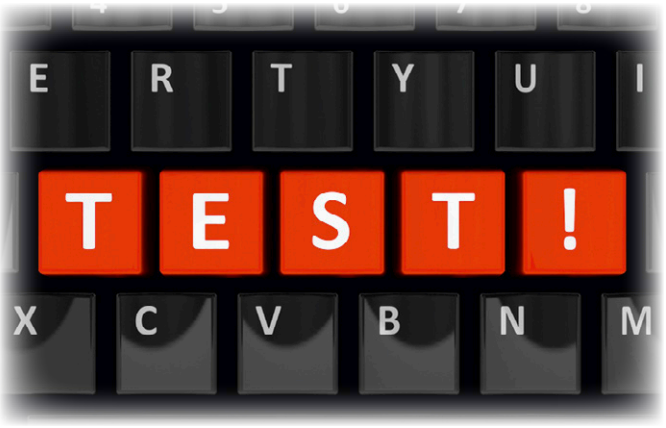
Test-IT mit testIT

Effiziente Abnahmetests im Rechenzentrum

Pascal Moll

Was meinen Sie? Wie lassen sich in einem der größten deutschen kommunalen Rechenzentren Deployments in mehr als hundert Kundenprozessen effizient und ressourcenschonend testen? Um permanent die Funktionalität neuer Features und Korrekturen in den Abläufen des Rechenzentrums gewährleisten zu können, gibt es ein nachhaltiges Rezept: automatisierte Tests. Dieser Artikel beschreibt die Konzeption, Implementierung und Einführung von automatisierten Abnahmetests auf Basis des testIT-WebTester-Frameworks der NovaTec Consulting GmbH. Abschließend werden noch praktische Tipps für eine nachhaltige Testqualität im agilen Umfeld gegeben.



November 2013: Ausgangslage

Wie geplant wurde die regelmäßige Wartung der Softwarelandschaft durch das Deployment eines neuen Release abgeschlossen. Etwa einen Monat lang haben dabei mehr als zehn Anwender das geänderte System erst einmal auf Herz und Nieren getestet, manuell versteht sich. Danach wurde das Produkt für die Kunden allgemein freigegeben. Zu diesem Zeitpunkt war aber der nächste Patch, mit allgemeinen Verbesserungen und Fehlerbehebungen, schon in Vorbereitung.

Zum Verständnis: Pro Quartal ist für das Rechenzentrum immer ein Release geplant, in dem neue Features eingeführt werden können, und alle zwei Wochen sind Patches zur allgemeinen Fehlerhebung und Usability-Verbesserung vorgesehen.

Das klingt nach viel Aufwand, um die Qualität hochzuhalten, oder? Das ist natürlich auch so. Genau das hatte auch der Kunde erkannt, weswegen er ein effektiveres, vor allem ein schnelleres und kostengünstigeres Testverfahren wollte.

Januar 2014: Analysephase

Nach dem Jahreswechsel begann die Analyse der Systemlandschaft. In ausführlichen Gesprächen mit dem Kunden wurden verschiedene Optionen vorgestellt. Statt beispielsweise das Testteam zu vergrößern, fiel die Wahl auf einen besseren Weg: Testautomatisierung.

Dadurch sollten nicht nur Zeit und Geld eingespart werden, ganz besonders stand die Möglichkeit einer Fehlerminimierung im Fokus. Indem der Testablauf durch eine Maschine ausgeführt wird, ist jeder Testfall reproduzierbar und läuft somit gleich ab. Mit diesem Vorgehen lassen sich menschliche Fehler ausschließen, die Reproduzierbarkeit ist bei Fehlerfällen sehr leicht gewährleistet und Fachkräfte können außerdem für andere Tätigkeiten eingesetzt werden. Der Testprozess ist zudem zeitlich und personell unabhängig ausführbar.

Da bei diesem Projekt keinerlei Schnittstellen nach außen verwendbar waren und es sich bei dem gesetzten Ziel auch um Abnahmetests handelte, blieb nur der Test über die Weboberfläche. Dies brachte zugleich noch die Simulation des Endanwenders als weiteren Vorteil mit sich.

Zur Umsetzung hatte sich aus Erfahrung der Einsatz des Selenium Driver bewährt. Allerdings führt die naive Verwendung von Selenium (s. Listing 1) zu nicht nachhaltigem Code.

```
public void fillContactFormular () {
  webDriver.findElement(By.if("firstname")).sendKeys("Max");
  webDriver.findElement(By.if("lastname")).sendKeys("Mustermann");
  webDriver.findElement(By.if("phone")).sendKeys("0800 1234567");
  webDriver.findElement(By.if("fax")).sendKeys("0800 1234567");
  webDriver.findElement(By.if("company")).sendKeys("Mustermann Ltd.");
  webDriver.findElement(By.if("ok")).click();
}
```

Listing 1: Standard-Selenium-Code

Listing 1 demonstriert das Ausfüllen eines Kontaktformulars. Bereits bei diesem simplen Beispiel fällt auf, dass der Code eine starke Tendenz zur Unübersichtlichkeit hat. Nach dem Abarbeiten der letzten Anweisung wird eine neue Seite geöffnet, dies spiegelt sich im Code jedoch nicht wieder. Je komplexer der Testfall wird, desto unübersichtlicher entwickelt sich der Code.

Hier fällt eine der markantesten Schwachstellen dieses Vorgehens auf: Die Wiederverwendbarkeit des Quellcodes ist nicht gegeben, da das erneute Auftreten desselben Programmablaufs mit denselben Daten extrem unwahrscheinlich ist.

Ein weiteres Problem ist, dass nicht erkennbar ist, mit welchen Arten von Objekten die entsprechenden Anweisungen interagieren, da Selenium Driver alles als generische Klasse `WebElement` behandelt. So wird blind die `sendKeys`-Methode aufgerufen. Genauso gut könnte dies versehentlich mit einem Button geschehen. Die Folge: `sendKeys` wird zwar ausgeführt, da der Button aber kein Eingabeelement ist, reagiert er nicht wie erwartet, was im Test zu unerwarteten Folgen führen kann.

Zur Behebung der benannten Problemstellungen ist eine Selenium-Erweiterung mit einer typsicheren Programmierschnittstelle und dem *Page Object Pattern* die geeignete Lösung. Hierdurch entsteht strukturierter Code, Entwicklungsfehler können vermieden werden und gleichzeitig wird die Wiederverwendbarkeit gefördert.

Page Object Pattern

Mit dem Page Object Pattern wird die fachliche Logik der zu testenden Anwendung in Klassen modelliert. Jede Klasse repräsentiert dabei einen Aspekt der Anwendung (z. B. eine Dialogmaske). Die Interaktion mit der Benutzeroberfläche geschieht damit ausschließlich durch fachliche Methoden in diesen Klassen. Dies sorgt für Übersichtlichkeit und vermeidet Redundanzen im Testcode. Außerdem werden Erweiterungen des Quellcodes und eventuelle Änderungen erleichtert.

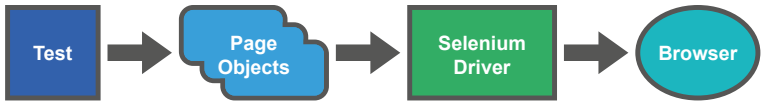


Abb. 1: Erklärung des Page Object Pattern

Jede Art von Kommunikation zwischen dem Page Object und der Webseite findet über Selenium Driver statt.

Diese Klassen bieten komfortable Methoden zur Elementinteraktion, wie zum Beispiel das Ändern eines Textfeldinhaltes mit einer Operation, statt wie bisher bei Selenium mit zweien.

Neben dem Anbieten von zusätzlichen Methoden werden nicht sinnvoll anwendbare Funktionen vor dem Anwender versteckt. Als Beispiel dient hier das Senden von Keys an eine Checkbox.

Um Abweichungen zwischen Tests und Oberfläche frühzeitig zu erkennen, überprüft jede Klasse zur Laufzeit automatisch, ob sie mit kompatiblen Elementen interagiert. Hierdurch ist die Anforderung nach Typsicherheit erfüllt.

Für eine optimale Erweiterbarkeit können anwendungsspezifische Fachklassen sehr leicht implementiert und in das Framework integriert werden. Als Beispiel seien hier Spezialelemente von UI-Frameworks wie Vaadin oder JQuery genannt.

Neben der Automatisierung von Tests wurde vom Kunden auch eine aussagekräftige Protokollierung gewünscht. Noch besser als ein simples Logging-System wäre ein Event-System. Dieses würde nicht nur das Logging von bestimmten Ereignissen ermöglichen, wie zum Beispiel Klicken eines Buttons oder Befüllen eines Eingabeelements, sondern dem Anwender die Möglichkeit geben, gezielt auf bestimmte Ereignisse (zum Beispiel Exceptions) zu reagieren. Somit wäre sogar im Fehlerfall noch eine Interaktion mit Maskenelementen möglich.

Anforderungsspezifikation an ein Testframework

Um all diese Anforderungen effizient erfüllen zu können, bedarf es eines Frameworks mit folgenden Eigenschaften:

- typsichere Programmierschnittstelle,
- Event-System für Protokollierung und Eventhandler,
- Nutzung des Page Object Pattern.

Wir hatten bereits mehrfach ähnliche, projektspezifische Frameworks implementiert und der Verdacht lag nahe, dass weitere Projekte dieser Art folgen würden. Aus diesem Grund fiel die Entscheidung, ein möglichst universelles Framework zu entwickeln und öffentlich zur Verfügung zu stellen. Dieses erhielt den Namen *WebTester*.

Februar 2014. Den ersten Schritt der Automatisierung stellte das Festhalten der zu testenden Szenarien dar. Hierfür wurden die bisher manuell ausgeführten Tests in Form von Screenshots dokumentiert. Die so entstandenen Storyboards ließen sich anschließend sehr einfach in automatisierte Tests überführen.

Als Basis für WebTester dient Selenium Driver, da er, wie bereits erwähnt, einer der besten Java basierten Driver auf dem Markt ist. Um den vollen Funktionsumfang von Selenium weiterhin gewährleisten zu können, war es wichtig, dass jederzeit auf das Selenium-API durchgegriffen werden kann.

Um Seleniums sehr technische `WebElement`-Klasse auf ein fachliches Niveau anzuheben, wurden die gängigsten Web-UI-Elemente durch Java-Klassen repräsentiert. Dies bietet den Vorteil, unterschiedliche Ausprägungen von gleichartigen Elementen an dieselbe Java-Klasse zu binden. Ein Beispiel hierfür ist Button, welcher sowohl das HTML-Element `<Input type='Button'>` als auch `<Button>` repräsentiert.

WebTester-Codebeispiel mit Erklärung

Listing 2 beschreibt ein PageObject, welches die Login-Seite einer zu testenden Anwendung abbildet. Der Login findet durch Eingabe von Benutzername und Passwort statt. Nachfolgend soll der Code näher erläutert werden:

- *(1)* Die Annotation `@IdentifyUsing` bestimmt, wie das Textfield zur Laufzeit identifiziert wird. Standardmäßig wird hier die ID des HTML-Elements verwendet. Auch andere Zugriffsmethoden wie Xpath, CSS-Selektor oder Name sind möglich. Um die Identifikationsinformationen zu ermitteln, können in der Regel die Developer-Tools des Browsers verwendet werden.
- *(2)* In der Zeile darunter wird das Textfield, welches durch (1) beschrieben wird, in die Variable `usernameField` injiziert. Die zuvor beschriebene typsichere Programmierschnittstelle stellt beim ersten Zugriff sicher, dass das injizierte Feld vom korrekten Typ ist.
- *(3)* Die Annotation `@AfterInitialization` sorgt dafür, dass die annotierte Methode ausgeführt wird, nachdem das `PageObject` initialisiert wurde. Diese Methoden werden generell dazu genutzt, um sicherzustellen, dass der Test sich auf der korrekten Seite befindet. Im konkreten Fall wird überprüft, ob der Titel der Webseite „TestApp: Login“ entspricht.
- *(4)* Die Methode `loginWithValidUser` ruft einen sogenannten Workflow auf. Workflows dienen generell dazu, Abläufe zu überspringen, welche für einen Test nicht relevant sind. Im konkreten Fall wird ein Login durchgeführt, ohne dass dem Test Benutzername oder Passwort bekannt sein muss.
- *(5)* Die Methode `setUsername` führt eine sogenannte Aktion auf dem Textfeld Benutzername durch. Aktionen sind grundsätzlich Operationen, welche den Inhalt einer Seite manipulieren, ohne sie zu verlassen. Im konkreten Beispiel wird der übergebene Benutzername in das entsprechende Textfeld eingetragen.

```
public class LoginPage extends PageObject {
  @IdentifyUsing ( "username" )                    // (1)
  TextField usernameField;                         // (2)

  @IdentifyUsing ( "password" )
  PasswordField passwordField;

  @IdentifyUsing ( "login" )                                    ➔
```

```
    Button loginButton;

    @AfterInitialization
    void assertThatCorrectPageIsDisplayed () {       // (3)
      asserThat(getBrowser().getPageTitle(), is("TestApp: Login:"));
    }
    /* workflows */
    public WelcomePage loginWithValidUser () {       // (4)
      return login("username", "123456");
    }
    public WelcomePage login (String username, String password) {
      return setUsername(username).setPassword(password).clickLogin();
    }
    /* actions */
    public LoginPage setUsername (String username) { // (5)
      this.usernameFiled.setText(username);
      return this;
    }
    ...
  }
```

Listing 2: Beispiel-WebTester-PageObject mit Erklärung

Erhöhung von Nachhaltigkeit und Reproduzierbarkeit

In der Praxis haben sich sowohl Screenshots im Fehlerfall als auch das Event-System bewährt. Trotzdem traten Situationen auf, in denen diese Form der Protokollierung für das Nachvollziehen eines Fehlers nicht ausreichend war. Um aussagekräftige Fehlertickets zu stellen, reichte eine Beschreibung des Vorganges zusammen mit den Protokollen und Screenshots manchmal nicht aus.

Eine einfache Möglichkeit, dieses Problem zu lösen, stellt der Einsatz eines Bildschirmrecorders dar. Ein Video zeigt auch Ereignisse, welche eventuell auf einem Screenshot oder in einem Protokoll verborgen bleiben.

Es wurde eine bestehende Open-Source-Bibliothek genutzt, um einen Bildschirmrecorder zu implementieren, welcher sich leicht in den Lebenszyklus von Tests einbinden lässt. Somit stehen folgende Dokumentationsmöglichkeiten zur Verfügung:

- Protokoll des Event-Systems,
- Screenshots im Fehlerfall,
- Video des Testfalls.

November 2014: Einführung und erste Erfahrungen

Die meisten Bereiche der Anwendung waren durch Tests abgedeckt. Der Kunde konnte auf Knopfdruck die zu Verfügung stehenden Tests ausführen und sich im Anschluss eine Auswertung über das Ergebnis anzeigen lassen.

In regelmäßigen Abständen wurden neue Tests unmittelbar nach Einspielung eines Release in die Testumgebung umgesetzt.

Durch die Umstellung von manuellen auf automatisierte Tests hat sich die Gesamtdauer des Abnahmeprozesses drastisch reduziert. Die Mitarbeiter des Rechenzentrums wenden nun nur noch einen Bruchteil der bisherigen Zeit für die Durchführung von Tests auf und haben somit mehr Zeit für die Auswertung der Ergebnisse. Nach Aussage des Kunden beträgt die Zeitersparnis mindestens 30 bis 40 Prozent, was einen beträchtlichen wirtschaftlichen Gewinn bedeutet.

Fazit

Der Kunde war mit dem Ergebnis des Projekts sehr zufrieden und hat entschieden, die positiven Effekte des automatisierten Testens auf die SAP-Systeme auszuweiten. Daraus folgte eine weitere Beauftragung.

Das WebTester-Framework ist seither öffentlich verfügbar und bei anderen Kunden im Einsatz. Noch in diesem Jahr wird es als Open-Source-Projekt veröffentlicht werden.

Für weitere Informationen wird auf den Dokumentationsbereich unter `https://documentation.novatec-gmbh.de/x/v4E6AQ` verwiesen.

Pascal Moll arbeitet als Consultant bei NovaTec Consulting GmbH in Frankfurt am Main (Twitter: @NT_AQE). Er befasst sich seit mehr als zehn Jahren mit Anwendungsentwicklung. Die Beratungsschwerpunkte von Pascal Moll liegen in der Java-Entwicklung, Testautomatisierung und Webentwicklung.
E-Mail: pascal.moll@novatec-gmbh.de