Marcel de Vries

De weg naar continuous delivery: het maken van onderhoudbare UI-testen met Microsoft CodedUI

Continuous delivery is momenteel een trends in de IT markt. Continuous delivery staat voor een aanpak waarbij we er niet alleen naar streven iedere sprint de software af te hebben, maar deze software ook daadwerkelijk in productie te hebben gebracht. Hierbij is het o.a. van groot belang dat we de tijd die nodig is voor het testen van onze applicaties, significant weten te reduceren. Hierbij wordt gebruik gemaakt van het geautomatiseerd testen van applicaties.

Microsoft CodedUI is een technologie die dit mogelijk maakt. Met CodedUI kun je een eindgebruiker simuleren die gebruik maakt van de gebruikersinterface van een applicatie. Kenmerkend aan het testen van gebruikersinterfaces is dat ze in het algemeen een stuk langzamer zijn dan unit testen en dat ze veel gevoeliger zijn voor veranderingen in de applicatie. Toch is deze vorm van testen onontbeerlijk. Met de huidige trend richting web-applicaties ontkom je bijvoorbeeld niet aan de cross-browser-problematiek. Het valideren of je applicatie op verschillende browsers hetzelfde gedrag vertoont, is één van de mogelijkheden die CodedUI biedt.

Omdat UI-testen extra gevoelig zijn voor veranderingen in de gebruikersinterface en omdat veranderingen regelmatig zullen plaatsvinden in de levenscyclus van een applicatie, is het extra belangrijk dat de testen die gemaakt worden, goed onderhoudbaar zijn.

In dit artikel zal ik ingaan op de vraag hoe je met Microsoft CodedUI goed onderhoudbare testen kunt maken met behulp van het Page Object pattern. Dit pattern heeft in de praktijk bewezen deze doelstelling relatief eenvoudig realiseerbaar te maken.

Introductie CodedUI

Microsoft Visual Studio CodedUI is een technologie gebaseerd op MSTest, die het mogelijk maakt UI-testen uit te voeren. Met CodedUI is het mogelijk zowel Windows-Forms-applicaties, WPF applicaties, Xaml-gebaseerde Store-applicaties als web-applicaties te testen op dezelfde manier waarop je normaal gesproken Unit testen schrijft.

CodedUI ondersteunt het concept van Record & Playback, waarbij je tools krijgt die het mogelijk maken schermacties in de applicatie op te nemen en later weer af te spelen. In dit artikel ga ik er niet op in hoe Record & Playback werkt, maar zal ik alleen ingaan op de vraag hoe je zogenaamde CodeFirst-testen opzet. Hierbij schrijf je zelf de testen direct tegen het CodedUI object model.

Om een eerste indruk te geven hoe dit werkt, beginnen we met een simpel voorbeeld waarbij we een zoekactie uitvoeren op de Bing website. Om te beginnen moeten we de applicatie die we willen testen opstarten. Hiervoor heeft Microsoft de ApplicationUnderTest class geïntroduceerd. Voor web-testen is er een gespecialiseerde versie van deze class namelijk de BrowserWindow class. Deze encapsuleert de browser op je testmachine en biedt ook de mogelijkheid verschillende browsers (Chrome en Firefox) te starten als je de cross browser extenties voor CodedUI hebt geïnstalleerd vanuit de Visual Studio Gallery.

Zodra de applicatie is gestart, kun je aan de slag gaan met het zoeken van scherm elementen. Dit doe je door afhankelijk van het type applicatie een CodedUI control class te instantiëren die het zoeken en besturen van een schermcontrol abstraheert. We noemen dit type control in de rest van het artikel een search-control om goed het onderscheid te kunnen maken tussen controls die elementen zijn van een gebruikersinterface en de controls die CodedUI gebruikt voor de besturing van de applicaties die worden onderworpen aan een test. Voor WindowsForms applicaties maak je gebruik van search-controls met de prefix: Win. Voor Wpf applicaties van search-controls met: Wpf, voor Xaml-gebaseerde Store-applicaties zijn search-controls met prefix Xaml beschikbaar en last but not least voor web-applicaties zijn er search controls beschikbaar met de prefix Htlm. Al deze search-controls erven van een gemeenschapplijke baseclass, UITestControl. In onderstaande figuur is deze hiërarchie weergegeven.

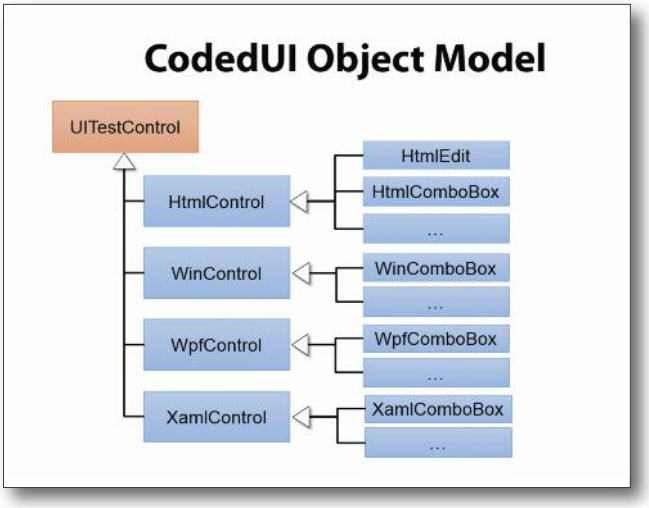

Fig. 1: CodedUI Object model

Nadat je een instantie maakt van het type search-control dat je wilt vinden, moet je criteria aangeven waarop een control gezocht kan worden. Hiervoor gebruik je, afhankelijk van de applicatie-technologie, een specifiek identificatie-mechanisme. Voor Web controls is dit bij voorkeur het HTML Element Id, voor Wpf controls de Automation identifier (dat is de xaml:name attribuut) en voor Windows applicaties de “Accessible Name”- property van het control.

In codevoorbeeld 1 start ik een web browser en zoek op de Bing.com website naar de search edit box waar ik vervolgens een zoek query invul. Daarna zoek ik de knop op de pagina om de zoek actie in Bing te starten. Na het klikken kunnen we valideren of we naar de resultatenpagina zijn genavigeerd.

```
[TestMethod]
public void CodedUITestMethodFirstTest()
{
  BrowserWindow _bw = BrowserWindow.Launch(new
Uri("http://www.bing.com"));
  EnterSearchValue(_bw, "Pluralsight CodedUI trai-
ning");
  ClickSearch(_bw);
  // valideer of we op een result page zijn geland….
}

private static void EnterSearchValue(BrowserWindow
_bw, string value)
{
  //create CodedUI Html Control we want to interact
with
  HtmlEdit searchBox = new HtmlEdit(_bw);
  searchBox.SearchProperties.Add(HtmlEdit.PropertyNa-
mes.Id, "sb_form_q");
  // interaction with the control, will trigger the
search
  searchBox.Text = value;
}

private static void ClickSearch(BrowserWindow _bw)
{
  HtmlInputButton searchButton = new HtmlInputBut-
ton(_bw);

searchButton.SearchProperties.Add(HtmlInputButton.Pro-
pertyNames.Id, "sb_form_go");
  //Use the Mouse static methods to click on controls
we search for
  Mouse.Click(searchButton);
}
```

Listing 1

Basis concept van search properties en interactie met controls

Als je kijkt naar hoe CodedUI naar controls zoekt dan gaat dat als volgt: Iedere search-control heeft een constructor. In de constructor kun je een ander search-control meegeven als de scope waarbinnen de engine moet zoeken naar een ander control. Dus als je wil zoeken in een browserwindow, dan geef je de instantie van de BrowserWindow-class mee als zoekscope. Door de scope te beperken zorg je ervoor dat er sneller wordt gezocht in het HTML-document. De engine gaat pas zoeken op het moment dat er interactie met het search-control is. Dit kan bijvoorbeeld een Mouse.Click() zijn, maar kan ook bijvoorbeeld het zetten van een property zijn op een search-control. Denk hierbij bijvoorbeeld aan het zetten van de text-property op een HtmlEdit search-control.

Bij het zoeken wordt er gezocht naar alle search-properties die in de SearchProperties collectie zijn gestopt van de search-control. Dus je kan meerdere zoekcriteria meegeven als een control bijvoorbeeld niet op ID kan worden gevonden. Alle zoekcriteria worden als een AND-search uitgevoerd. Als het zoeken resulteert in één control, wordt deze direct gebruikt. Indien er meer controls worden gevonden, worden FilterProperties gebruikt om ervoor te zorgen dat we maar één control overblijft. Filter properties zijn vergelijkbaar met search properties alleen worden deze alleen gebruikt indien filtering nodig is voor het zoek proces. Blijven er nog steeds meerdere controls over na het toepassen van de filter-properties, dan wordt de eerste in de lijst die gevonden is, teruggegeven. Indien de control niet wordt gevonden, zal de engine het opnieuw proberen (Er kan immers iets zijn veranderd in de UI in de tussentijd) en dit blijven doen totdat de default time-out van 2 minuten is verstreken.

Kenmerken van onderhoudbare testen

Nu we weten hoe CodedUI in de basis werkt, kunnen we gaan kijken naar de onderhoudbaarheid van de testen. Hiervoor passen we standaard een aantal principes toe. Principe één is: SOLID. SOLID is een algemeen geaccepteerde manier om code te schrijven die beter onderhoudbaar is. Het gaat te ver om dat hier in detail uit te leggen. Daarom pak ik het meest gebruikte concept hieruit, namelijk: het Single Responsibility principe. Dit betekent dat een class maar één verantwoordelijkheid heeft. Dit probeer je niet alleen voor classes te doen, maar ook voor methodes van die class. Verder gebruik je het DRY principe, dat staat voor Dont Repeat Yourself. Dit is om “Copy & Paste”-coding te voorkomen, hetgeen helaas opvallend veel vaker wordt toegepast voor het schrijven van testen. Het laatste Principe dat we toepassen is DAMP hetgeen staat voor Descriptive And Meaningful Phrases. Dit laatste passen we met name toe voor het schrijven van het uiteindelijke testscenario. Hierbij is het doel dat je de test scenario’s goed leesbaar opschrijft zodat ze min of meer leesbaar zijn als een zin. Dit is mogelijk door met behulp van het page object pattern een abstractie te schrijven voor je applicatie-onder-test die resulteert in een fluent API.

Het Page object pattern

Het page object pattern betreft een abstractie waarbij je de keuzes maakt om schermonderdelen, of in het geval van webapplicaties, webpagina’s laat aansturen door een specifieke class. Deze class is dan het zogenaamde page object. Vanuit het SOLID principe maken we dus één class verantwoordelijk voor één pagina en deze zal alle interactie met die achterliggende web pagina abstraheren. Voor alle acties die je op een pagina of schermonderdeel kunt uitvoeren, maken we publieke methodes die deze acties dan beschikbaar stellen. Verder zorg je ervoor dat ieder pageobject methodes heeft die condities evalueren die je wilt controleren in je testen. Deze methodes retourneren dus altijd een boolean-waarde. Door deze methodes op het page-object te plaatsen, zorg je ervoor dat het page-object alle kennis heeft van de pagina die hij abstraheert en zorg je ervoor dat de scenario’s die je maakt als test, geen kennis hebben van de pagina’s. Om een goed beeld te geven hoe dit werkt in code, heb ik een voorbeeld uitgewerkt dat toepasbaar is op de “TailSpin Toys”-website. Deze website is veel gebruikt in demonstraties van Microsoft en onderdeel van te downloaden test-virtual-machines. Als voorbeeld heb ik de Home Page, de Category Page, de Details Page en de Shopping Cart Page uitgewerkt. De flow van de pagina’s is in onderstaande figuur weergegeven.

Fig. 2: Home page

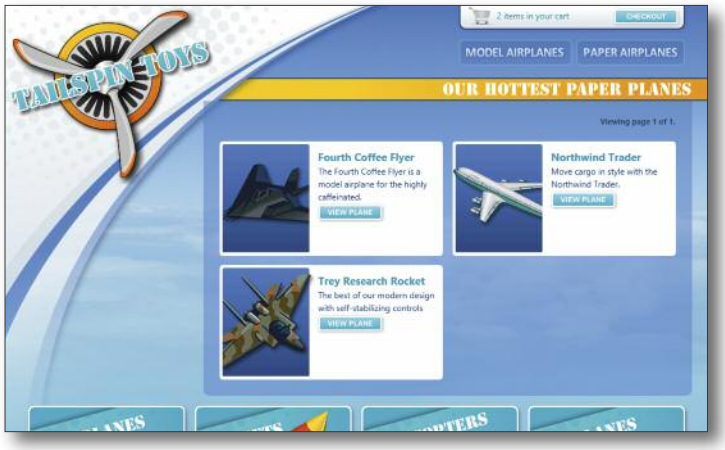

Fig. 3: Category page

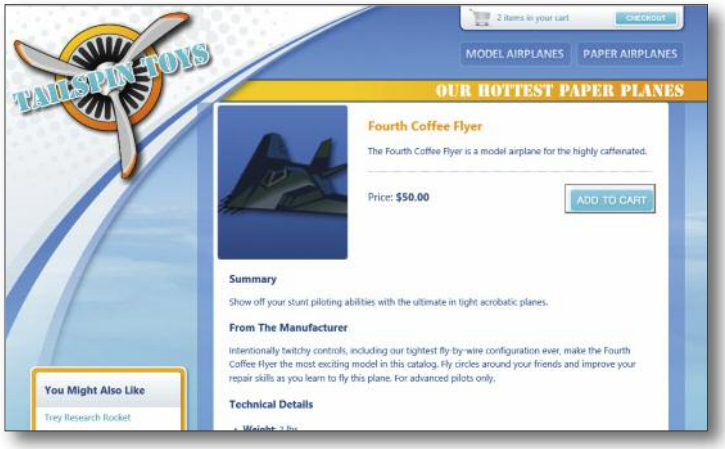

Fig. 4: Details page

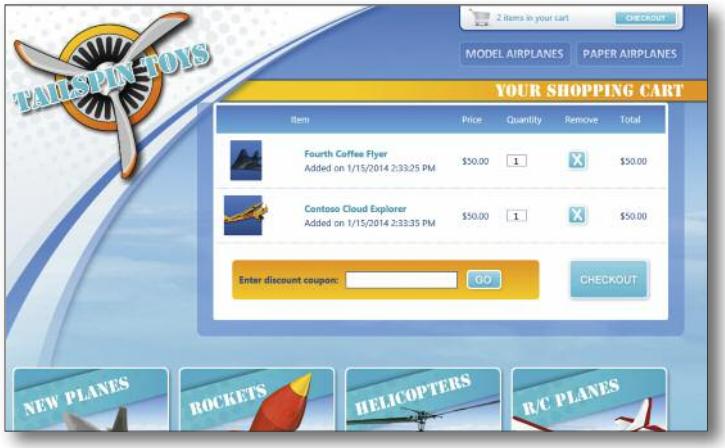

Fig. 5: Shopping Cart page

Het Home Page object kent een aantal acties, namelijk: het terug gaan naar de home page, het kiezen van een categorie en het navigeren naar de shopping cart page. Dit wordt vertaald in methodes als NavigateToShoppingCart(), NavigateToHome(), SelectCategory(string category). Om met behulp van de page-objects een fluent API te maken, zorg je ervoor dat iedere methode die je maakt, altijd een nieuw page-object terug geeft. Dus in dit geval zou de Select Category een nieuw CategoryPage object moeten retourneren. In Code Voorbeeld 2 is te zien hoe je de HomePage class kunt uitwerken. Zoals je kunt zien heeft het HomePage object nog een extra methode om de testen te starten: hiervoor is de static methode StartApplication() gemaakt, die dan zelf het HomePage Object terug-geeft. Verder zie je dat het WebBrowser window wordt vastgehouden en wordt gebruikt bij het zoeken naar controls. Dit doen we om de scope van het zoeken te beperken tot de web pagina in de browser.

```
public partial class HomePage
{
   private BrowserWindow _browserWindow;
   public HomePage(BrowserWindow browserWindow)
   {
       _browserWindow = browserWindow;
   }

   public static HomePage StartApplication()
   {
       var bw =
BrowserWindow.Launch("http://www.tailspintoys.com/");
       // zorg er voor dat de browser altijd maximi-
zed is
       bw.Maximized = true;
       return new HomePage(bw);
   }

   public CategoryPage SelectCategory(string catego-
ryName)
   {
       HtmlControl link = new HtmlControl(_browser-
Window);
       link.SearchProperties.Add(HtmlControl.Proper-
tyNames.InnerText, categoryName);
       HtmlHyperlink innerLink = new
HtmlHyperlink(link);
       Mouse.Click(innerLink);
       return new CategoryPage(_browserWindow);
   }
}
```

Listing 2

Indien we dat niet doen, zal het zoeken altijd starten op de desktop van windows en moet je een combinatie van CodedUI Windows-Search-controls en Html-search-controls gebruiken om bij een element op een webpagina te komen. Zodra we de SelectCategory Methode aanroepen krijgen we een CategoryPage-object terug. Op de CategoryPage kan je een vliegtuig selecteren. Hiervoor maken we een methode SelectAirplane(string planeName) die een AirplaneDetail page object terug geeft. AirplaneDetail heeft op zijn beurt de optie om het geselecteerde vliegtuig aan de cart toe te voegen, dus daar maken we dan een methode AddToCart() voor die een ShoppingCartPage-object teruggeeft. De ShoppingCartPage biedt de mogelijkheden items uit de cart te verwijderen en naar het afrekenen te gaan. Hiervoor maken we dan methodes als RemoveitemFromShoppingBasket(string nameItem) en Checkout(). De Remove() methode laten we opnieuw een ShoppingCart page object retourneren, aangezien er niet van de pagina wordt weggenavigeerd. In codevoorbeeld 3 zijn de resterende page objects uitgewerkt.

```
public class CategoryPage
{
  private BrowserWindow _browserWindow;
  public CategoryPage(BrowserWindow browserWindow)
  {
    _browserWindow = browserWindow;
  }
  public AirplaneDetailPage SelectAirplane(string air-
planeName)
  {
    var hyperlink = new HtmlHyperlink(_browserWindow);
    hyperlink.SearchProperties[HtmlHyperlink.Property-
Names.InnerText] = airplaneName;
   Mouse.Click(hyperlink);
```

```
    return new AirplaneDetailPage(_browserWindow);
  }
}

public class AirplaneDetailPage
{
  private BrowserWindow _browserWindow;
  public AirplaneDetailPage(BrowserWindow browserWindow)
  {
     _browserWindow = browserWindow;
  }
  public ShoppingCartPage AddItemToShoppingCart()
  {
     var link = new HtmlInputButton(_browserWindow);
      link.SearchProperties.Add(HtmlButton.PropertyNa-
mes.Type, "submit");
       Mouse.Click(link);
       return new ShoppingCartPage(_browserWindow);
  }
}
public partial class ShoppingCartPage
{
  private BrowserWindow _browserWindow;
  public ShoppingCartPage(BrowserWindow browserWindow)
  {
    _browserWindow = browserWindow;
  }
  public CheckOutPage CheckOut()
  {
    var link = new HtmlInputButton(_browserWindow);
     link.SearchProperties.Add(HtmlButton.PropertyNa-
mes.DisplayText,"Checkout");
     link.SearchProperties.Add(HtmlButton.PropertyNa-
mes.Type,"button");
    Mouse.Click(link);
    return new CheckOutPage(_browserWindow);
  }
}
public partial class CheckOutPage
{
  private BrowserWindow _browserWindow;
  public CheckOutPage(BrowserWindow broserWindow)
  {
    _browserWindow = broserWindow;
  }
  public bool IsPageValid()
  {
     return FindStreet1Edit().TryFind();
  }
  private HtmlEdit FindStreet1Edit()
  {
     HtmlEdit edit = new HtmlEdit(_browserWindow);
       edit.SearchProperties.Add(HtmlEdit.PropertyNa-
mes.Id, "Street1");
     return edit;
  }
}
```

Listing 3

Voor het laatste page-object heb ik alleen de query methode geïmplementeerd die we vanuit onze test methode gebruiken om de controle uit te voeren of de pagina correct is. Door een methode te maken IsPageValid() en aan de implementatie van het page object over te laten wat een geldige pagina is, wordt geen kennis “gelekt” naar de testmethode en hebben aanpassingen in een pagina alleen gevolgen voor de page-objecten. Dus door page-objecten te gebruiken hebben we een zeer eenvoudige onderhoudsrelatie. Voor iedere pagina die is aangepast, moet je het bijbehorende page object controleren of dit nog steeds werkt.

Doordat we page-objecten hebben voorzien van methoden die zelf weer page-objecten teruggeven, hebben we de mogelijkheid gecreëerd de test scenario’s beschrijvend op te stellen. Stel je wil het volgende scenario testen: We browsen eerst naar een categorie, dan selecteren we een vliegtuig om een vliegtuigdetail te bekijken waarna we ervoor kiezen het vliegtuig aan te schaffen. Als laatste gaan we naar de kassa om af te rekenen. Dit scenario kunnen we nu volledig uitschrijven met behulp van een fluent API. In codevoorbeeld 4 is het beschreven test scenario uitgewerkt op basis van het gemaakte object model en de daaruit volgende fluent API. Dit is dus een test scenario wat we hebben geschreven conform het DAMP principe.

```
[TestMethod]
public void BuyModelAirplaneAndCheckOut()
{
   Assert.IsTrue(

   HomePage.StartApplication()
           .SelectCategory("Model Airplanes")
           .SelectAirplane("Fourth Coffee Flyer")
           .AddItemToShoppingCart()
           .CheckOut()
           .IsPageValid(),"Expected to be on page for
checkout");
}
```

Listing 4

Conclusie

Met CodedUI is het mogelijk om testautomatisering door te voeren op eindgebruikerstesten via de gebruikersinterface van applicaties. Deze testen zijn in het algemeen lastig te onderhouden, maar als we een aantal principes toepassen in combinatie met het Page Object Pattern, is het mogelijk geworden onze testen dusdanig te bouwen dat ze goed te onderhouden zijn. We maken onze page-objecten op basis van de principes SOLID en DRY en door gebruik te maken van de techniek waarbij iedere methode een nieuw page-object teruggeeft, is het ook mogelijk geworden zogenaamde DAMP testscenario’s te schrijven. Door je testen te automatiseren en dit op een goed onderhoudbare wijze te doen, is het doel continuous delivery weer een stapje dichterbij gekomen. •

Marcel de Vries

Marcel spends most of his time helping customers build enterprise systems based on Microsoft Technology. He has been working with Microsoft technology since he graduated in Computer Science in 1996. He started mainly with C/C++ and MFC.When Microsoft launched its new .NET platform in he immediately used it to write the first commercial application to go life in the Netherlands based on ASP.NET. Marcel writes articles and whitepapers on .NET, Application Lifecycle Management and Mobile solutions for MSDN, The Architecture journal and local magazines like Microsoft .NET magazine. Marcel is a frequent speaker at conferences like Microsoft TechDays, Visual Studio Live!, Microsoft Tech Ed and local user group events. Marcel is Technology Manager at Info Support and in his role he works as consultant in the Architect role for a variety of Financial and Insurance companies. He also teaches courses on topics like Visual Studio ALM, .NET , Mobile and Web development at the Info Support Knowledge Center. Marcel is awarded the Microsoft ALM MVP award and is a Microsoft Regional Director.