Sonderdruck für b+m Informatik

Ausgabe 02|2018

Deutschland € 9,90 Österreich € 10,90 Schweiz sfr 18,20

4 194164 109900 02

OBJEKTspektrum

IT-Management und Software-Engineering

www.OBJEKTspektrum.de

Modernes Testen

Organisation und Werkzeuge

Interview mit Stella Schiffczyk, Augenzeugin im NSA-Untersuchungsausschuss,

über BND, Handyortung und die Rolle der Politik

- Software-Engineering für robuste, wartbare Oberflächentests
- KI und Automatisierung im agilen Projektmanagement
- Schnellere und bessere Software mit DevOps?

Modernes Testen – Organisation und Werkzeuge

G 6540F

UI-Regressionstests

Software-Engineering für robuste, wartbare Oberflächentests

Seit Jahren werden Oberflächentests in der Praxis von stets wiederkehrenden Problemen begleitet: Aufgrund mangelnder Abstraktionen sind sie schwer wartbar und technisch fragil. Da klassische Capture-Replay-Ansätze dieses Problem besonders tief in sich tragen, konzentriert sich dieser Artikel auf API-basierte Verfahren und stellt moderne Software-Engineering-Konzepte und -Patterns vor, welche die Domäne „UI-Test" ins Zentrum stellen und dadurch die Wartbarkeit signifikant verbessern. Ein weiteres Augenmerk legt dieser Artikel auf die Integration in etablierte Werkzeugketten.

Mit diesem Artikel widmen wir uns dem Thema der integrativen Oberflächen-Regressionstests. Diese dienen der Ergänzung von Unittests, welche mittlerweile durch eine reichhaltige Werkzeugpalette unterstützt werden, aber naturgemäß nicht-integrativ testen und auch unterhalb des sichtbaren Teils der User Interfaces (UI) ansetzen. Oberflächentests führen hingegen längere fachliche Szenarien unter Berücksichtigung von Screenflow und Persistenz durch. Natürlich ist auch auf dieser Ebene Testautomation das Ziel.

Es existieren zahlreiche Werkzeuge, die sich dieser Aufgabe annehmen. Weit verbreitet ist dabei das Vorgehen, Skripte mithilfe eines Automations-API (z. B. Selenium WebDriver API für Webanwendungen) zu entwickeln. Im Gegensatz zu Capture-Replay-Verfahren, die Aktionen aufzeichnen und später abspielen, ist es durch die Orchestrierung solcher Skripte möglich, Redundanzen zu vermeiden und die Wiederverwendbarkeit und Wartbarkeit zu erhöhen.

Dennoch sehen wir auch bei diesem Ansatz gravierende Probleme: Die Skriptsammlung wird mit der Zeit schwer wartbar, insbesondere dann, wenn keine Technologie-Abstraktionen eingebaut werden, sondern der Skriptcode direkt auf der HTML-Ebene aufsetzt. Funktionale und vor allem technische Evolution, beispielsweise der Austausch des UI-Frameworks im System under Test (SUT), erzeugt daher massiven Aufwand bei der Skriptpflege.

Dadurch, dass dem Testentwickler nur Skripte zur Verfügung gestellt werden, benötigt er zwar kaum Programmierkenntnisse, ist aber auch sehr eingeschränkt in seiner Handlungsfähigkeit und ist bei fehlenden Funktionalitäten auf die Erstellung der Skripte durch Entwickler angewiesen. Es existiert also eine Lücke zwischen

- der Freiheit des Testentwicklers, die Testfälle unabhängig von Dritten formulieren zu können, und
- der Möglichkeit, die Tests frei von technischen Aspekten zu halten.

Obwohl diese Schwierigkeiten immer wieder auftreten und häufig diskutiert werden, gibt es dafür leider keine Lösung „von der Stange". Wir haben uns jedoch nicht geschlagen gegeben und konnten feststellen, dass man mit modernen Software-Engineering-Ansätzen diese Lücke weitestgehend schließen kann. Dieses Wissen möchten wir mit diesem Artikel teilen und zur Diskussion stellen.

Lösungsansatz

Im Kern geht es – wie so oft in der Informatik – um geeignete Abstraktionen und Modularisierung. Wir (die Autoren) verfügen über langjährige Expertise im Bereich DSL-Engineering (DSL = Domain Specific Language) und dem entsprechenden Tooling.

Bislang setzten wir dieses Wissen vornehmlich in der Applikationsentwicklung ein. Es stellte sich jedoch schnell heraus,

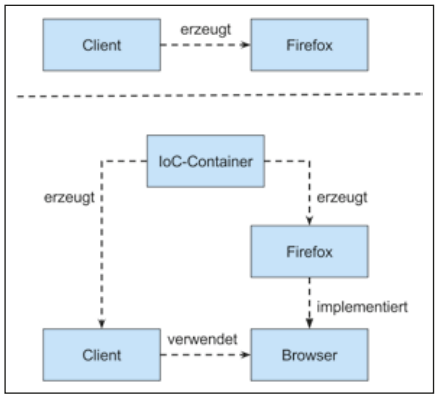


Abb. 1: Erläuterung von Inversion of Control

Inversion of Control (IoC)

Inversion of Control bezeichnet ein Umsetzungsparadigma. Die grundsätzliche Idee ist, dass ein Client nicht selbst den Kontrollfluss steuert, sondern lediglich definierte Schnittstellen verwendet.

Das folgende Beispiel (**siehe Abbildung 1**) illustriert dies anhand eines Web-Browsers. Der obere Teil zeigt, wie der Kontrollfluss ohne IoC aussieht: Der Client erzeugt den Firefox und verwendet ihn.

Zentrales Konzept bei IoC ist ein Container, der die Instanzen erzeugt und verwaltet. Der Client instanziiert Firefox nicht selbst, sondern gibt nur an, dass er einen Browser verwenden möchte. Die Konfiguration des Containers legt nun fest, welche Instanz injiziert (Dependency Injection) wird.

Durch diesen Mechanismus steuert der Container, welcher Browser verwendet wird. Der Client muss nicht angepasst werden, um zum Beispiel von Firefox zu Chrome zu wechseln.

Kasten 1

dass die DSL-Engineering-Perspektive auf die Domäne UI-Testing neue, zielführende Impulse liefert, um unter anderem das oben erwähnte Wartbarkeitsproblem anzugehen.
Besonders gute Ergebnisse ergaben sich dann in Kombination mit zwar bekannten, aber in der Praxis nicht immer konsequent eingesetzten Mustern wie dem *PageObject*-Pattern, auf das wir später genauer eingehen.
Außerdem wollten wir existierende, gut funktionierende Räder, wie eben Selenium zur Browser-Automation oder Spring Boot als IoC-Container (**siehe Kasten 1**), nicht neu erfinden, sondern uns diese zunutze machen und darauf aufbauen.
Softwaretechnisch haben wir unsere Ansätze in einem neuen Framework mit dem Namen tapir (*Test API* against *R*egression) gebündelt, welches wir über Maven Central publizieren (vgl. [Tapir]). Dieses Framework verwenden wir im Folgenden als konkretes Beispiel für die dargestellten Konzepte. Die ausführbaren Codebeispiele sind bei GitHub frei verfügbar (vgl. [ShowCase]).

Testentwicklersicht: Domäne in den Fokus stellen

Wir nähern uns nun den Konzepten an, indem wir zunächst die Rolle des Testentwicklers einnehmen. In unserem Beispiel geht es darum, die Vorschläge zur Vervollständigung des Suchbegriffs, die die Google-Suche dem Benutzer unterbreitet, auszugeben (**siehe Abbildung 2**).
Es sind also die folgenden Schritte durchzuführen:

- Google-Webseite öffnen,
- Suchbegriff eingeben und
- vorgeschlagene Begriffe ausgeben.

Die in **Listing 1** gezeigte Lösung stammt aus dem Selenium *Getting Started Guide* (vgl. [Sel]).
An dem Code lassen sich die Probleme der fehlenden Abstraktion schnell erkennen:

- Der FirefoxDriver wird direkt instanziiert. Ein anderer Browser kann nur durch Änderung des Codes genutzt werden.
- Abhängigkeit vom HTML-Code: Sobald sich einer der Identifier im HTML-Code ändert, müssen alle Tests angepasst werden, die mit dem geänderten Oberflächenelement interagieren.
- Fachlicher Code und technische Aspekte sind miteinander vermischt, dies umfasst erstens die Notwendigkeit, den WebDriver wieder explizit zu schließen (driver.quit()), und zweitens eine while-Schleife, um auf die Vorschlagsliste zu warten (Synchronisation).

```
public class GoogleSuggest {
  public static void main( String[] args )
      throws Exception {
    WebDriver driver = new FirefoxDriver( );
    driver.get(
     "http://www.google.com/webhp?complete=1&hl=en" );
    WebElement query = driver.findElement( By.name( "q" ) );
    query.sendKeys( "Cheese" );
    long end = System.currentTimeMillis( ) + 5000;
    while ( System.currentTimeMillis( ) < end ) {
      ArrayList<WebElement> resultsDiv =
      ( ArrayList<WebElement> ) driver.findElements(
        By.className( "sbsb_a" ) );
      if ( resultsDiv.size( ) > 0 ) {
        break;
      }
    }
    List<WebElement> allSuggestions =
      driver.findElements(
        By.xpath( "//div[@class='sbqs_c']" ) );
    for ( WebElement suggestion : allSuggestions ) {
     System.out.println( suggestion.getText( ) );
    }
    driver.quit( );
  }
}
```

Listing 1: Lösung aus Selenium Getting Started Guide

Abb. 2: Screenshot der Google-Vorschläge

All dies führt dazu, dass der Leser von der eigentlichen Testspezifikation abgelenkt wird und diese nur schwer extrahieren kann. In dem *Getting Started Guide* wurde der Code noch um insgesamt sieben Kommentare angereichert, die wir in Listing 1 entfernt haben. Dass diese Kommentare überhaupt zum Verständnis benötigt werden, ist ein weiteres Indiz für mangelnde Lesbarkeit.
Fairerweise muss man berücksichtigen, dass es sich bei Selenium WebDriver nicht um ein Test-API, sondern ein Browserautomation-API handelt, das absichtlich auf einem möglichst niedrigen Abstraktionsniveau operiert und somit kein Testdomänen-Wissen hat. Genau aus diesem Grund ist es elementar, nicht gegen dieses generische API seine Tests zu entwickeln, sondern gegen ein spezifisches Test-API.
Um uns diesem Ansatz anzunähern, ist es sinnvoll, die Domäne, wie im oberen Teil von **Abbildung 3** illustriert, zunächst zu definieren. Der fachliche Test benutzt dabei Pages, die sich aus mehreren UI-Elementen zusammensetzen. Jedes UI-Element wird durch eine UI-Komponente repräsentiert. In Anlehnung an UML-Stereotypen mappen wir diese Domäne im unteren Teil der Abbildung jetzt auf unser Google-Beispiel.
Sich über die Domäne im Klaren zu sein, ist eine fundamentale Voraussetzung zur

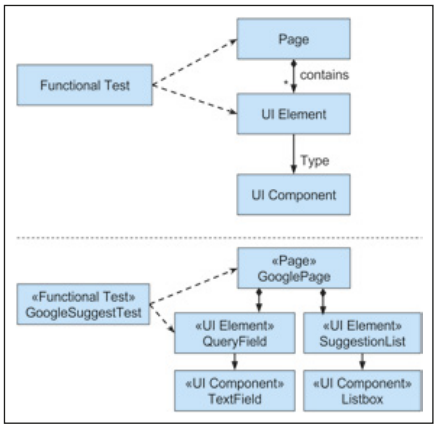


Abb. 3: Mapping des Google-Beispiels auf die Domäne UI-Test

```
@Page
class GooglePage {
  @SeleniumElement(name="q")
  TextField queryField

  @SeleniumElement(className="sbsb_a")
  Listbox suggestionList
}
```

Listing 2: PageObject für die Google-Suchseite

```
@UnitTest
@UseExtension(BrowserInteractionService)
class GoogleSuggest {
  @Autowired
  GooglePage googlePage

  @Test
  def void printSuggestions() {
    openURL("http://www.google.com/webhp?complete=1&hl=en")
    googlePage.queryField.text = "Cheese"
    googlePage.suggestionList.options.forEach[println(text)]
  }
}
```

Listing 3: Unittest zum Testen der Google-Suchseite

Schaffung von hilfreichen DSLs, APIs oder Patterns. Dem Testentwickler sind die Domänenkonzepte bekannt und aus diesem Grund ist es sinnvoll, wenn nicht sogar notwendig, dass er auch bei der Testentwicklung mit genau dieser Domäne arbeitet.
Die dargestellte Domäne ist nicht neu und so hat sich daraus vor allem das PageObject-Pattern (**siehe Kasten 2**) etabliert. Dieses Entwurfsmuster repräsentiert eine HTML-Seite gegenüber dem Testcode und stellt alle für den Test benötigten Funktionalitäten bereit. Die technische Bindung an die HTML-Seite befindet sich innerhalb des PageObject beziehungsweise innerhalb der darin verwendeten UI-Elemente.
Im Folgenden zeigen wir am Beispiel unseres Test-Frameworks, wie man dieses Pattern umsetzen kann. Wir machen dabei intensiven Gebrauch von Xtend (**siehe Kasten 3**).

Listing 2 zeigt ein PageObject für die Google-Webseite. Auffällig ist, dass sich hier unsere Domänenkonzepte wiederfinden:

- Die Klasse ist mit @Page annotiert.
- Die Klasse umfasst mehrere UI-Elemente, denen wir jeweils Namen geben.
- Jedes UI-Element ist vom Typ einer UI-Komponente.

Zusätzlich erfolgt noch eine Bindung an die HTML-Seite über den Selektor, der der SeleniumElement-Annotation übergeben wird. Dieses Binding wird aber nicht zum Testcode exponiert.
Die UI-Komponenten spiegeln exakt die Möglichkeiten wider, die auch der Benutzer hat, wenn er über die Weboberfläche mit der Komponente interagiert. So umfasst das TextField-Interface zum Beispiel folgende Methoden:

- String getText( ): Liefert den Text des Feldes.
- void setText (String): Schreibt den übergebenen Text in das Feld.
- boolean isEnabled(): Prüft, ob das Textfeld schreibbar ist.
- boolean isDisplayed(): Prüft, ob das Textfeld angezeigt wird.

UI-Komponenten können beliebig komplex und geschachtelt sein. Wie sie implementiert werden, betrachten wir zunächst nicht genauer, da wir in diesem Abschnitt die Rolle des Testentwicklers einnehmen.
Durch die PageObjects, UI-Elemente und UI-Komponenten haben wir eine Abstraktion geschaffen, die dafür sorgt, dass der Testcode sich rein auf die Fachlichkeit beschränkt (**siehe Listing 3**).
Kommentare sind nicht mehr notwendig, da die einzelnen Anweisungen wie eine Spezifikation gelesen werden können

PageObject-Pattern [Fow13]

Problem: Testcode interagiert direkt mit dem Code. Dies führt zu Problemen bei der Nachvollziehbarkeit und Wartbarkeit

Context: UI-Tests

Forces: Kapselung, Modularisierung, Abstraktion

Solution:
PageObjects dienen als Schnittstelle zwischen der HTML-Seite und dem Test.

Dem Testentwickler werden dabei die gleichen Interaktionsmöglichkeiten bereitgestellt, als würde er die Anwendung mit dem Browser manuell bedienen.

Innerhalb des PageObject erfolgt die Bindung an die HTML-Seite. Dies ist aber nicht Bestandteil des API und bleibt für den Testentwickler verborgen.

PageObjects führen selbst grundsätzlich keine Prüfungen durch, da sie kein Wissen über die Fachlichkeit des SUT haben.

Kasten 2

Xtend

Xtend ist eine moderne, statisch getypte, General-Purpose-Programmiersprache auf Basis der Java Virtual Machine (JVM). Syntaktisch und semantisch ist sie eng an Java angelehnt, wertet Java aber durch eine kompaktere Syntax und Extension-Methoden auf. Der Xtend-Compiler erzeugt keinen Bytecode, sondern Java-Code, der dann vom Java-Compiler übersetzt wird.

Ihren Ursprung hat die Sprache im DSL-Engineering. Sie besitzt ihre Stärken unter anderem in der Erzeugung von mehrzeiligen, eingerückten Strings und hat sich dadurch vor allem bei der Entwicklung von Code-Generatoren etabliert.

Durch Verwendung sogenannter Active Annotations können Konzepte der Domäne (in unserem Fall UI-Test) oder Patterns in Xtend hinterlegt werden. Ein Annotation-Prozessor kennt diese Konzepte und kann daher entsprechende Validierungen durchführen. Darüber kann er sich in den Kompilierungsprozess (von Xtend nach Java) einklinken und somit Domänenkonzepte in den Java-Code integrieren.

Kasten 3

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
class ListboxImpl extends AbstractSingleSeleniumElement
    implements Listbox {
  @Autowired
  SeleniumElementFactory seleniumElementFactory

  override getOptions() {
    val webElements = webElement.findElements(
      By.xpath("//div[@class='sbqs_c']"))
    webElements.map[webElement |
      seleniumElementFactory.getSeleniumElement(
        webElement, Link)]
  }
}
```

Listing 4: Implementierung einer UI-Komponente

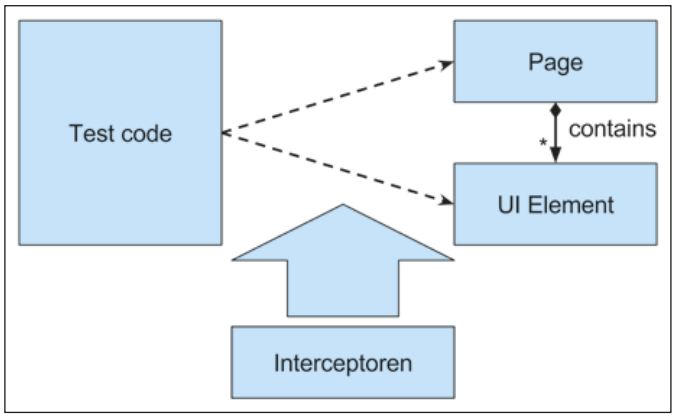


Abb. 4: Integration von Interceptoren

("Good code is its own best documentation" - *Steve McConnell*). Es handelt sich um einen JUnit-Test und als ebensolcher kann er auch aus der IDE oder vom Build-Werkzeug ausgeführt werden.
Da unser Test-Framework Spring als IoC-Container einsetzt, können wir die Pages mithilfe von @Autowired in unseren Test injizieren. Auch ausgelagerte Codeblöcke (wir bezeichnen diese als *Actions*) könnten auf diese Weise eingebunden werden. Damit der IoC-Container auch im Unittest-Umfeld genutzt werden kann, muss die Testklasse lediglich mit @UnitTest annotiert werden.

Architektursicht

UI-Komponenten repräsentieren die Möglichkeiten des Anwenders
Durch das domänengetriebene Design der Programmierschnittstelle erleichtern wir dem Testentwickler seine Arbeit, da er sich vollständig auf die Fachlichkeit konzentrieren kann. Viele technische Aspekte, wie das Warten, bis die Google-Vorschlagliste erscheint, sind bisher allerdings nicht wieder aufgegriffen worden. Genau das zeigt die Mächtigkeit einer domänenspezifischen Programmierschnittstelle: Aus Testentwicklersicht sind diese Aspekte irrelevant.
Dennoch müssen sie natürlich gelöst werden. Dazu wechseln wir die Perspektive vom Testentwickler zum Framework-Entwickler und betrachten die Architektur.
Während dem Testentwickler nur die Schnittstelle zu den Pages und ihren UI-Elementen und -Komponenten bekannt ist, müssen diese natürlich dennoch einmalig implementiert werden. Vor allem Standard-UI-Komponenten wie Textfelder können dabei ständig, auch innerhalb komplexerer Komponenten, wiederverwendet werden.
Ein Beispiel für ein solche UI-Komponente haben wir bereits kennengelernt: die Listbox aus unserem Google-Beispiel. Die Listbox-Komponente ist ein wiederverwendbares Control der Google-Webseite. Ihre Programmierschnittstelle ist sehr simpel:

- List<Link> getOptions(): Liefert alle Optionen als Link.

Das Link-Interface ist ähnlich simpel und umfasst unter anderem die folgenden Methoden:

- String getText( ): Liefert den Text des Links.
- void click(): Klickt den Link an.

Hier sieht man, dass es sinnvoll ist, UI-Komponenten zu schachteln.
Die Implementierung erfolgt in Xtend. Teile des Codes in **Listing 4** haben wir bereits in dem WebDriver-Beispiel gesehen. Es werden alle Elemente anhand eines xpath-Ausdrucks selektiert und ein entsprechender Link dafür erzeugt. UI-Komponenten interagieren in der Implementierung also mit der WebDriver-Programmierschnittstelle. Der Testcode wird hingegen nur gegen das Interface entwickelt.
Da Spring nach allen mit @Component annotierten Klassen sucht, ist es lediglich notwendig, ListboxImpl damit zu annotieren. Die Bindung an das Listbox-Interface erfolgt automatisch.

Aspekte identifizieren und implementieren
Gerade in sehr dynamischen Webanwendungen, die auf modernen JavaScript-Frameworks basieren, wird der HTML-DOM ständig verändert. Genau dies führt in der Praxis zu großer Instabilität (Flakiness) der Oberflächentests. In dem WebDriver-Beispiel wurde ausprogrammiert, dass auf das Erscheinen eines Elementes (bis zu einem Timeout) gewartet werden soll. Wenn man sich vorstellt, dass eine solche Behandlung quasi bei jeder Interaktion mit einem Oberflächenelement gemacht werden muss, wird schnell klar, dass dies zwangsläufig zu unlesbarem und unwartbarem Testcode führt.
Das Test-Framework wartet daher immer, wenn mit einem UI-Element interagiert werden soll, bis dieses auf der Oberfläche erscheint. Das Verhalten könnte man innerhalb der Komponentenimplementierung umsetzen, allerdings hat sich in der praktischen Nutzung herausgestellt, dass es sich vielmehr um einen querschnittlichen Aspekt handelt. Genau genommen sind Aspekte nicht Teil der Domäne „UI-Test". Vielmehr haben sie unterstützenden Charakter, um die Domänenkonzepte auf eine ausführbare Ebene zu heben, ohne sowohl den Testcode als auch die Komponentenimplementierung zu „verschmutzen".
Auf die Softwarearchitektur projiziert, lassen sich solche Aspekte mithilfe von Interceptoren realisieren. Sie werden unter anderem immer dann genutzt, wenn der Testcode mit den UI-Elementen interagiert. **Abbildung 4** skizziert, dass sowohl der Test als auch die Page beziehungsweise UI-Komponente dabei keinerlei Wissen über die Interceptoren haben und daher unabhängig von ihnen sind.
Neben den typischen Anwendungsfällen wie Logging, Monitoring oder Sicherheitsprüfungen eröffnet *Aspektorientierte Programmierung (AOP)* im Kontext der Oberflächentests die Möglichkeit, unerwartete technische Fehlersituationen zu behandeln. Dabei stehen verschiedene Strategien zur Verfügung:

- Wenn ein Element nicht gefunden wird, wird der Fehler nicht direkt zum Test durchgereicht, sondern es wird mit einem zu konfigurierenden Timeout noch einmal versucht, das Element zu lokalisieren

```
@TestClass
@UseExtension(BrowserInteractionService)
class GoogleSuggestTest {
  @Autowired
  GooglePage googlePage

  @Step
  def void openWebsite() {
    openURL("http://google.com/?hl=en")
  }

  @Step
  def void printSuggestions() {
    googlePage.queryField.text = "Cheese"
    googlePage.suggestionList.options.forEach[println(text)]
  }
}
```

Listing 5: Systemtest zum Testen der Google-Suchseite

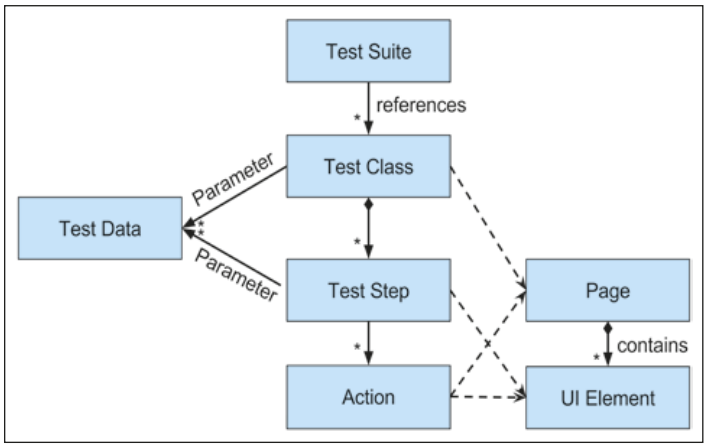


Abb. 5: Domäne „UI-Test" im Kontext von Systemtests

- Wenn die Seite neu geladen wurde oder ein Element (kurzzeitig) aus dem HTML-DOM verschwindet, hält der Test noch eine Referenz auf ein veraltetes Element. Im Selenium-Kontext tritt eine *StaleElementReferenceException* auf. Interceptoren können diese Fehler auffangen und das Element über die angegebenen Selektoren erneut lokalisieren.

Diese Mechanismen sorgen dafür, dass der Test stabiler läuft, da das Test-Framework wesentlich robuster wird und nicht direkt bei einem unerwarteten technischen Verhalten aussteigt. Die Erfahrung hat gezeigt, dass neben der Wartbarkeit vor allem die Robustheit eine große Hürde für Oberflächentests darstellt, dessen Wichtigkeit sich durch den immer größer werdenden Funktionsumfang moderner Browser und komplexer JavaScript-Frameworks weiter erhöht. Für Testautomation ist Instabilität verheerend, da man sich nicht darauf verlassen kann, dass ein fehlgeschlagener Test wirklich auf einen Fehler des SUT zurückzuführen ist, was massive Analyseaufwände zur Folge hat. Die Stakeholder verlieren dadurch schnell das Vertrauen in die Tests.

Interceptoren können aber nicht nur für Synchronisationsaspekte genutzt werden. So ist es möglich, dass immer bevor der Test mit einem UI-Element interagiert, zunächst geprüft wird, ob der Browser sich aktuell überhaupt auf der erwarteten Webseite befindet. Dadurch können zum einen sehr exakte Fehlermeldungen ausgegeben werden und zum anderen ist sichergestellt, dass der Test nicht mit einer falschen Seite interagiert.

Jede Page kann optional ein Interface implementieren und dadurch selbst entscheiden, ob sie gerade die im Browser aktive Seite ist. Immer wenn der Testcode mit einem UI-Element interagiert, wird die zugehörige Page vom Framework gefragt, ob sie überhaupt gerade aktiv ist. Ist dies nicht der Fall, schlägt der Test fehl.

Systemtest statt Unittest

In diesem Artikel haben wir uns bisher auf die Schnittstelle zwischen Test und der zu testenden Software konzentriert. Das API haben wir dabei im Rahmen von Unittests eingesetzt. Unittests sind in diesem Kontext sinnvoll, wenn einzelne UI-Komponenten getestet werden sollen.

UI-Regressionstests haben grundsätzlich aber einen weitaus integrativeren Fokus und operieren daher eher auf der Ebene von Systemtests. Tests werden aus der Perspektive des Anwenders formuliert und sind somit tendenziell eher auf Anwendungsfälle bezogen.

Unittests im Allgemeinen und JUnit im Speziellen sind für diese Art von Tests nicht konzipiert. In JUnit dürfen Testmethoden keinerlei Abhängigkeiten zueinander haben, was dadurch unterstützt wird, dass die Ausführungsreihenfolge nicht spezifiziert und somit nicht deterministisch ist. Würden wir Systemtests mit JUnit umsetzen, müsste also der komplette Test innerhalb einer Testmethode ablaufen. Nicht nur aus Reporting-Sicht ist das unzumutbar.

Wie auch schon bei der Schnittstelle zwischen Test und SUT nähern wir uns der Problemlösung über die Domäne (**siehe Abbildung 5**). Test Suites und Test Classes sind Elemente, die auch in der Unit-Testing-Domäne existieren. Der entscheidende Unterschied liegt in den Test Steps, die sequenziell nacheinander ausgeführt werden. Im Gegensatz zu Unittest-Methoden sind die Steps immer von ihren Vorgängern abhängig und werden auch nur ausgeführt, wenn die Vorgänger erfolgreich ausgeführt wurden. Sowohl Test Classes als auch Test Steps können parametrisiert sein. Test Steps sind nicht wiederverwendbar, allerdings können wiederkehrende Bausteine in Actions ausgelagert werden, die von mehreren Steps genutzt werden.

Sowohl Test Classes/Steps als auch Actions können mit Pages und deren UI-Elementen interagieren.

Xtend bietet durch Active Annotations ein starkes Konzept, um domänenspezifische Konzepte zu integrieren. Dieses machen wir uns zunutze, indem wir unsere Domäne mithilfe von Annotationen abbilden. Kommen wir zum besseren Verständnis noch einmal auf den GoogleSuggestTest zurück. Übertragen auf die skizzierte Systemtest-Domäne, sieht der Test wie in **Listing 5** aus.

Die Unterschiede wirken zunächst marginal. Statt mit @UnitTest ist die Klasse mit @TestClass annotiert. Des Weiteren wurde die @Test-annotierte Methode durch zwei Steps ersetzt. Die Vorteile durch die Berücksichtigung der Domäne werden offensichtlich, wenn man den Test wie im Folgenden weiterentwickelt.

Oberflächentests liefern bereits dadurch hilfreiche Erkenntnisse, ob sie erfolgreich durchlaufen oder nicht. Dennoch ist es sinnvoll, an geeigneten Stellen fachliche Prüfungen zu hinterlegen. In unserem Beispiel bietet es sich an, die Google-Vorschläge mit einer erwarteten Vorschlagmenge abzugleichen. Diese kann statisch im Testcode hinterlegt werden. Sinnvoller ist es aber, Testdaten und Testcode voneinander zu trennen. Die Vorteile betrachten wir detaillierter, wenn wir uns mit den Skillsets der beteiligten Personen beschäftigen.

In unserem Beispiel gehen wir auf die Datenherkunft nicht genauer ein, aber denkbar wäre, dass Excel-Dateien, csv-Dateien, Datenbanken oder REST-Services eingesetzt werden.

Die Testdaten könnten beispielsweise folgendermaßen in einer csv-Datei hinterlegt sein:

```
@Step
def void testSuggestions(@IteratedParameter @Resource(
      "classpath:searchTermExpectations.csv"))
      SearchTermExpectation searchTermExpectation) {
  googlePage.queryField.text = searchTermExpectation.searchTerm
  assertThat(googlePage.suggestionList.options.map[text],
    hasItems(searchTermExpectation.expectedSuggestions))
}
```

Listing 6: Parametrisierter Testschritt

```
searchTerm;expectedSuggestions
Cheese;cheese,cheesecake
Sausage;vegetable
```

Auch beim Zugriff auf die Daten gilt das Paradigma, dass der Testentwickler sich nicht mit technischen Aspekten wie dem Parsen der csv-Datei befassen muss. Er kann die Datei mittels einer Annotation anziehen. Der Schritt *testSuggestions* wird dann für jeden Datensatz sequenziell aufgeführt, wobei der entsprechende Datensatz jeweils der Methode übergeben wird (**siehe Listing 6**).

Den Adapter, der die Einträge in der csv-Datei auf das Object SearchTermExpectiation mappt, betrachten wir an dieser Stelle nicht näher. Er ist aber Bestandteil des Showcases (vgl. [ShowCase]).

Die Domänenkonzepte bei der Implementierung der Tests zu nutzen, ist aber nur eine Seite der Medaille, denn wir benötigen auch eine *Execution Runtime*, die die Domäne kennt und sie ausführbar macht. Hier haben wir uns für eine eigene, leichtgewichtige Lösung, die exakt auf die Domäne zugeschnitten wurde, entschieden. **Abbildung 6** illustriert die Funktionsweise. Beim Start einer Test Suite oder einer Test Class sammelt die Execution Runtime alle auszuführenden Tests und führt sie anschließend aus, indem die Test Classes instanziiert und die einzelnen Steps durchlaufen werden. Testentwickler können festlegen, ob die Tests parallel oder sequenziell durchgeführt werden. Die Steps einer Test Class können gemäß ihrer Semantik nur sequenziell durchgeführt werde.

Die Runtime ist von Grund auf so offen wie möglich implementiert, um sich nahtlos in bestehende Tools und Frameworks zu integrieren. Beim Starten, Beenden oder Überspringen von Test Suites, Test Classes und Steps werden Events geworfen, auf die entsprechende Listener reagieren können.

Konkret ist dies vor allem für Reporting-Tools und -Frameworks interessant. Ein konkretes Beispiel ist der JUnitExecutionListener, der die Events der Runtime entgegennimmt und alle registrierten JUnitRunListener notifiziert. Da das JUnitRunListener-Interface von vielen gängigen Tools wie Eclipse oder Maven bereits implementiert wird, gelingt dadurch eine Integration. Mit anderen Worten: JUnit dient nicht als Ausführungsengine, sondern wir nutzen es lediglich als Reporting-Werkzeug.

Für den Testentwickler verhält sich die Ausführung wie bei einem klassischen JUnit-Test. Er kann die Ausführung aus Eclipse heraus anstoßen und in der JUnit-Ansicht (**siehe Abbildung 7**) die einzelnen

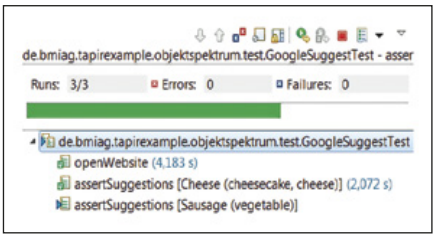


Abb. 7: Screenshot der Testausführung in der Entwicklungsumgebung

Testschritte und den Fortschritt ablesen. Dass eigentlich gar nicht JUnit die Ausführung übernimmt, bleibt verborgen.

Durch diese Entkopplung sind somit weitreichende Integrationsszenarien denkbar. Wir haben dabei unter anderem eine Anbindung an Allure [All] implementiert, das wir als unser primäres Reporting-Tool einsetzen.

Organisationssicht

Nur beiläufig haben wir uns bisher mit der Organisationssicht beschäftigt. Die Separierung von UI-Komponenten, PageObjects, Testcode und Testdaten lässt sich direkt auf die Organisation mithilfe des Skillstacks aus **Abbildung 8** abbilden.

Die Basis bildet dabei die Schaffung von UI-Komponenten. Sie erfordert ein tiefes Know-how des Test-Frameworks und des Selenium WebDriver API. Diese Tätigkeit kann allerdings gänzlich entfallen, wenn für das eingesetzte UI-Framework bereits eine Implementierung existiert.

Für die Erzeugung der PageObjects sollte man mindestens rudimentäre HTML-Kenntnisse haben, um die Elemente sinnvoll zu lokalisieren. Die PageObjects können dann aus beliebig vielen Tests heraus verwendet werden. Für Softwarehersteller eröffnet sich sogar ein weiteres Spektrum: Neben dem Einsatz der PageObjects und UI-Komponenten für interne Tests können diese aber auch mit der Software ausgeliefert werden. Die Kunden können auf diese Weise eigene Tests gegen das API entwickeln.

Für die Erstellung der Testfälle werden sowohl fachliches Know-how als auch

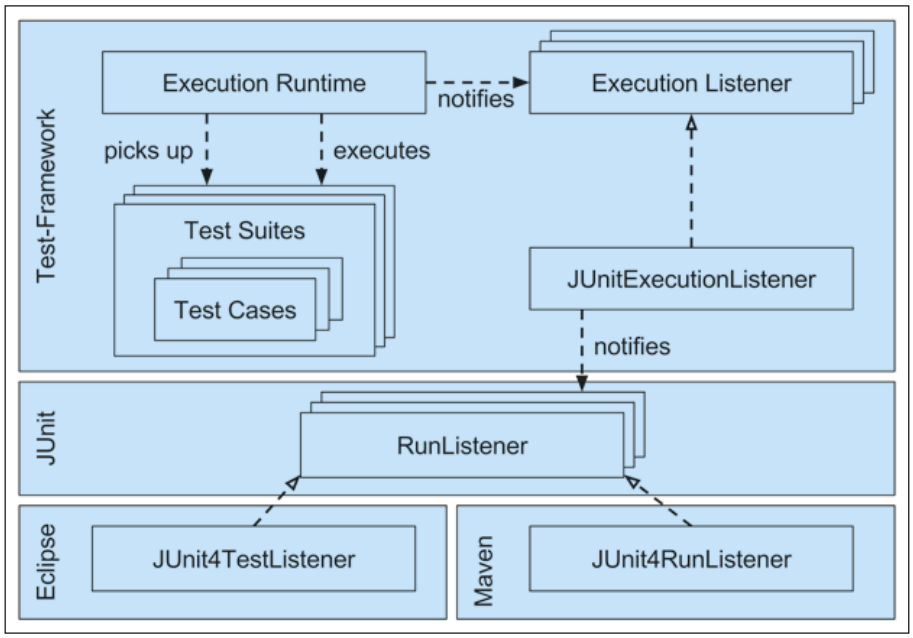


Abb. 6: Funktionsweise der Execution Runtime

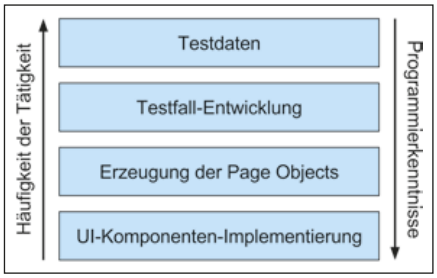


Abb. 8: Skillstack

Literatur & Links

[All] Allure Test Report, siehe: **http://allure.qatools.ru/**

[Fow13] M. Fowler, PageObject, 2013, siehe: **https://martinfowler.com/bliki/PageObject.html**

[Sel] Selenium Getting Started Guide, siehe: **https://github.com/SeleniumHQ/selenium/wiki/Getting-Started**

[ShowCase] Codebeispiele, siehe: **https://github.com/tapir-test/tapir-showcase**

[Sta06] T. Stahl, M. Völter, Model-Driven Software Development, Wiley, 2006

[Tapir] UI Regressiontest Framework, siehe: **https://www.tapir-test.io**

mindestens grundlegende Programmierkenntnisse benötigt. Der Testentwickler muss zum einen das Test-API kennen und benutzen können und zum anderen ein Verständnis für die zu testenden Funktionen des SUT haben.
Die Testdaten können von reinen Fachexperten zum Beispiel in Form von Excel-Tabellen spezifiziert werden. Wenn die Tests datengetrieben aufgebaut wurden, haben Fachexperten somit ein mächtiges Werkzeug an der Hand, um verschiedene fachliche Konstellationen testen zu können, ohne selbst Testcode schreiben zu müssen.

Synergien mit DSL-Engineering in der Anwendungsentwicklung

Die Stabilität und Robustheit von Oberflächentests ist eng mit der Qualität der Selektoren verknüpft. In den PageObjects werden Oberflächenelemente über verschiedene besser oder schlechter geeignete Strategien lokalisiert: css-, xpath-, name- oder id-Attribute sind die geläufigsten. Speziell xpath ist dabei tendenziell langsam und vor allem fragil gegenüber Refactorings, da es sich direkt am HTML-DOM orientiert.
Robuste Selektoren zu wählen, stellt daher die größte Herausforderung bei der Erstellung von PageObjects dar. Leider gibt es bisher keine Software, die diese Aufgabe zufriedenstellend löst, sodass Selektoren in den PageObjects meist per Hand und über das Inspizieren des HTML-DOMs erzeugt werden.
Durch den Einsatz von DSL-Engineering/Model-Driven (Software) Development (MDSD, vgl. [Sta06]) in der Anwendungsentwicklung können Softwarehersteller dieses Problem auf äußerst elegante Weise lösen. Mit einer UI-DSL ist es möglich, Oberflächen auf einer abstrakten Ebene zu deklarieren. Neben dem gängigen Muster, daraus ausführbaren Code für unsere Anwendung zu generieren, eröffnet uns die DSL im UI-Testumfeld eine weitere spannende Möglichkeit: Da der Generator weiß, wie die Bestandteile der UI-DSL in die Anwendung übersetzt werden, kann er entsprechende PageObjects und darin enthaltene UI-Elemente für den UI-Test einschließlich der entsprechenden Selektoren ebenfalls komplett generieren. **Abbildung 9** zeigt dies schematisch.

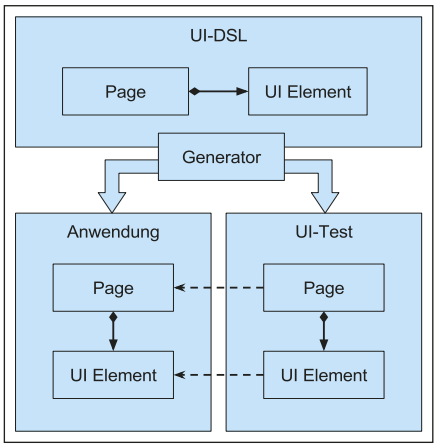
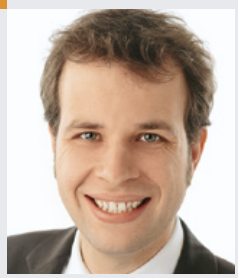


Abb. 9: Zusammenspiel zwischen DSL-Engineering in Anwendungs- und Testentwicklung

Fazit

Robuste Oberflächentests nachvollziehbar und wartbar zu halten, ist ein komplexes Unterfangen. In diesem Artikel haben wir gezeigt, wie man mit modernem Software-Engineering diese Probleme angehen und somit erhebliche Fortschritte erzielen kann.
Der Fokus auf die Domäne ist hierbei das zentrale Konzept, das eine große Palette an Möglichkeiten eröffnet: Trennung von fachlichen und technischen Aspekten, Externalisierung von Testdaten, Integration in etablierte Werkzeugketten und letztlich die Einbringung von Expertenwissen auf einem jeweils angepassten Abstraktionsniveau. ||

Die Autoren



Oliver Libutzki
(oliver.libutzki@bmiag.de)
arbeitet als Softwarearchitekt bei der b+m Informatik AG in Melsdorf. Seit 2006 beschäftigt er sich mit DSL-Engineering, vor allem im Java-Umfeld. Oliver Libutzki ist für das Design und die Implementierung des UI-Testautomation-Frameworks tapir verantwortlich.

Thomas Stahl
(thomas.stahl@bmiag.de)
ist Lead Architect und CTO der b+m Informatik AG. Er beschäftigt sich seit über 20 Jahren intensiv mit Software-Engineering und -Architektur. Er ist Buchautor und Sprecher auf Konferenzen.

Nils Christian Ehmke
(nils-christian.ehmke@bmiag.de)
ist Softwarearchitekt der b+m Informatik AG in Melsdorf. Er arbeitet an Business-Lösungen für die Finanzwirtschaft mithilfe von DSL-Engineering. Nils Ehmke ist für das Design und die Implementierung des UI-Testautomation-Frameworks tapir verantwortlich.