Test-Automation mit Selenium WebDriver

Der große Formular-Check

Webanwendungen wiederholt von Hand zu testen ist monoton und ineffizient. Nach Erweiterung der Grundfunktionen nimmt WebDriver Ihnen lästige Arbeiten wie das Ausfüllen von Formularen ab.

Auf einen Blick

Martin Gossen ist als IT-Berater bei iks tätig. Er erstellt individuelle Geschäftsanwendungen mit C#, ASP.NET und SQL Server. Besonderen Fokus legt er dabei auf automatisiertes Testen und Usability-Analysen. Sie erreichen ihn unter m.gossen@iks-gmbh.com.

Inhalt

- Selenium-Tools für die GUI-Automatisation.
- Das Tool Selenium WebDriver im Detail.
- Beispiel eines automatisierten Tests.

dnpCode
A1305Selenium

Unit-Tests sind zu aufwendig. Für Integrationstests fehlt die Zeit. Systemtests können auch noch am Projektende gemacht werden. Und ohnehin, Sie machen doch Ihre Arbeit von Anfang an richtig – warum also so umfangreich (und wiederholt) testen?

Diese – hier überspitzt dargestellte – Ansicht kommt Ihnen vermutlich aus einigen IT-Projekten bekannt vor. Als professioneller Entwickler wissen Sie zwar, dass hochwertige Software nur mithilfe von Qualitätssicherungsmaßnahmen entstehen kann, doch gelegentlich ist etwas Überzeugungsarbeit nötig, um den Nutzen solcher Maßnahmen zu vermitteln.

Ein möglicher Ansatz: Führen Sie einen automatisierten GUI-Test vor. Ein Formular, das sich wie von selbst mit Daten füllt, automatisch auf richtiges Verhalten geprüft und schließlich abgesendet wird, sagt mehr als tausend Worte; besonders wenn dies mit fünf verschiedenen Datensätzen in jeweils drei unterschiedlichen Clients geschieht. Noch besser: Diese Tests können zusammen mit der entwickelten Software ausgeliefert und im Rahmen eines Abnahmetests eingesetzt werden, sodass ein unmittelbarer Zeitgewinn sichtbar wird. Dies kommt allen Beteiligten zugute – eine höhere Akzeptanz von regelmäßigen Systemtests und damit oft der Qualitätssicherung allgemein ist ein wichtiger Beitrag zum Gelingen des Projekts.

Das Tool

Im .NET-Umfeld stehen diverse Tools für die GUI-Automation zur Verfügung, von denen vier in [1] vorgestellt wurden. Dabei bietet sich als Testumgebung für Webanwendungen das weitverbreitete und frei verfügbare Produkt Selenium [2] an. Genau genommen handelt es sich bei Selenium um eine Sammlung mehrerer Tools, die in Tabelle 1 aufgeführt sind.

Dieser Artikel befasst sich mit Selenium WebDriver (Selenium 2), einer Implementierung des WebDriver-APIs, das als Entwurf für eine W3C-Spezifikation vorliegt [3]. Ähnlich Selenium IDE imitiert Selenium WebDriver das Verhalten eines realen Anwenders, indem es mit dem Webbrowser beziehungsweise der geladenen Webseite interagiert. Elemente der Webseite (DOM-Elemente) werden dabei über Selektoren ausgewählt und anschließend ausgelesen oder manipuliert. Dabei sind alle Aktionen in der gleichen Weise eingeschränkt wie bei einem echten Anwender; zum Beispiel kann ein Button, der ausgeblendet oder inaktiv ist, auch von den beiden Automations-Tools nicht angeklickt werden.

Während Selenium IDE als reines Firefox-Plug-in eher die Funktion eines aufgebohrten Makro-Recorders erfüllt, lässt sich Selenium WebDriver für anspruchsvolle Steuerungsaufgaben auch in anderen Browsern und damit für professionelle Systemtests von Webanwendungen einsetzen. Technisch gesehen verfolgt Selenium WebDriver auch den solideren Ansatz, denn es nutzt im Gegensatz zu Selenium IDE die in aktuellen Browsern bereits integrierten Funktionen zur Automation [4].

Im Folgenden soll der Begriff „WebDriver“ nicht die Schnittstelle, sondern das Selenium-Tool bezeichnen. Eine Dokumentation zum kompletten Selenium-API finden Sie in [5].

Tabelle 1

Selenium Tools.

Tool	Beschreibung
Selenium IDE	Ein Firefox-Add-on, das als Makro-Recorder fungiert. In die Makros können Überprüfungen mittels Assert und Verify eingebunden werden. Zudem ist das schrittweise Abspielen von Testfällen und das Setzen von Breakpoints möglich.
Selenium Client API	Erweitert Selenium IDE um diverse Programmiersprachen (inklusive C#).
Selenium RC (Remote Control)	Ein Proxy-Server, der von einem Selenium-Client-Treiber ferngesteuert wird. Ermöglicht die Testautomatisierung von GUI-Tests in Kombination mit Continuous Integration. Ist offiziell als deprecated (überholt) gekennzeichnet.
Selenium WebDriver (Selenium 2)	Der Nachfolger von Selenium RC mit objektorientiertem API. Spricht den Browser direkt über dessen Automations-API an.
Selenium Grid	Eine Erweiterung, die eine parallele Ausführung von Tests auf mehreren Servern ermöglicht.

Installation und Einrichtung

Am einfachsten legen Sie mit WebDriver los, indem Sie in Visual Studio ein neues Projekt vom Typ *Klassenbibliothek* erstellen (vorzugsweise nicht auf Laufwerk *C:*, siehe unten) und über die Paket-Manager-Konsole die folgenden NuGet-Pakete hinzufügen:

```
PM> Install-Package Selenium.WebDriver
PM> Install-Package Selenium.Support
```

Es ist außerdem sinnvoll, ein Test-Framework wie NUnit [6] zu installieren. Es soll in diesem

Listing 1

Ein einfacher WebDriver-Test.

```
using System;
using NUnit.Framework;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;

namespace SeleniumDemo.DotNetPro.Test
{
  public class DotNetProTest1
  {
    [Test]
    public void FindSeleniumArticle()
    {
      // Instantiate browser driver
      using (IWebDriver driver = new FirefoxDriver())
      {
        // Set timeout
        TimeSpan waitTimespan = TimeSpan.FromSeconds(15);
        driver.Manage().Timeouts().ImplicitlyWait(waitTimespan);

        // Navigate to target URL
        driver.Url = "http://www.dotnetpro.de";

        // Get page elements for searching
        IWebElement searchBox =
          driver.FindElement(By.Id("TB_SearchTerm"));
        IWebElement searchButton =
          driver.FindElement(By.Id("BT_Search"));

        // Execute search
        searchBox.Clear();
        searchBox.SendKeys("Selenium");
        searchButton.Click();

        // Get element from result page
        IWebElement searchResultsArea =
          driver.FindElement(By.Id("ctl09_PNL_SearchResult"));

        // Assert correct result
        Assert.IsTrue(searchResultsArea.Text.Contains(
          "Fahren Sie schon Automatik?"));

        // Finish up
        try
        {
          driver.Quit();
        }
        catch
        {
          // Ignore if browser can't be closed
        }
      }
    }
  }
}
```

Zusammenhang zwar nicht für Unit-Tests eingesetzt werden, eignet sich aber gut für die Konfiguration und Ausführung der Einzeltests. Eine Einführung in NUnit finden Sie in [7]. Sie können anschließend die installierte NUnit-Assembly Ihrem Projekt hinzufügen oder sich wieder bei NuGet bedienen:

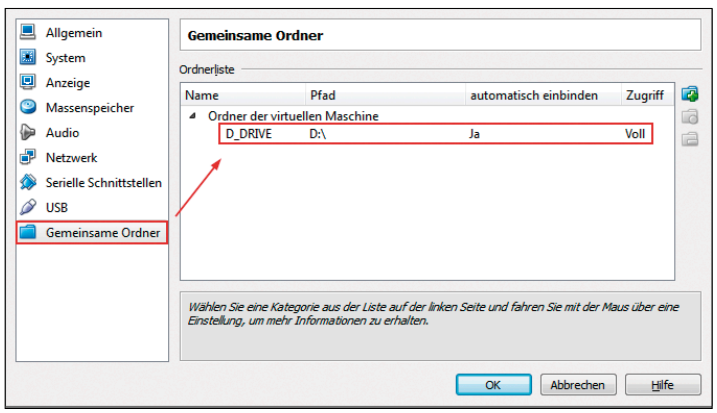
```
PM> Install-Package NUnit
```

Je nachdem, in welchen Browsern Sie Ihre Tests ausführen wollen, sind noch externe Browser-Treiber notwendig. Einzelheiten hierzu finden Sie auf der Selenium-Download-Seite; allerdings sind die dortigen Angaben und verknüpften Versionen nicht immer aktuell. So soll man der Windows-Umgebungsvariablen *%PATH%* den Speicherort des Internet-Explorer-Treibers hinzufügen, was aber nicht unbedingt nötig ist. Hingegen sollte man darauf achten, dass die Browser-Treiber nicht durch eine Firewall behindert werden; unter Windows 7 erhalten Sie beim ersten Zugriffsversuch eine entsprechende Meldung.

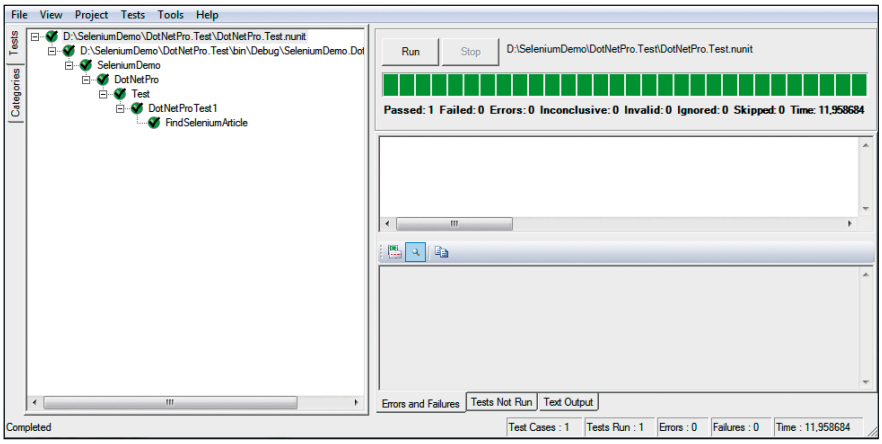
Den IE-Treiber finden Sie in [8], den Chrome-Treiber in [9]. Für Firefox ist kein weiterer Treiber nötig. Beachten Sie, dass neuere Treiber eventuell nicht mit alten Versionen der Browser funktionieren; während die ZIP-Datei des Chrome-Treibers die Versionsnummer im Dateinamen trägt, müssen Sie beim Internet Explorer im Zweifelsfall mit älteren Treiberdateien experimentieren. Legen Sie die Treiberdateien ins Assembly-Verzeichnis des Visual-Studio-Projekts (zum Beispiel */bin/Debug*).

Wenn Sie mit genau festgelegten Browserversionen testen müssen, können Sie einen virtuellen Rechner zum Beispiel mit Oracles VM VirtualBox [10] aufsetzen und auf diesem die gewünschten Versionen installieren (automatische Programm-Updates ausschalten!). Wenn Sie Ihr Visual-Studio-Projekt nun beispielsweise auf Laufwerk *D:* angelegt haben, können Sie dieses in der virtuellen Maschine als gemeinsamen Ordner einbinden (siehe **Abbildung 1**).

Dies hat den Vorteil, dass Konfigurationspfade auf beiden Rechnern gleich sind und die Tests jederzeit aus der virtuellen Maschine wie auch Ihrem Entwicklerrechner heraus ausgeführt werden können (sofern beide mit den gleichen Browser-Treibern zurechtkommen).



[Abb. 1] Einbinden eines gemeinsamen Ordners in VirtualBox.



[Abb. 2] **Ein erfolgreicher NUnit-Test.**

Hello dotnetpro

Ein erster Test ist schnell entwickelt. Nachdem ein Browser-Treiber instanziert und mit einem Timeout versehen ist, können Sie schon zur Zielseite navigieren. Nun wählen Sie Seitenelemente (zum Beispiel Textfelder eines Formulars) anhand ihrer ID aus und führen Aktionen auf Ihnen aus. Anschließend testen Sie, ob die Seite wie erwartet reagiert hat, indem Sie die Werte weiterer Seitenelemente prüfen. **Listing 1** zeigt dies am Beispiel der Artikelsuche auf der dotnetpro-Startseite. In diesem Fall wird getestet, ob der in [1] genannte Artikel gefunden wird, wenn man eine Suche mit dem Stichwort „Selenium“ durchführt.

Nachdem Sie das Projekt kompiliert haben, können Sie ein NUnit-Projekt anlegen, die erzeugte Assembly des Visual-Studio-Projekts dem NUnit-Projekt hinzufügen und den Test ausführen. Wenn alles gut geht, können Sie WebDriver nun bei der Arbeit zusehen und am Ende ein positives Testergebnis abnicken (siehe **Abbildung 2**).

Selektoren und Aktionen

Nicht immer besitzen Seitenelemente eine eindeutige ID, anhand derer sie identifiziert werden können; oder aber die ID ändert sich vom einen zum anderen Seitenaufruf (zum Beispiel bei dynamisch erzeugten Controls). Aus diesem Grund bringt WebDriver eine Reihe von Selektor-Strategien mit, die in **Tabelle 2** aufgeführt sind. So lassen sich Elemente unter anderem anhand eines *Name*-Attributs, ihrer CSS-Klasse oder der Position in der DOM-Hierarchie ermitteln. Mithilfe von JavaScript ergeben sich weitere Möglichkeiten; zum Beispiel lassen sich die jQuery-Selektoren [11] verwenden. Da Selektoren auch mehrere Elemente zurückgeben können, ist eine weitere Filterung durch eigenen Code möglich. Aus Performance-Gründen sollten Sie aber möglichst eindeutige ID- und CSS-Selektoren benutzen. Sofern die zu testende Webseite unter Ihrer eigenen Kontrolle ist, empfiehlt es sich daher, alle relevanten Seitenelemente mit einer ID zu versehen.

Bei CSS- und XPath-Selektoren muss bedacht werden, dass diese von den Browsern unterschiedlich unterstützt werden; so wird Groß-/Kleinschreibung teilweise ignoriert, teilweise nicht. Einzelheiten hierzu finden Sie in [12].

Haben Sie das gewünschte Seitenelement über die *FindElement*- beziehungsweise *FindElements*-Methode gefunden (siehe **Listing 1**), steht es Ihnen als Objekt vom Typ *IWebElement* zur Verfügung und Sie können über seine Eigenschaften beispielsweise ermitteln, ob es momentan angezeigt wird, ob es aktiv ist und wie sein Inhaltstext lautet. Über Methoden können Sie es unter anderem anklicken oder einen Text setzen. Wenn Sie wissen, dass es sich um ein Control mit selektierbaren Werten handelt (zum Beispiel ein Drop-down-Control), können Sie es in ein *SelectElement*-Objekt kapseln und auf diesem eine von mehreren *SelectBy*-Methoden aufrufen, um einen Wert zu setzen.

```
IWebElement dropdown =
  driver.FindElement(By.Id("ctrl1"));
SelectElement selectableDropdown =
  new SelectElement(dropdown);

// Possible "select by" methods
selectableDropdown.SelectByIndex(42);
selectableDropdown.SelectByValue("foo");
selectableDropdown.SelectByText("bar");
```

Tabelle 2

Selektoren für DOM-Elemente.

Selektor	Beschreibung
ClassName	*Class*-Attribut des DOM-Elements.
CssSelector	CSS-Selektor.
Id	*ID*-Attribut des DOM-Elements.
LinkText	Vollständiger Text eines Hyperlinks.
Name	*Name*-Attribut des DOM-Elements.
PartialLinkText	Partieller Text eines Hyperlinks.
TagName	Name des DOM-Elements.
XPath	XPath-Selektor.
JavaScript	Rückgabewert einer JavaScript-Funktion.

Das Entwurfsmuster Page Object

Den Prinzipien der Clean-Code-Entwicklung [13] zufolge sollte jede Anwendung das Single-Responsibility-Prinzip beachten. Die Entwickler von WebDriver haben sich das zu Herzen genommen und empfehlen den Einsatz des Page-Object-Musters [14]. Das Page Object repräsentiert die zu testende Webseite; dem steht eine separate Testklasse gegenüber. Innerhalb des Page Object lassen sich Felder über einen Selektor deklarativ an DOM-Elemente

Listing 2

Die Klasse für das Page Object der Startseite.

```
public class DefaultPage {
  const string url =
     "http://www.dotnetpro.de";
  const string pageTitle =
    "Herzlich willkommen!";

  // Page Elements
  [FindsBy(Using = "TB_SearchTerm")]
  IWebElement searchBox;

  [FindsBy(Using = "BT_Search")]
  IWebElement searchButton;

  // Properties
  public string Url {
    get { return url; }}
  public string PageTitle {
    get { return pageTitle; }}

  // Methods
  public SearchPage Search(
    string searchText) {
    // Execute search
    searchBox.Clear();
    searchBox.SendKeys(searchText);
    searchButton.Click();
    return new SearchPage();
  }
}
```

Listing 3

Die Klasse für das Page Object der Suchseite.

```
public class SearchPage
{
  const string pageTitle = "Suche";

  // Properties
  public string PageTitle {
    get { return pageTitle; }}

  // Page Elements
  [FindsBy(Using =
    "ctl09_PNL_SearchResult")]
  IWebElement searchResultsArea;

  // Properties
  public IWebElement SearchResultsArea
  {
    get { return searchResultsArea; }
  }
}
```

Listing 4

Die Testklasse für die Page Objects aus Listing 2 und 3.

```
public class DotNetProTest2
{
  [Test]
  public void FindSeleniumArticle()
  {
    // Instantiate browser driver
    using (IWebDriver driver = new FirefoxDriver())
    {
      // Set timeout
      TimeSpan waitTimespan = TimeSpan.FromSeconds(15);
      driver.Manage().Timeouts().ImplicitlyWait(waitTimespan);

      // Create and initialize page object
      DefaultPage defaultPage = new DefaultPage();
      PageFactory.InitElements(driver, defaultPage);

      // Navigate to target URL
      driver.Url = defaultPage.Url;

      // Make sure correct page was loaded
      Assert.AreEqual(driver.Title, defaultPage.PageTitle);

      // Execute search
      SearchPage searchPage = defaultPage.Search("Selenium");

      // Initialize resulting page
      PageFactory.InitElements(driver, searchPage);

      // Make sure correct page was loaded
      Assert.AreEqual(driver.Title, searchPage.PageTitle);

      // Assert correct result
      Assert.IsTrue(searchPage.SearchResultsArea.Text.Contains(
        "Fahren Sie schon Automatik?"));

      // Finish up
      try
      {
        driver.Quit();
      }
      catch
      {
        // Ignore if browser can't be closed
      }
    }
  }
}
```

binden; hierzu dient das *FindsBy*-Attribut. Weiterhin stellt das Page Object nach außen hin Eigenschaften und Methoden bereit, die für die Testklasse von Nutzen sind.

Ein Beispiel: Das Page Object einer Login-Seite könnte eine *LogOn*-Methode bereitstellen sowie eine Eigenschaft *IsLoggedOn*. Diese können von der Testklasse benutzt werden, um den Anmeldevorgang eines Anwenders zu automatisieren und anschließend auf Erfolg zu prüfen.

Das Page Object selbst sollte keinen Testcode enthalten, also kein *Assert* ausführen.

Hierbei gibt es allerdings eine Ausnahme: Bei der Instanzierung kann das Page Object sicherstellen, dass die richtige Webseite (und eventuell wichtige Seitenelemente) geladen wurde und bereit ist, auf Testeingaben zu antworten. Dies wird am einfachsten anhand des Seitentitels festgestellt; wenn dieser nicht eindeutig ist, kann auch auf Vorhandensein eines beliebigen DOM-Elements geprüft werden.

Das Zusammenspiel von Page Object und Testklasse veranschaulichen die Listings 2, 3 und 4.

Screenshots und Protokollierung

Bekanntlich können Tests auch fehlschlagen – ist es doch gerade Sinn eines Tests, Fehler aufzudecken. Um Hinweise auf die Fehlerursache zu erhalten, können Sie im entscheidenden Moment einen Screenshot der Webseite anfertigen lassen. Hierfür bringt WebDriver eigens eine *GetScreenshot*-Methode mit.

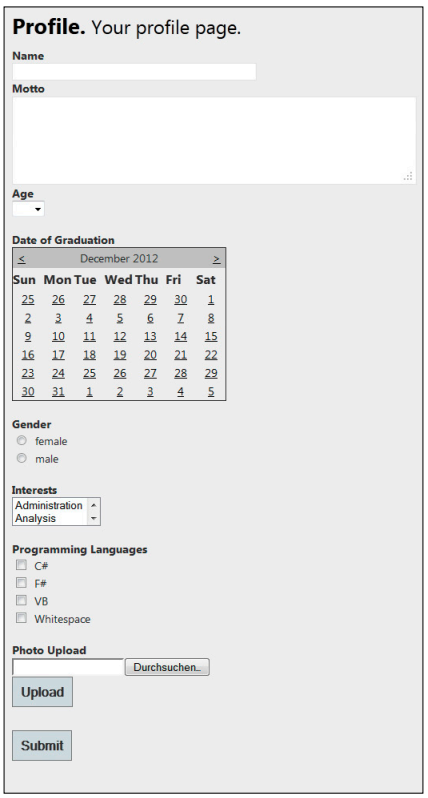
```
var screenshot =
  ((ITakesScreenshot)driver).GetScreenshot();
screenshot.SaveAsFile(path, ImageFormat.Png);
```

Für weitergehende Protokollierung des Fehlers oder auch des Erfolgsfalls für die Testdokumentation müssen Sie auf die .NET-Standardklassen oder auf Logging-Frameworks zurückgreifen.

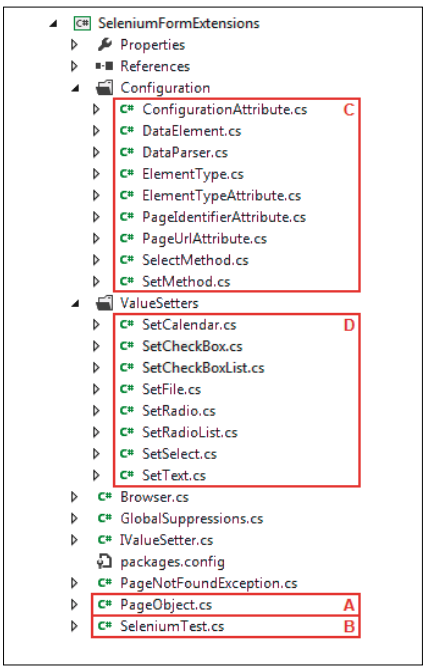
Eine Beispiellösung für Formular-Tests

Zu diesem Artikel finden Sie auf der Heft-DVD eine VS2012-Solution (.NET 4.5), die aus den folgenden vier Projekten besteht:

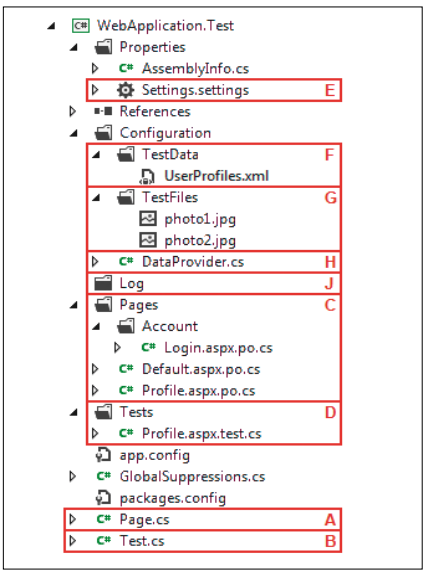
- *DotNetPro.Test* (enthält die abgedruckten Listings),
- *SeleniumFormExtensions* (Formular-Erweiterungen),
- *WebApplication* (eine Beispiel-Website),
- *WebApplication.Test* (Tests für die Beispiel-Website).



[Abb. 3] Die zu testende Seite Profile.aspx.



[Abb. 4] Das Projekt SeleniumFormExtensions.



[Abb. 5] Das Projekt WebApplication.Test.

Um mit der Beispiellösung arbeiten zu können, erstellen Sie zunächst im Webserver (IIS) eine neue Site und weisen den Ordner *WebApplication* als Stammverzeichnis zu, sodass die Beispiel-Website als *localhost* erreichbar ist. Der Anwendungspool muss .NET 4.0 verwenden. Gegebenenfalls müssen Sie dem Benutzerkonto des Webservers noch Zugriff erteilen, indem Sie entsprechende Rechte auf dem Ordner einrichten. Auf dem Unterordner *Upload* werden zudem Schreibrechte benötigt. Ob die Website wie gewünscht funktioniert, sehen Sie, wenn Sie im Browser zu *localhost/Profile.aspx* navigieren und sich dort an der Website anmelden (User name: *TestUser*, Password: *TestPw*). Nun sollte die Zielseite *Profile.aspx* erscheinen (siehe **Abbildung 3**).

Der eigentliche Test besteht darin, das Formular auszufüllen, abzusenden und auf Korrektheit der übermittelten Werte zu prüfen. Der Test ist dann erfolgreich, wenn alle eingegebenen Werte auf der anschließenden Bestätigungsseite korrekt angezeigt werden. *Profile.aspx* ist absichtlich so konzipiert, dass verschiedene Typen von Formular-Controls ausgefüllt werden können.

Bei dem Projekt *SeleniumFormExtensions* handelt es sich um eine wiederverwendbare Klassenbibliothek, die Ihnen die Arbeit mit Webformularen erleichtert. Sie enthält im Wesentlichen folgende Klassen: (siehe **Abbildung 4**):

- Eine Basisklasse *PageObject*, die eine *FillForm*- und einige Hilfsmethoden zur Verfügung stellt, welche das Ausfüllen von Formularen und generell die Interaktion mit Seitenelementen vereinfachen *(A)*.
- Eine Basisklasse für Tests (*SeleniumTest*), die sich um die Browser-Treiber und die Protokollierung von Fehlern kümmert sowie das Navigieren zu Webseiten erleichtert *(B)*.
- Eine Reihe von Konfigurationsklassen, mit denen Sie Eigenschaften von Page Objects und das genaue Verhalten beim Ausfüllen der Formularfelder deklarativ festlegen können. Hier wird zudem das Einlesen von Testdaten aus XML-Dateien ermöglicht *(C)*.
- Strategien zum Setzen von Werten (*Value Setters*) für verschiedene Control-Typen *(D)*.

Hierauf aufbauend können Sie Ihre eigenen Tests erstellen, wie im Projekt *WebApplication.Test* demonstriert (siehe **Abbildung 5**):

- Die Klasse *Page* dient als Basisklasse für die eigentlichen Page-Object-Klassen. Somit repräsentiert die *Page*-Klasse die Master Page der Website und enthält folgerichtig eine Logout-Methode. Das entsprechende Seitenelement – der Log-Off-Hyperlink – befindet sich in der Master Page *(A)*.
- Die Klasse *Test* dient als Basisklasse für die konkreten Testklassen. In dieser Klasse erfolgt die Vorbereitung (Setup) und das Aufräumen (Teardown), das vor beziehungsweise nach jedem Test notwendig ist. Im Beispiel erfolgt hier die An- und Abmeldung an der Website *(B)*.
- Der Ordner *Pages* enthält die Page-Object-Klassen, hierarchisch gegliedert wie die entsprechenden Webseiten. Wie diese konfiguriert werden können, sehen Sie in **Tabelle 3**. Weitere Erläuterungen dazu finden Sie auch im Quellcode *(C)*.
- Der Ordner *Tests* beinhaltet die konkreten Testklassen. Hier können Sie im *TestFixture*-Attribut angeben, für welche Browser die Tests ausgeführt werden sollen, und im *TestCaseSource*-Attribut bestimmen Sie, woher die Testdaten stammen. Die Testmethoden benutzen die Schablonenmethode *Execute* ihrer Basisklasse, damit Fehler abgefangen und protokolliert werden können *(D)*.
- Unter *Properties/Settings.settings* konfigurieren Sie das Projekt. Die meisten Werte sind selbsterklärend. *TestUserName* und *TestUserPassword* sind Anmeldedaten, die für jeden Test benötigt werden; daher sind sie hier zentral abgelegt. *SourceCulture* und *TargetCulture* werden zum Umwandeln von Formaten – speziell von eingelesenen Datumswerten – verwendet *(E)*.
- *Configuration/TestData* enthält XML-Dateien mit den zu verwendenden Testdaten *(F)*.
- *Configuration/TestFiles* enthält Binärdateien zum Testen der Upload-Funktion *(G)*.

- Die Klasse *Configuration/DataProvider* besorgt die Testdaten und stellt sie den Testklassen zur Verfügung. Im Beispiel liefert die *UserProfiles*-Eigenschaft sukzessiv die Werte aller XML-Knoten namens *Profile* der verfügbaren XML-Dateien *(H)*.
- Der Ordner *Log* ist der Speicherort für Fehlerprotokolle und zugehörige Screenshots *(J)*.

Um die Beispiellösung in Aktion zu sehen, müssen Sie lediglich ein NUnit-Projekt für die Assembly *SeleniumDemo.WebApplication.Test.dll* anlegen und ausführen. Damit es in allen drei Browsern (Firefox, Internet Explorer, Chrome) klappt, müssen diese natürlich installiert sein. Je nach Browser-Version kann es auch nötig sein, die vorhandenen Browser-Treiber (*IEDriverServer Win32 2.28.0* und *chromedriver win 23.0.1240.0*) auszutauschen.

Fazit

Selenium WebDriver bringt eine Reihe solider Funktionen mit, die das Testen von Webanwendungen erleichtern. Im Zusammenspiel mit NUnit und etwas zusätzlicher Handarbeit lässt sich daraus ein hoch konfigurierbares System bauen, das Ihnen viel Tipparbeit beim Testen von Webformularen abnimmt und sich in einen Continuous-Integration-Prozess einbinden lässt. Nebenbei dient es als anschauliche Demonstration für wiederverwendbare Systemtests und kann damit erheblich zum Projekterfolg beitragen. **[bl]**

[1] Hendrik Lösch, Michael Hanslik, Ein Vergleich verbreiteter UI-Testing-Frameworks, Fahren Sie schon Automatik?, dotnetpro 11/2012, Seite 102 ff., www.dotnetpro.de/A1211CodedUI
[2] Selenium, http://seleniumhq.org
[3] W3C Entwurf WebDriver-API, www.w3.org/TR/webdriver
[4] Selenium WebDriver Dokumentation, www.dotnetpro.de/SL1304Selenium1
[5] Selenium WebDriver API, www.dotnetpro.de/SL1304Selenium2
[6] NUnit, www.nunit.org
[7] Tam Hanna, Unit-Tests mit NUnit, Ebenbürtiger Ersatz, dotnetpro 12/2012, Seite 110 ff., www.dotnetpro.de/A1212Testen
[8] Internet Explorer Driver Server, www.dotnetpro.de/SL1304Selenium3
[9] Chrome Driver, www.dotnetpro.de/SL1304Selenium4
[10] Oracle VM VirtualBox, www.virtualbox.org
[11] jQuery-Selektoren, www.dotnetpro.de/SL1304Selenium5
[12] Selenium, Locating UI Elements (WebElements): CSS- und XPath-Eigenheiten, www.dotnetpro.de/SL1304Selenium6
[13] Clean Code, www.clean-code-developer.de
[14] Selenium-Wiki: Page-Object-Entwurfsmuster, www.dotnetpro.de/SL1304Selenium7

Tabelle 3

Attribute zur Konfiguration von Page Objects.

Attribut	Zielelement	Eigenschaft	Beschreibung
PageIdentifier	Page-Object-Klasse	*Title*	Titel der Webseite. Wird benutzt, um festzustellen, ob die korrekte Webseite geladen wurde.
		ElementId	ID eines DOM-Elements der Webseite. Wird benutzt, um festzustellen, ob die korrekte Webseite geladen wurde.
PageUrl	Page-Object-Klasse	*Url*	URL der repräsentierten Webseite. Zu diesem URL navigiert der Browser-Treiber.
Configuration	*IWebElement*-Feld	*Delay*	Zeitverzögerung beim Setzen eines Wertes in Millisekunden. Nützlich, wenn das Element Zeit zum Reagieren benötigt.
		Tab	Name eines Formularreiters, der angeklickt wird, bevor ein Wert gesetzt wird.
		Click	Das Element wird vor dem Setzen eines Wertes angeklickt.
		Leave	Das Element verliert nach dem Setzen eines Wertes den Fokus.
		DataType	Datentyp des zu setzenden Wertes. Wird bei der Umwandlung von Werten in eine andere Culture benutzt.
		SourceCulture	Culture des eingelesenen Wertes.
		TargetCulture	Culture, in die ein Wert umgewandelt wird.
		Format	Format, in das ein Wert umgewandelt wird.
		ElementType	Control-Typ des Elements (zum Beispiel *text* oder *select*). Für jeden Typ kann eine eigene Strategie zum Setzen von Werten benutzt werden.
		SetMethod	Ansatz, der zum Setzen eines Wertes gewählt wird.
		SelectMethod	Ansatz, der zur Auswahl eines Wertes gewählt wird.