläuft.

Architektur für testbare GUI-Schichten

Die Idee von einem testbaren Design wurde durch das MVVM-Pattern schon gut umgesetzt. MVVM ist das Akronym für Model, View und ViewModel. Alle drei unterscheiden sich deutlich im Hinblick auf ihre Testbarkeit mit Unit-Tests. Im Model wird die Businesslogik umgesetzt, daher ist dieser Teil der Anwendung am leichtesten mit Unit-Tests zu testen. Anders sieht die Sache bei der View aus. Sie beschreibt das Aussehen der Anwendung, in der View sollte nach Möglichkeit keine Logik stehen, weil hier wenig bis gar nicht mit Unit-Tests getestet werden kann. An dieser Stelle sind GUI-Tests eine sinnvolle Option. Die ganze Entscheidungslogik der GUI sollte deshalb in das ViewModel, das sich auch sehr gut mit Unit-Tests prüfen lässt.

Generell können die wichtigsten Punkte, die in einer WPF-Anwendung auftreten, mit Unit-Tests einfach getestet werden. Da diese Punkte im Allgemeinen weit über 80 Prozent des Codes betreffen werden, dürfte es meist ausreichen, wenn man sie richtig umsetzt.

Aktualisierung der View vom ViewModel aus

Bei MVVM arbeitet man in der View mit Bindings. Die View bindet sich an eine Property. Sobald in der View die Property verändert wird, wird der Wert automatisch in das ViewModel geschrieben. Um vom ViewModel aus nun eine Änderung an die View zu melden, gibt es das Event *PropertyChanged* aus dem *INotifyPropertyChanged*-Interface.

```
<TextBlock Text="{Binding Name}" />
public string Name {
  get { return _name; }
  set { _name = value;
  RaisePropertyChanged(); }
}
```

Listing 1

PropertyChangedHandler.

```
public class PropertyChangedHandler {
  private readonly IList<string>
    _raisedProperties =
      new List<string>();
  public void PropertyChanged(
    object sender,
    PropertyChangedEventArgs args)
  {
    _raisedProperties.Add(
      args.PropertyName);
  }
  public void Verify(string property) {
   Assert.IsTrue(
    _raisedProperties.Contains(property),
    "PropertyChanged Event for <" +
       property + "> missing");
  }
}
```

Listing 2

Testmethode für PropertyChanged.

```
[TestMethod]
public void
  SettingNameRaisesPropertyChanged()
{
  var model = new ViewModel();
  model.PropertyChanged +=
    _propertyChangedHandler.
    PropertyChanged;
  model.Name = "Das bin ich";
  _propertyChangedHandler.Verify("Name");
}
```

In diesem einfachen Fall ist es vielleicht nicht notwendig, sich mittels eines Unit-Tests zu vergewissern, dass das *PropertyChanged*-Event auch wirklich geworfen wurde. Bei komplexeren Fällen dagegen kann es durchaus interessant werden. Und da es recht aufwendig ist, jedes Mal an das Event zu binden, kann man hier eine einfache Hilfsklasse bauen, die das Testen erleichtert (Listing 1).

Den Unit-Test für die beschriebene Property zeigt Listing 2.

Der *PropertyChangedHandler* macht es also einfach, dieses Event zu testen, der Testcode bleibt schlank und gut zu warten.

Mittels Properties kann man nun auch die Sichtbarkeit oder die Farben von Elementen beeinflussen. Entweder nutzt man *bool*-Properties oder *Enums*, um in der View dann über Trigger die passenden Farben zu wählen, oder man gibt im ViewModel direkt eine *Visibility*- oder eine *Brush*-Eigenschaft zurück. In beiden Fällen ist es einfach, einen Test zu schreiben.

Interaktion mit dem Benutzer: Commands

Um die Interaktion mit dem Benutzer zu steuern, benötigt man Commands. Beim Klick auf einen Button beispielsweise wird ein Command ausgelöst. In MVVM ist es nun üblich, diesen als *RelayCommand* zu implementieren:

```
<Button Command="{Binding SaveCommand}" />
public ICommand SaveCommand {
  get {
    return _saveCommand ?? (_saveCommand =
      new RelayCommand(SaveMethod));
  }
}
```

Um diesen Command nun testen zu können, kann man ihn – ohne zusätzlichen Aufwand – direkt ausführen:

```
[TestMethod]
public void CommandIsExecuted()
{
  var model = new ViewModel();
  model.SaveCommand.Execute(null);

  // verify result
}
```

Leider muss man bei parameterlosen Commands trotzdem den Parameter *null* übergeben. Commands mit Parameter kann man über *RelayCommand<TParameter>* abbilden.

Commands und Properties bieten nun schon eine Vielzahl von Möglichkeiten. Allerdings gibt es noch viele Ereignisse in der WPF, die nicht als Command, sondern nur als Event angeboten werden.

Testen von Events

Leider stellen sich viele dieser Events als nur schwer testbar heraus. Oft implementiert man die Events direkt als private Methoden im Codebehind zu XAML. In einem Unit-Test müsste man daher erst einmal die View erstellen, was der Forderung nach einem schnellen und kurzen Test komplett zuwiderläuft. Gesucht ist also eine andere Lösung, wie sie Microsoft Interactivity anbietet: Man bindet einfach alle Events an einen Command.

Auf diese Weise ist es nun auch möglich, Events im ViewModel zu implementieren und zu testen.

Listing 3

Events an einen Command binden.

```
<ListBox x:Name="Liste" ...>
  <i:Interaction.Triggers>
    <i:EventTrigger
       EventName="SelectionChanged">
      <i:InvokeCommandAction
         Command="{Binding
         SelectItemCommand}"
         CommandParameter="
          {Binding SelectedItem,
           ElementName=Liste}"/>
    </i:EventTrigger>
  </i:Interaction.Triggers>
</ListBox>
```

Komplexere Elemente

Mit den genannten Methoden kann man nun schon einen Großteil der Anwendung testen. Dennoch gibt es immer wieder Fälle, in denen diese Methoden nicht ans Ziel führen. Baut man zum Beispiel View-Elemente im Code zusammen, so wird es schwer, sein Werk ohne GUI zu testen. Deshalb sollte man sich dann auch überlegen, ob es nicht sinnvoller wäre, ein eigenes Steuerelement zu bauen, sodass man Aussehen und Verhalten wieder trennen kann.

Will man zum Beispiel eine Matrix darstellen, die sich dynamisch verändert, so könnte man dafür ein eigenes Element programmieren. Das Matrix Model kann man gut testen, damit ist es kein Problem sicherzustellen, dass im ViewModel die richtige Matrix steht. Das Steuerelement würde dann aus dem ViewModel die benötigte View bauen. Für diesen kleineren Teil wären – zugegebenermaßen – GUI-Tests nötig, aber für die Berechnung der Matrix kann man wieder auf Unit-Tests zugreifen.

Grundsätzlich sollte man darauf achten, möglichst wenig Code in den Codebehind zu schreiben, da dieser schlecht testbar und noch schlechter wiederverwendbar ist. In den meisten Fällen findet sich eine Möglichkeit, entweder diesen Code in das ViewModel zu verschieben oder eine wiederverwendbare Komponente zu schreiben, die den fraglichen Code enthält.

Alle genannten Techniken zeigen eines ganz deutlich: Auch die Präsentationsschicht lässt sich einfach und mit wenig Aufwand testgetrieben entwickeln. Das macht Spaß und senkt auf lange Sicht auch die Bug-Quote. [bl]