TESTAUTOMATION MIT WINDOWS APPLICATION DRIVER

Volle Fenster-Kontrolle

Microsofts Automationsserver eignet sich für umfassende GUI-Tests unter Windows.

Das Tool Selenium WebDriver [1], das der Automation von Webbrowsern dient, hat sich in der Entwicklerwelt etabliert, insbesondere für GUI-Tests von Webanwendungen [2, 3]. In unserer vernetzten Welt spielen mobile Apps aber inzwischen eine mindestens ebenso große Rolle wie Webanwendungen. Als Reaktion darauf rief die Entwicklergemeinde das Automations-Framework Appium [4, 5] ins Leben. Wesentlicher Teil von Appium ist ein auf Node.js basierender Server, der das WebDriver-Protokoll [6] unterstützt und um Funktionen zum Testen mobiler Apps erweitert. Somit lassen sich auch native und hybride Apps auf Android und iOS testen. Beispielsweise kann Appium Multi-Touch-Aktionen ausführen und das Drücken der Hardwaretasten von Mobilgeräten simulieren.

Microsoft ist auf diesen Zug aufgesprungen und hat seine eigene Variante eines Automationsservers entwickelt: den Windows Application Driver (WinAppDriver) [7]. Er nutzt Microsoft UI Automation, um Windows-Anwendungen beziehungsweise Apps aller wesentlichen Technologien (UWP, WPF, Windows Forms, MFC sowie klassisches WinAPI) programmatisch anzusteuern – siehe auch Kasten Technologien für Windows-Anwendungen.

WinAppDriver ist quelloffen und mit dem Appium-Protokoll kompatibel, das bedeutet, dass es mithilfe des Appium Windows Driver [8, 9] nahtlos in eine bestehende Appium-Testumgebung integriert werden kann. Wegen dieser Vorteile hat Microsoft WinAppDriver als offiziellen Nachfolger der Coded-UI-Test-Technologie angekündigt.

Dieser Artikel zeigt, wie Sie WinAppDriver für automatisierte GUI-Tests mit und ohne Appium einsetzen können. Die für die Testbeispiele verwendete Software ist: Windows 10 Pro 1803, Visual Studio Community 2017 mit den Workloads *.NET-Desktopentwicklung* und *Desktopentwicklung mit C++*, .NET 4.7.2, WinAppDriver 1.1, Appium.WebDriver 4.0.0.4-beta, MSTest 1.4.0, Selenium.WebDriver 3.141.0, Appium Desktop 1.9.1.

Testbeispiele und Installation

Bei der Vorbereitung für diesen Artikel wurde ein und derselbe Test mit einer Reihe von Beispielanwendungen durch-

Technologien für Windows-Anwendungen

Die **Universal Windows Platform (UWP)** ist die neueste Laufzeitumgebung für Anwendungen unter Windows 10. Mit UWP kann eine Anwendung mit einer einzigen Codebasis für unterschiedliche Klassen von Zielgeräten (PC, Mobilgeräte, Xbox und mehr) entwickelt und im Windows Store bereitgestellt werden.

Die **Windows Presentation Foundation (WPF)** ist ein Grafik-Framework und Fenstersystem des .NET Frameworks. Es wurde 2006 eingeführt und ermöglicht die Gestaltung von grafischen Benutzeroberflächen und die Integration von Multimedia-Komponenten und Animationen. WPF vereint DirectX, Windows Forms, Adobe Flash, HTML und CSS.

Windows Forms ist das erste GUI-Toolkit des .NET Frameworks. Windows Forms umschließt das klassische WinAPI durch Managed Code. Im Rahmen des Mono-Projekts steht es weitgehend auch unter Linux und macOS zur Verfügung.

Microsoft Foundation Classes (MFC) existieren seit 1992. MFC kann für die Programmierung von grafischen Benutzeroberflächen für Windows-Anwendungen mit C++ genutzt werden und dient dabei als objektorientierte Schnittstelle zum klassischen WinAPI.

WinAPI (Windows Application Programming Interface) ist die ursprüngliche Windows-Programmierschnittstelle ohne Objektorientierung. Sie existiert in verschiedenen Versionen (Win16, Win32, Win32s, Win64) und abstrahiert die nativen Funktionen des Betriebssystems.

Tabelle 1: Testbeispiele

Beispiel Nr.	Technologie	Anwendungsname	Quelle
1	UWP	Rechner	Vorinstalliert mit Windows 10
2	WPF	Calculator-Demo	https://github.com/Microsoft/WPF-Samples/tree/master/Sample%20Applications/CalculatorDemo
3	WinForms	Simple Calculator	https://code.msdn.microsoft.com/Simple-Calculator-fd7cb93c
4	MFC	Calculator	www.codeproject.com/Articles/3280/Calculator
5	Win32 API	Windows Calculator	https://mitxela.com/projects/windows_calculator

GUI aus Testbeispiel 1 (Bild 1)

geführt, die in Tabelle 1 aufgeführt sind. Bei allen fünf Beispielen handelt es sich um (Taschen-)Rechner-Anwendungen, mit denen eine Division ausgeführt wird. Anschließend wird auf das korrekte Rechenergebnis in der Anzeige geprüft.

Da das Vorgehen bei allen Beispielanwendungen im Wesentlichen identisch ist, beschränkt sich die Beschreibung im Folgenden auf Beispiel 1, die UWP-App Rechner, siehe Bild 1.

Wenn Sie Appium bereits einsetzen oder zukünftig einsetzen wollen, dann sollten Sie möglichst die WinAppDriver-Version nutzen, die mit Appium ausgeliefert wird, da die beiden Produkte aufeinander abgestimmt sind – zumindest in der Theorie.

In der Praxis ist es gelegentlich auch schon vorgekommen, dass die ausgelieferte Version inkompatibel war. In diesem Fall – oder auch, wenn Sie WinAppDriver ohne Appium einsetzen wollen – bekommen Sie die aktuellste Version unter [10]. Zur Installation führen Sie die Datei *WindowsApplicationDriver.msi* aus.

Für die Test-Entwicklung benötigen Sie zudem die .NET-Client-Bibliotheken für Appium, die Sie am besten per NuGet in Ihr Softwareprojekt einbinden. Legen Sie dazu zuerst in Visual Studio ein Projekt vom Typ *Komponententestprojekt (.NET Framework)* an und wählen Sie *Projekt | NuGet-Pakete verwalten*.

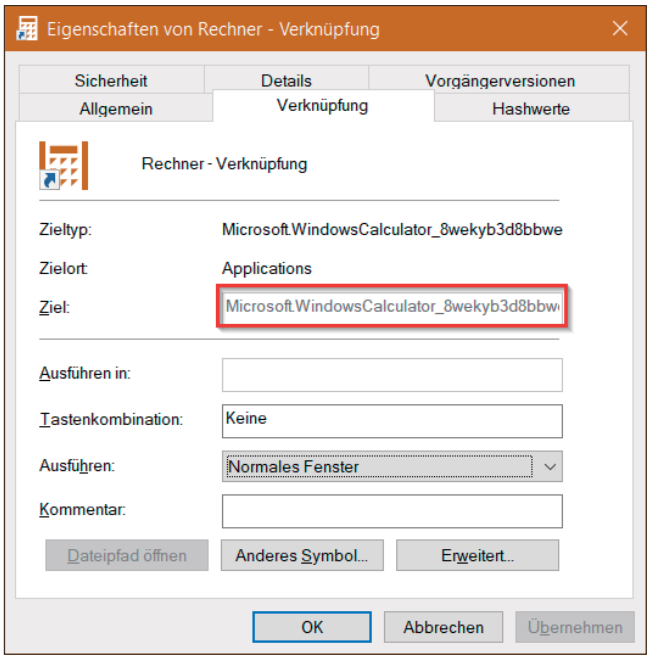
Von dort installieren Sie anschließend das Paket *Appium.WebDriver* [11]. In der Dokumentation wird diese Komponente gelegentlich auch *Appium dotnet Driver* genannt. Für die Installation von Vorabversionen muss ein Haken gesetzt werden, wie in Bild 2 zu sehen ist.

Listing 1: PowerShell: Ausgabe der AUMID

```
$installedApps = get-AppxPackage -allusers
$aumidList = @()

foreach ($app in $installedApps)
{
  foreach ($id in (Get-AppxPackageManifest $app)
    .package.applications.application.id)
  {
    $aumidList += $app.packagefamilyname + "!" + $id
  }
}
$aumidList | Sort-Object
```

Appium.WebDriver, per NuGet installiert (Bild 2)



Die AUMID der Rechner-App ist im Eigenschaftsdialog nicht vollständig sichtbar (Bild 3)

Zu testende Anwendung identifizieren

Um eine UWP-App programmatisch starten zu können, muss zunächst deren ID (Application User Model ID = AUMID) bekannt sein. Sie lässt sich zum Beispiel ermitteln, indem man eine Verknüpfung der App auf dem Desktop anlegt, dann per Rechtsklick das Kontextmenü startet und über *Eigenschaften | Verknüpfung* den Wert unter *Ziel* einsieht. Unter Umständen wird die ID an dieser Stelle allerdings abgeschnitten dargestellt (Bild 3). Alternativ können Sie das PowerShell-Skript aus Listing 1 mit Administratorrechten ausführen. Es listet die IDs aller installierten Apps, und die gesuchte ID lässt sich in der Regel am Namen erkennen. Bei allen Anwendungen, die keine UWP-Anwendungen sind, muss anstelle der AUMID der Pfad zur ausführbaren Datei angegeben werden.

Ansprechen von GUI-Elementen

Damit Sie in einer laufenden Anwendung ein GUI-Element ansprechen können, muss das Element eindeutig identifiziert werden. Dazu stellt WinAppDriver diverse Lokator-Methoden (Selektoren) zur Verfügung (Tabelle 2). Mithilfe des Windows-SDK-Tools Inspect [12], das nach der Installation von Visual Studio mittels *C:\Program Files (x86)\Windows Kits\10\bin\(Version)\(Plattform)\inspect.exe* gestartet werden ▶

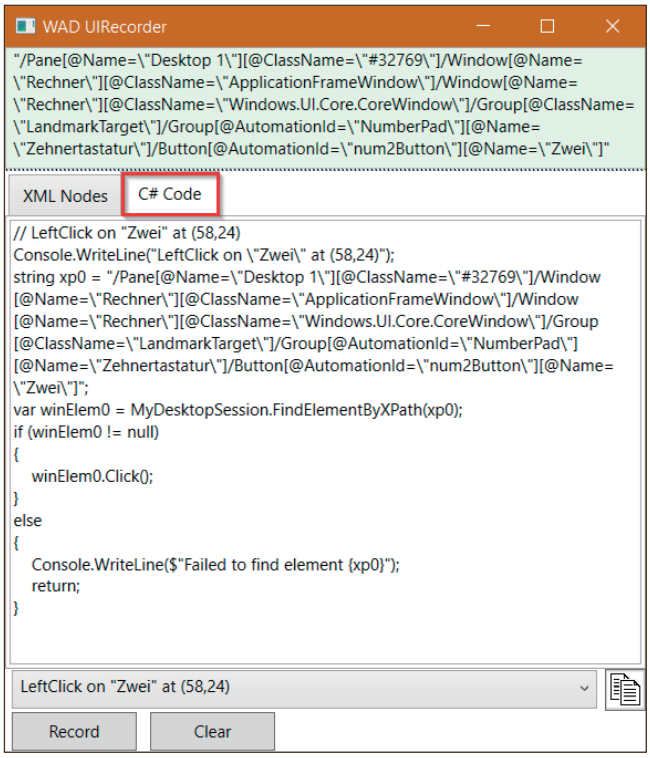
Tabelle 2: Unterstützte Lokator-Methoden

Methode	Attribut in Inspect	Beispiel
FindElement-ByAccessibilityId	AutomationId	plusButton
FindElement ByClassName	ClassName	Button
FindElementById	RuntimeId (dezimal)	42.656828.4.8
FindElement ByName	Name	Plus
FindElement-ByTagName	ControlType (Textfragment zwischen *UIA_*... und ... *ControlTypeId*). Referenz der möglichen Werte siehe [17].	Button
FindElement ByXPath	Alle	//Button[@ Name='Plus']

kann, lassen sich die konkreten Werte aus der Anwendungssitzung ermitteln. Dazu müssen Sie das gewünschte GUI-Element mit der Maus in den Fokus nehmen. Daraufhin werden die Attribute des Elements angezeigt. Tabelle 2 zeigt, welche Lokator-Methode welches Attribut erkennt.

Wenn das gesuchte GUI-Element ein *AutomationId*-Attribut besitzt, sollte dieses möglichst zur Identifikation genutzt werden, da es speziell für solche Automationszwecke vorgesehen ist [13] – siehe (Bild 4).

Mitunter gibt die Lokator-Methode allerdings gar kein Element zurück. So wird in der Rechner-App nach erfolgreicher Ausführung das Ergebnis *42* angezeigt; das Element mit der AutomationId *CalculatorResults* enthält aber (in der deutschen Version) den Text *Die Anzeige lautet 42*. Die untergeordneten Elemente mit der AutomationId *textContainer* beziehungsweise *normalOutput*, die das eigentliche Ergebnis *42* enthalten, werden nicht gefunden. In diesem Fall müssen Sie den Wert *42* aus den *CalculatorResults* herausparsen. Erwähnenswert ist noch der UIRecorder [14], ein Tool, das ähnlich wie Inspect arbeitet. Nachdem Sie ein GUI-Element der laufenden Anwendung in den Fokus genommen haben, werden in diesem Fall aber nicht die Attribute des Elements, sondern der Pfad zum Element im XPath-Format ausgegeben. Im Aufnahmemodus können Sie zudem C#-Code generieren, den Sie über die Registerkarte *C# Code* herauskopieren können (Bild 5). Starten Sie Inspect und UIRecorder aber nicht gleichzeitig – sie kommen sich in die Quere.

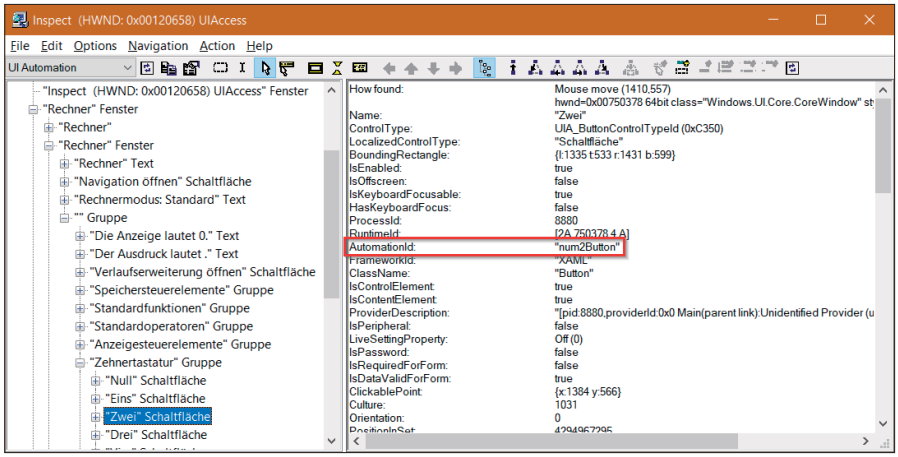


Der UIRecorder zeigt oben einen XPath-Wert und in der Registerkarte unten C#-Code an. Hier wurde die Schaltfläche 2 angeklickt (Bild 5)

Testdesign und -ausführung

In Listing 2 finden Sie alle Schritte eines typischen Tests. Zu diesen zählen im Einzelnen:

- die Angabe der AUMID oder des Pfades zur Anwendung,
- die Angabe des URLs zum Automationsserver WinAppDriver,
- das Erzeugen einer Automationssitzung,
- der Wechsel des obersten Anwendungsfensters (optional),
- die Angabe eines Timeouts (ebenfalls optional),
- die Bedienung der zu testenden Anwendung,
- das Auslesen des Rechenergebnisses aus der Anzeige,
- das Testen (Assert) auf das korrekte Ergebnis und
- schließlich noch das Vernichten der Automationssitzung. ▶



Inspect zeigt die Attribute der Rechner-App an. Der Fokus liegt auf Schaltfläche 2. Hervorgehoben ist die, die für die Automation genutzt werden sollte (Bild 4)

Listing 2: Test einer UWP-App

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenQA.Selenium.Appium;
using OpenQA.Selenium.Appium.Windows;

namespace GUITest
{
  [TestClass]
  public class WinGUITest {
    private const string appID =
      "Microsoft.WindowsCalculator_8wekyb3d8bbwe!App";

    // Alternatively, for non-UWP applications use
    // path to executable file:
    // private const string appID =
    //   @"C:\...\filename.exe";

    private const string winAppDriverUrl =
      "http://127.0.0.1:4723"; // Default IP and port

    protected static WindowsDriver<
      WindowsElement> session;

    [ClassInitialize]
    public static void ClassInitialize(
      TestContext context) {
      var options = new AppiumOptions();
      options.AddAdditionalCapability("app", appID);

      if (session == null) {
        // Create automation session
        session = new WindowsDriver<WindowsElement>(
          new Uri(winAppDriverUrl), options);

        // Alternatively, if Appium is used
        // instead of WinAppDriver:
        // options.AddAdditionalCapability(
        //   "deviceName", "WindowsPC");
        // session = new WindowsDriver<
        //   WindowsElement>(new Uri(
        //   winAppDriverUri + '/wd/hub'), options);

        Assert.IsNotNull(session);

        // If the application displays a
        // splash screen, wait until it disappears
        // and reassign the session to the correct
        // window:
        // System.Threading.Thread.Sleep(
        //   TimeSpan.FromSeconds(5));
        // session.SwitchTo().Window(session
        // .WindowHandles[0]);

        // Set timeout for element search
        session.Manage().Timeouts().ImplicitWait =
          TimeSpan.FromSeconds(3);
      }
    }

    [TestInitialize]
    public void TestInitialize() {
      // Clear the calculation display
      session.FindElementByAccessibilityId(
        "clearButton").Click();
    }

    [TestMethod]
    public void Test2688Div64() {
      // Do the calculation
      session.FindElementByAccessibilityId(
        "num2Button").Click();
      session.FindElementByAccessibilityId(
        "num6Button").Click();
      session.FindElementByAccessibilityId(
        "num8Button").Click();
      session.FindElementByAccessibilityId(
        "num8Button").Click();
      session.FindElementByAccessibilityId(
        "divideButton").Click();
      session.FindElementByAccessibilityId(
        "num6Button").Click();
      session.FindElementByAccessibilityId(
        "num4Button").Click();
      session.FindElementByAccessibilityId(
        "equalButton").Click();

      string result = new System.Text.
        RegularExpressions.Regex(@"\d+").Match(
        session.FindElementByAccessibilityId(
        "CalculatorResults").Text).Value;

      Assert.AreEqual("42", result);
    }

    [TestCleanup]
    public void TestCleanup()
    { }

    [ClassCleanup]
    public static void ClassCleanup() {
      if (session != null) {
        session.Close();
        session.Quit();
        session = null;
      }
    }
  }
}
```

Tabelle 3: Relevante Capabilities

Capability	Pflichtfeld	Beschreibung	Beispiel
app	Entweder app oder appTop-Level-Window	AUMID, Pfad zur ausführ-baren Datei	Microsoft.Windows Calculator _8wekyb3 d8bbwe!App C:\Windows\ System32\notepad.exe
appTopLevel-Window	Entweder app oder appTopLe-velWindow	Handle des obersten Fensters einer laufenden Anwendung	B822E2
app-Arguments	Nein	Startargu-mente	MyFile.txt
app-WorkingDir	Nein	Arbeitsver-zeichnis einer Anwendung	C:\MyFolder\
platform-Name	Nein	Name der Zielplattform	Windows
platform-Version	Nein	Version der Zielplattform	10
deviceName (nur bei Ausführung über Appium)	Ja	Gerätename	WindowsPC

Zum Erzeugen der Automationssitzung müssen sogenannte Capabilities angegeben werden. Es handelt sich dabei schlicht um Konfigurationsparameter. Eine Liste der für WinAppDriver nutzbaren Capabilities finden Sie in Tabelle 3.

Beachten Sie, dass die Automationssitzung bei Nutzung von Appium anstelle von WinAppDriver etwas anders erzeugt wird. Sie müssen dann zusätzlich den *deviceName WindowsPC* angeben und außerdem die Pfadangabe */wd/hub* an den URL anhängen.

Nützlich ist, dass Sie die Automationssitzung auch für eine laufende Anwendung erzeugen können. Anstelle der AUMID geben Sie dann das Handle des obersten Fensters der Anwendung im Hexadezimalformat an. Das Handle können Sie dynamisch über eine Desktop-Sitzung ermitteln, vergleiche Listing 3.

Wenn der Test fertiggestellt ist, starten Sie WinAppDriver (*C:\Program Files (x86)\Windows Application Driver\WinAppDriver.exe*) mit Administratorrechten. Es öffnet sich ein Konsolenfenster, in dem der Server seine Bereitschaft signalisiert und HTTP-Aktivitäten protokolliert. Stoßen Sie den Test über Visual Studio an und schauen Sie zu, wie die Rechner-App von WinAppDriver bedient wird. Die Ausgabe im Konsolenfenster zeigt Bild 6. Am Ende sollte der Test einen Erfolg melden.

Falls Sie die anderen Beispiele aus Tabelle 1 selbst nachvollziehen, werden Sie bemerken, dass der Test in Beispiel 4 fehlschlägt. Das ist korrekt, denn dieser Rechner – ein unprofessionelles Hobbyprojekt – rechnet falsch.

Weiterführende Aspekte

Neben der Methode *WindowsElement.Click*, die im Beispiel mehrfach benutzt wird, ist vor allem die Methode *WindowsElement.SendKeys* wichtig. Mit ihr können Sie Text in GUI-Elemente schreiben.

In der Dokumentation und in den offiziellen Beispielen zum Automationsserver WinAppDriver wird oft noch *RemoteWebDriver.Keyboard.SendKeys* benutzt – dieses Vorgehen wird

Listing 3: Erzeugen einer Automationssitzung

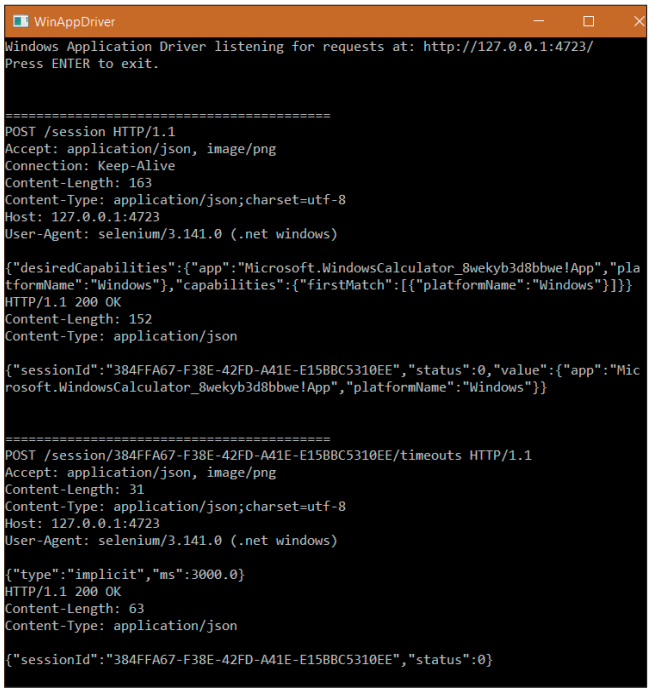
```
[ClassInitialize]
public static void ClassInitialize(
  TestContext context)
{
  var options = new AppiumOptions();
  string handle = GetWindowHandleInHex("Rechner");
  options.AddAdditionalCapability(
    "appTopLevelWindow", handle);

  if (session == null) {
    ...
  }
}

private static string GetWindowHandleInHex(
  string applicationName)
{
  var optionsRoot = new AppiumOptions();
  optionsRoot.AddAdditionalCapability("app", "Root");
  var desktopSession =
    new WindowsDriver<WindowsElement>(
    new Uri(winAppDriverUrl), optionsRoot);

  string handle = desktopSession.FindElementByName(
    applicationName).GetAttribute(
    "NativeWindowHandle");
  // Convert to hex
  handle = (int.Parse(handle)).ToString("x");

  if (desktopSession != null) {
    desktopSession.Close();
    desktopSession.Quit();
    desktopSession = null;
  }
  return handle;
}
```



Ausschnitt der Protokollierung der WinAppDriver-Aktivitäten (Bild 6)

allerdings nicht mehr empfohlen, da die Keyboard-Klasse abgekündigt (deprecated) ist.

Ein fortgeschrittenes Beispiel ist unter [15] verfügbar. Dort wird gezeigt, wie man Mausbewegungen automatisiert.

Ein typisches Problem für die Automatisierung sind Splashscreens, also Fenster, die beim Anwendungsstart kurz angezeigt und dann geschlossen werden. In diesem Fall müssen Sie nach Erzeugen der Sitzung, wie in Listing 2 gezeigt, eine Warteschleife einbauen und anschließend das gewünschte Fenster mit der *SwitchTo*-Methode neu zuweisen.

Schließlich sollten Sie darauf achten, bei komplexeren GUIs die Tests nach dem Page-Object-Muster zu erstellen [16]. Dies trennt den eigentlichen Test von den technischen GUI-Details und entspricht gutem objektorientiertem Design.

Fazit

Microsoft stellt mit WinAppDriver ein kostenloses und quelloffenes Automations-Tool zur Verfügung, das mit neuen wie alten Windows-Technologien funktioniert und auch noch mit dem verbreiteten WebDriver-Protokoll beziehungsweise Appium kompatibel ist. Das ist insbesondere in heterogenen Softwareumgebungen eine sehr interessante Lösung. Wer bisher auf Coded-UI-Tests gesetzt hat, muss sich allerdings umorientieren.

Da WinAppDriver noch recht jung ist, stellt es (noch) nicht den kompletten Funktionsumfang zur Verfügung, den andere Tools bieten. Zum Beispiel wäre ein ausgereifteres Recording-Tool zur Aufnahme von GUI-Interaktionen hilfreich. Es kommt gelegentlich auch noch zu Kompatibilitätsproblemen mit Appium – offenbar funktioniert die Abstimmung zwischen der Appium-Gemeinde und den Entwicklern bei Microsoft nicht immer.

Wie nützlich WinAppDriver im Testalltag sein wird, muss sich zeigen. Einfache Tests laufen erfahrungsgemäß stabil. Für komplexere Testszenarien, beispielsweise bei dynamisch generierten GUI-Elementen, beim Scrollen von Fensterinhalten, bei Anwendungen mit mehreren Fenstern oder beim Ansprechen von Touchscreens, fehlen noch Erfahrungswerte. Insbesondere muss die Technologie auch bei den regelmäßig stattfindenden Betriebssystem-Updates stabil bleiben. Da WinAppDriver jedoch mit der hauseigenen Microsoft UI Automation arbeitet, hat der Softwareriese aus Redmond zumindest beste Voraussetzungen, eine integrierbare und robuste Lösung für Automations- und Testaufgaben unter Windows zu schaffen. ■

[1] Selenium WebDriver, www.dotnetpro.de/SL1906WinAppDriver1
[2] Martin Gossen, Test-Automation mit Selenium WebDriver, Der große Formular-Check, dotnetpro 5/2013, Seite 34 ff., www.dotnetpro.de/A1305Selenium
[3] Georg Hansbauer, Qualitätssicherung bei der Website-Entwicklung, Auf gut Test, dotnetpro 7/2018, Seite 64 ff., www.dotnetpro.de/A1807WebsiteTesting
[4] Appium, http://appium.io
[5] Appium Quellcode, https://github.com/appium
[6] WebDriver-Protokoll, www.w3.org/TR/webdriver
[7] Windows Application Driver, www.dotnetpro.de/SL1906WinAppDriver2
[8] Appium Windows Driver, www.dotnetpro.de/SL1906WinAppDriver3
[9] Appium Windows Driver Quellcode, www.dotnetpro.de/SL1906WinAppDriver4
[10] Windows Application Driver Download, www.dotnetpro.de/SL1906WinAppDriver5
[11] Appium.WebDriver, www.dotnetpro.de/SL1906WinAppDriver6
[12] Inspect, www.dotnetpro.de/SL1906WinAppDriver7
[13] Use the AutomationID Property, www.dotnetpro.de/SL1906WinAppDriver8
[14] WinAppDriver UI Recorder Tool, www.dotnetpro.de/SL1906WinAppDriver9
[15] Windows Application Driver Beispiele in C#, www.dotnetpro.de/SL1906WinAppDriver10
[16] WinAppDriver Page Objects, www.dotnetpro.de/SL1906WinAppDriver11
[17] UI Automation Control Types, www.dotnetpro.de/SL1906WinAppDriver12

Martin Gossen
ist IT-Berater bei der IKS GmbH in Hilden. Er erstellt seit 20 Jahren Softwarelösungen vorwiegend mit C#, .NET und Microsoft SQL Server. Sie erreichen Ihn unter **m.gossen@iks-gmbh.com**.

dnpCode A1906WinAppDriver