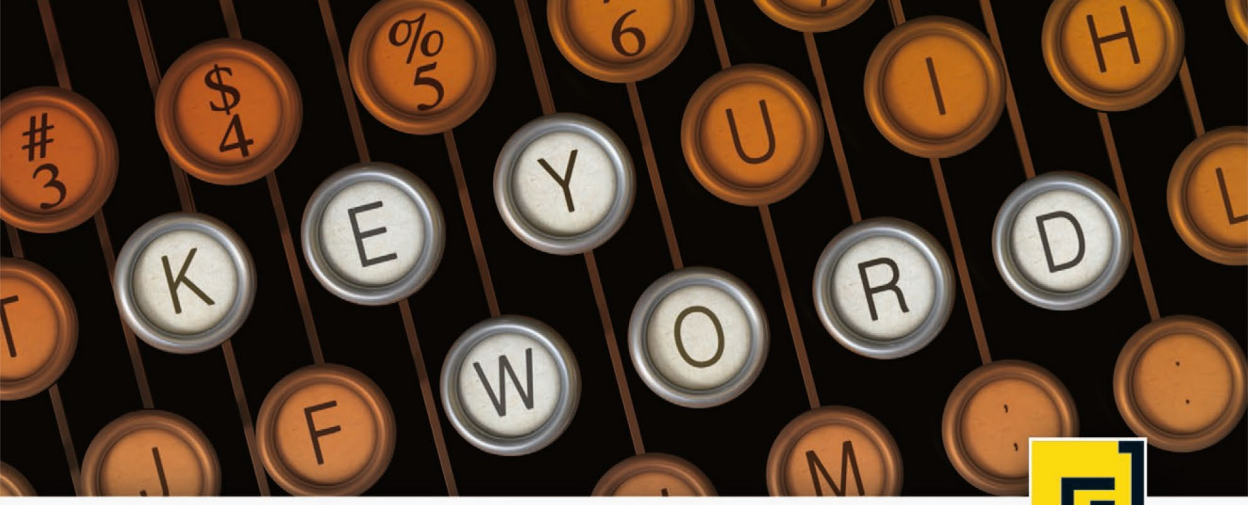



Matthias Daigl · René Rohner

# Keyword-Driven Testing

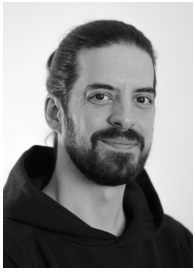
Grundlage für effiziente Testspezifikation und Automatisierung

Openbook zu »Keyword-Driven Testing Frameworks«

ISBN: 978-3-86490-570-4

dpunkt.verlag

**Matthias Daigl** ist Product Owner bei der imbus AG. Er ist als Sprecher auf internationalen Konferenzen unterwegs, arbeitet in Arbeitsgruppen des German Testing Board, des ISTQB® und im Normungsausschuss von DIN und ISO mit, war Editor der Norm ISO/IEC/IEEE 29119-5 »Keyword-Driven Testing« und ist Autor des Buches »ISO 29119: Die Softwaretest-Normen verstehen und anwenden«.



**René Rohner** ist Product Owner des Value Streams Testautomatisierung sowie Senior Berater mit den Spezialgebieten Keyword-Driven Testing und Testautomatisierung bei der imbus AG. Er ist als Softwareentwickler, Trainer und Chairman of the Board der Robot Framework® Foundation international im Bereich des Keyword-Driven Testing tätig.

**Matthias Daigl · René Rohner**

# Keyword-Driven Testing

## Grundlage für effiziente Testspezifikation und Automatisierung

dpunkt.verlag

Matthias Daigl
*KDT@daigl.de*

René Rohner
*buch@keyword-driven.de*

Lektorat: Christa Preisendanz
Lektoratsassistenz: Julia Griebel
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Matthias Daigl
Herstellung: Stefanie Weidner, Frank Heidt
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-86490-570-4
PDF 978-3-96088-482-8
ePub 978-3-96088-483-5
mobi 978-3-96088-484-2


Wieblinger Weg 17
69123 Heidelberg

*Hinweis:*
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

PEFC zertifiziert
Das Papier für dieses Buch stammt aus nachhaltig bewirtschafteten Wäldern und kontrollierten Quellen.
PEFC
PEFC/04-31-0810
www.pefc.de

*Schreiben Sie uns:*
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *hallo@dpunkt.de*.



5 4 3 2 1 0

# Vorwort

In Gesprächen mit anderen Softwaretesterinnen und -testern haben wir die Erfahrung gemacht, dass das Konzept »schlüsselwortbasierter Test« oder »Keyword-Driven Testing« vielen ein Begriff ist. Andererseits ist es uns öfter so gegangen, wenn jemand darüber sprach, dass uns der Gedanke kam – *ja, das ist es auch, aber nicht nur …*

Im Kern geht es darum, Tests besser zu formulieren, für Mensch und Maschine.

Hinter dem Begriff »Keyword-Driven Testing« verbirgt sich dabei mehr, als es auf den ersten Blick scheint. Auch macht es zunächst den Eindruck, dass es ganz einfach ist. Gleichzeitig hat es viele Facetten und bietet zahlreiche Möglichkeiten, die im Alltag beim Softwaretest helfen können, aber einfach nicht bekannt genug sind.

Vieles davon ließe sich auf die Formel bringen: »Nutzt bewährte Praktiken aus der Softwareentwicklung und nutzt imperative Programmierung, um Tester und Testerinnen sowie Testautomaten die Testanweisungen zu übermitteln. Haltet euch dabei an die allgemeingültigen Best Practices, was Lesbarkeit und Verständlichkeit angeht.«

Natürlich ist es nicht so einfach. Was wir über Keyword-Driven Testing wissen, haben wir über Jahrzehnte gelernt. Inzwischen haben wir das Thema in der Theorie vertreten, auf Ebene von ISO-Normen, und in der Praxis in einer Vielzahl von Beratungsprojekten Teams beim Einstieg in Keyword-Driven Testing unterstützt. Weitere Interessierte können wir jedoch nicht persönlich erreichen und gleichzeitig gibt es einfach noch zu wenig Literatur zum Thema.

Auftrag angenommen: Keyword-Driven Testing muss bekannter und gut aufbereitet zugänglicher werden.

Sie halten nun unseres Wissens das erste Buch in den Händen, das sich ausschließlich mit dem Thema »Keyword-Driven Testing« befasst. Wir hoffen, dass Sie darin viele Anregungen finden, Ihre Tests effizienter und wartbarer zu gestalten. Viel Erfolg damit!

Erlangen, im Februar 2022
Matthias Daigl

Königswinter, im Februar 2022
René Rohner

# Dank

Unser Dank gilt an dieser Stelle den Kolleginnen und Kollegen sowie Personen aus dem Freundeskreis, die uns beim Schreiben dieses Buches zur Seite gestanden sind. In vielen Gesprächen haben sie sich Zeit für uns und unser Vorhaben genommen, ihre Erfahrungen und Ideen mit uns geteilt und uns Inspiration geschenkt.

Ganz herzlich danken wollen wir auch dem dpunkt.team und unserer Lektorin, Christa Preisendanz, für die schier unendliche Geduld mit uns. Wir haben es ihnen nicht leicht gemacht.

Am meisten wollen wir aber unseren Familien danken, die uns jederzeit unterstützt haben, oft in den Schreibphasen auf uns verzichten mussten und uns wieder aufgerichtet haben, wenn es nicht vorangehen wollte. Ohne Euch hätten wir das nicht geschafft!

# Inhaltsverzeichnis

# 1 Einführung

Dieses Buch hat das Ziel, das Thema »Keyword-Driven Testing« möglichst umfassend zu beleuchten, unter anderem, damit Keyword-Driven Testing häufiger und intensiver eingesetzt wird. Aber was verstehen wir überhaupt unter Keyword-Driven Testing?

Keyword-Driven Testing ist eine Methode zur Spezifikation von Testfällen im Sinne einer Notation. Es macht Testen effizienter. Es entfaltet seinen Nutzen vor allen Dingen in Teams, hilft aber auch einzelnen Testerinnen und Testern. Und entgegen anderslautender Gerüchte ist »Keyword-Driven Testing« nicht nur für Testautomatisierung hilfreich – aber gerade auch dort.

Das erwartet Sie nun in den einzelnen Kapiteln:

*Überblick über das Buch*

**Einführung – dieses Kapitel**: Was »Keyword-Driven Testing« ist, warum man es nutzen sollte, Begriffe und die Einführung eines Beispiels, auf das sich das Buch im weiteren Verlauf bezieht. *Ergebnis: Sie wissen, worum es geht.*

**Konzepte:** Unterschiedliche (einfache und komplexe) Ansätze, mit denen »Keyword-Driven Testing« genutzt werden kann. *Ergebnis: Sie wissen, was Keyword-Driven Testing ist, und kennen Zusammenhänge.*

**Umsetzung:** Wie man Keywords auswählt, wie man sie strukturieren kann, was es aus sprachlicher Sicht zu beachten gilt und wie man Qualitätssicherung für Keywords betreiben kann. Und was das wirtschaftlich bedeutet. *Ergebnis: Sie wissen, wie man mit Keywords umgeht.*

**Keywords und Normen:** Was für Normen es im Testen und speziell zu Keywords gibt und was die Normen zu Keywords und Frameworks sagen. *Ergebnis: Sie wissen, wie Ihnen Normen zu Keyword-Driven Testing helfen können.*

**Testautomatisierungsarchitektur:** Was so alles gebraucht wird, wenn man Testautomatisierung und Keyword-Driven Testing betreiben möchten, und auch wie ISTQB®dazu steht. *Ergebnis: Sie kennen verschiedene Sichtweisen auf die Testautomatisierungsarchitektur und verfügen über die Grundlagen, sich Ihre Architektur im Hinblick auf Keyword-Driven Testing zu erarbeiten.*

**Keyword-Driven Testing Frameworks:** Praxisbeispiele, wie Keyword-Driven Testing in Projekten angewendet wird, einschließlich einer Bewertung der eingesetzten Frameworks. *Ergebnis: Sie haben einen Eindruck von unterschiedlichen Einsatzszenarien von Keyword-Driven Testing.*

**Praxis mit Robot Framework:** Anhand eines konkreten Frameworks zeigen wir, wie eine Umsetzung von Keyword-Driven Testing im Kontext aussehen kann. *Ergebnis: Sie können die Anwendung von Keyword-Driven Testing mit diesem Framework nachvollziehen.*

**Brückenschlag:** Verbindung von Keyword-Driven Testing mit verbreiteten Ansätzen im Softwaretest, die zwar nicht »Keyword-Driven Testing« im engeren Sinne sind, aber gut damit zusammenwirken. *Ergebnis: Sie können Keyword-Driven Testing im Hinblick auf diese Ansätze einordnen.*

**Ausblick:** Wie geht es weiter mit Keyword-Driven Testing. *Ergebnis: Sie haben das Buch geschafft und wissen nun, wie Sie mit Keyword-Driven Testing umgehen wollen!*

## 1.1 Wortwahl

Bevor wir mit dem Inhaltlichen beginnen, gibt es noch zwei Dinge bezüglich der Wortwahl, die uns Autoren am Herzen liegen und die wir daher hier kurz darlegen wollen:

**Geschlechtsspezifische Begriffe:** In diesem Buch werden wir häufig darüber reden, was eine Person – z.B. ein Tester oder eine Testerin – tut, denkt, tun kann ... Wir sind der festen Überzeugung, dass das Geschlecht einer Person grundsätzlich keine Rolle zu spielen hat. Dennoch leben wir im Bereich der Softwareentwicklung und des Softwaretests in einer stark männerdominierten Branche. Gerade deswegen ist uns eine Geschlechtergleichstellung auch besonders in sprachlicher Form wichtig.

Zugunsten einer besseren Lesbarkeit werden wir nicht jedes Mal beide sprachlichen Geschlechter aufzählen. Auch haben wir uns gegen eine Schreibweise wie »TesterIn« oder »Tester*in« entschieden.

Das generisches Maskulinum, also nur die männliche Form, zu verwenden, schied völlig aus.

Unser Weg ist es nun, für die in diesem Buch wichtigsten sechs Rollen uns jeweils zu gleichen Teilen für die weibliche und die männliche Form zu entscheiden und dann zugunsten der Lesbarkeit dabei zu bleiben – in der Hoffnung, dass jede und jeder sich so ausreichend angesprochen und wertgeschätzt fühlt.

Wir reden also von Testerinnen, Testmanagerinnen und Entwicklerinnen sowie von Testdesignern, Anwendern und Testautomatisierern.

**Englisch – Deutsch:** Dies ist ein deutschsprachiges Buch und im Allgemeinen streben wir an, Anglizismen zu vermeiden. Ausgerechnet für den Schlüsselbegriff dieses Buches – »Keyword-Driven Testing« – gibt es zwar mit »schlüsselwortbasiertem Test« eine deutsche Entsprechung, diese ist aber nach unserem Empfinden so wenig gebräuchlich, dass wir uns entschieden haben, innerhalb des Buches dennoch den englischen Begriff »Keyword-Driven Testing« zu bevorzugen. Gleiches gilt für die Verwendung von »Keyword« und »Schlüsselwort«. Die Germanisten mögen uns verzeihen.

## 1.2 Was ist Keyword-Driven Testing

Was verbirgt sich nun hinter dem Begriff »Keyword-Driven Testing«?

Es gibt dazu einige Analogien – ein Favorit: Testfälle werden aus Bausteinen zusammengesetzt. Wir alle erinnern uns gerne an unsere Kindheit. Die Vorstellung, Testfälle spielerisch zusammenzusetzen, ist daher bestimmt attraktiv.

*Modularisierung*

Große Dinge aus kleinen Bausteinen zusammenzusetzen ist aber jenseits vom spielerischen Aspekt eine generell bewährte Vorgehensweise, um Komplexität zu bewältigen. Wir sprechen hier von Modularisierung.

Während Modularisierung in der Softwareentwicklung im Allgemeinen selbstverständlich ist, und das vielleicht auch noch bei Testautomatisierung zutrifft, so ist sie im Zusammenhang mit Testspezifikation bei Weitem nicht so häufig anzutreffen.

Keyword-Driven Testing (KDT) ist eine Methode, um Modularisierung bei der Spezifikation von Testfällen zu erreichen.

Ein Keyword ist in diesem Sinne ein Baustein, aus dem Testfälle zusammengesetzt werden. Dabei steht das Keyword – typischerweise ein Verb zusammen mit einem Substantiv, beispielsweise `Erzeuge User` – für eine oder mehrere Aktivitäten, die bei der Testausführung später auszuführen sind.

Idealerweise stehen beim Erstellen der Testfälle alle benötigten Keywords fertig definiert zur Verfügung. Dann können alle Testfälle aus Keywords zusammengesetzt werden.

**Keine Testtechnik**

Unter »Testtechnik« versteht man meist »Testentwurfsverfahren«. Laut ISO/IEC 29119 [48] geht es dabei darum, auf einem definierten Weg Testfälle zu ermitteln (einem Prozess folgend).

Keyword-Driven Testing ist also keine Testtechnik in diesem Sinne. Unabhängig von Keyword-Driven Testing setzen wir für das Ableiten von Testfällen, also das Testdesign, voraus, dass eine Testtechnik (oder mehrere) eingesetzt werden.

Keyword-Driven Testing legt jedoch fest, wie die aus dem Testdesign entstandenen Testfälle aufgeschrieben oder dokumentiert werden.

## 1.3 Begriffe

Nur die beiden wichtigsten zentralen Begriffe – Keyword und Framework –, die in diesem Buch verwendet werden, sollen hier definiert und erläutert werden. Umfassendere Begriffssammlungen finden sich auf der Webseite von imbus [23] sowie auf der Seite des ISTQB® [32] (als Online-Anwendung), und von IEEE sind die Begriffe von IEEE, ISO, IEC und PMI veröffentlicht und als Norm ISO/IEC/IEEE 2765 [50] käuflich zu erwerben, aber auch im Web legal frei verfügbar [28].

### 1.3.1 Der Begriff »Keyword«

Der zentrale Begriff dieses Buches ist ohne Zweifel »Keyword« oder zu Deutsch »Schlüsselwort«.

*Mehrdeutigkeit*

Der Begriff »Keyword« selbst ist eigentlich ein Teekesselchen[1]: Man versteht darunter, je nach Zusammenhang, die Bezeichnung (im Beispiel oben »Erzeuge User«), die dahinter liegenden Aktivitäten (könnte hier das Auswählen eines Menüeintrags sein, dann das Ausfüllen der Felder eines Dialoges ...) oder ein vielleicht vorhandenes zugeordnetes Automatisierungsskript. Oder dies alles gemeinsam.

[1] Ein »Teekesselchen« – das Fachwort wäre Homonym – ist ein Begriff, der mehrere Bedeutungen hat. In einem Kinderspiel lässt man raten: »Mein Teekesselchen kann schwimmen oder man braucht es zum Zelten – was ist das?« – Antwort: ein Hering.

In der Norm ISO 29119-5 [49] ist dieser Begriff folgendermaßen definiert:

| Keyword | Schlüsselwort |
| --- | --- |
| one or more words used as a reference to a specific set of actions intended to be performed during the execution of one or more test cases | ein oder mehrere Wörter, die als Verweis auf eine festgelegte Menge von Aktivitäten verwendet werden und die während der Durchführung eines oder mehrerer Testfälle ausgeführt werden sollen |
| (Definition laut [49]) | (Übersetzung der Autoren) |

Ein Keyword besteht demnach aus ein oder mehreren Wörtern, die beschreiben, was an der Stelle, an der diese Wörter in einem Testfall vorkommen, getan werden soll. Ein Beispiel für ein einzelnes Wort könnte `Login` sein, eines für mehrere Wörter `Datensatz anlegen`. Meist wird ein Keyword ein Verb enthalten (so fordert es auch die ISO 29119-5), allein schon deswegen, weil ja durch das Keyword eine auszuführende Tätigkeit beschrieben ist.

Auch wenn es rein technisch gesehen nicht notwendig ist, Keywords mit einem Verb zu bilden, so hat es sich doch bewährt. Die Lesbarkeit profitiert davon, und daher ist es empfehlenswert, sich nach dieser Regel zu richten.

Um zu der Begriffsfrage zurückzukehren: Testautomatisierer verstehen unter einem »Keyword« gerne eine aufrufbare Funktion oder ein Skript, die bzw. das im Sourcecode in einem Testautomatisierungswerkzeug implementiert ist. Testerinnen denken dagegen vielleicht eher an eine Zusammenfassung mehrerer Testschritte zu einer fachlichen Tätigkeit.

Ein Keyword kann beides sein.

Ein Keyword kann eine automatisierte Funktion widerspiegeln oder manuelle Testschritte zusammenfassen. Und: Ein Keyword kann sogar eine Zusammenfassung anderer Keywords sein.

Häufig wird unter dem Begriff »Keyword« einfach der Name des Keywords (hier: `Benutzer anmelden`) verstanden – der ja im Alltag, beim Spezifizieren von Tests, auch am meisten gebraucht wird. Im Hinterkopf sollte man aber behalten, dass dazu einiges mehr gehört.

In Abschnitt 2.1 »Verschlagwortung« werden wir auf die sprachlichen Aspekte, die hier nur angerissen wurden, noch sehr viel genauer eingehen.

### 1.3.2 Der Begriff »Framework«

*Framework*

Den Begriff »Framework« benutzen wir in diesem Buch bezogen auf Keyword-Driven Testing. Alleine stehend ist er ein zu gängiges Wort der natürlichen englischen Sprache, als dass es einer ISO oder IEEE in den Sinn kommen würde, dafür eine Definition anzugeben. Im Zusammenhang mit Testautomatisierung bietet jedoch das ISTQB®eine Definition, dankenswerterweise sowohl auf englisch als auch auf deutsch:

| *Test automation framework* | *Testautomatisierungsframework* |
|---|---|
| A tool that provides an environment for test automation. It usually includes a test harness and test libraries. | Ein Werkzeug, das eine Umgebung zur Testautomatisierung bereitstellt. Es beinhaltet üblicherweise einen Testrahmen und Testbibliotheken. |
| (Definition laut [32] – englisch) | (Definition laut [32] – deutsch) |

Auch in diesem Buch steht der Begriff »Framework« im Zusammenhang mit Testautomatisierung. Allerdings ist der Kontext noch etwas spezifischer. Unter »Keyword-Driven Testing Framework« verstehen wir hier die Gesamtheit der Werkzeuge, Skripte und Bibliotheken, die als Grundlagen für eine bestimmte Variante von Keyword-Driven Testing – meist mit Fokus auf Automatisierung – benötigt und verwendet werden.

Mehr zum Thema Framework, was alles dazugehört und was man davon erwarten kann, findet sich in Abschnitt 4.3 (Sichtweise der Norm ISO 29119-5) und vor allem in Kapitel 6 »Keyword-Driven Testing Frameworks«.

## 1.4 Keywords unter der Lupe

*Elementare Eigenschaften von Keywords*

Um uns Keyword-Driven Testing weiter anzunähern, wollen wir uns in den folgenden Abschnitten Keywords genauer ansehen. Erst einmal geht es um die elementaren Eigenschaften – was ist ein Keyword, was gehört auf jeden Fall dazu? Und dann geht es um einen ganz praktischen, wichtigen Aspekt, nämlich die Struktur, die ein Keyword haben kann.

**Verschlagwortung:** Eines der Grundkonzepte hinter Keyword-Driven Testing ist, dass Tests nicht als Prosatext verfasst werden, sondern in einzelne, klar voneinander abgegrenzte, sehr kurze Testschritte gegliedert werden. Jeder dieser Testschritte soll keine komplette Beschreibung der Tätigkeit sein, sondern nur ein kurzer Befehl, der

die entsprechende Tätigkeit zusammenfasst. Daher rührt der Begriff »Keyword«. Meist reicht es nicht aus, sich auf ein einzelnes Wort als Benennung für ein Keyword zu beschränken. Mehrere Wörter sind also in Ordnung. Als Faustformel für die Länge der Keywords kann man von 1-6 Wörtern ausgehen.

*1 Keyword ≈ 1-6 Wörter*

Kurze Anweisungen von bis zu sechs Wörtern können von einem Leser schnell erfasst und verstanden werden. Oft reicht einer Testerin ein einziger Blick auf eine solche Anweisung, um zu wissen, was zu tun ist.

Einige Beispiele für Keywords:

```
Webbrowser starten
Benutzerverwaltung öffnen
Benutzer an Windows anmelden
Device über WLAN verbinden
Mobilgerät entsperren
```

**Beschreibung:** Keyword-Driven Testing würde für manuelles Testen nicht funktionieren, wenn man sich ausschließlich auf die sprechenden Namen verlassen würde. Bestenfalls könnte man dann erwarten, dass mit den Keywords sehr vertraute Testerinnen sie mühelos und zuverlässig ausführen.

Neue Kolleginnen oder Testerinnen, die die Testspezifikation nur auszuführen haben, aber die Schritte (in Form von Keywords) vorher nicht kennen, könnten nicht wissen, was der Testdesigner von ihnen genau erwartet, und die Tests demnach auch nicht präzise ausführen. Zu der Definition eines Keywords gehört daher obligatorisch auch ein beschreibender Text.

Auch im Hinblick auf Automatisierung ist man sehr gut beraten, eine Dokumentation für das Keyword zu erfassen. Schließlich stellen Keywords hier die Schnittstelle zwischen fachlicher Sicht und technischer Sicht dar. Testautomatisierer müssen genau wissen, was sie implementieren sollen.

Bei der Keyword-Dokumentation geht es nicht nur darum, zu beschreiben, was das Keyword tun soll, sondern auch, wie es zu verwenden ist.

**Parameter:** Eine Interaktion zwischen einer Testerin und dem Testobjekt ist oft von Daten getrieben oder wird anhand von Daten verändert. Daher können Keywords in ihrer Verwendung ebenfalls über Testdaten variiert werden.

Die Benutzeranmeldung an einem System wird beispielsweise immer auf die gleiche Art und Weise durchgeführt. Je nach Benutzerdaten, die für die Anmeldung verwendet werden, unterscheidet sich das Verhalten des Testobjektes.

Diese Daten eines Keywords werden üblicherweise als Parameter oder Argumente bezeichnet. Parameter sind typischerweise im Keyword-Driven Testing dem eigentlichen Keyword angehängt.

Ob dieses Anhängen bzw. Trennen vom Keyword selbst wie im folgenden Beispiel mit zwei oder mehr Leerstellen oder in Tabellen mit mehreren Spalten oder wie in der Programmierung üblich in Klammern erfolgt, ist je nach Tool unterschiedlich und spielt keine Rolle.

Im Folgenden sehen wir ein Beispiel mit einigen Parametern:

```
Webbrowser starten        www.keyword-driven.de
Benutzername eingeben     admin
Passwort eingeben         123456
Anmelden klicken
Benutzerverwaltung öffnen
Device mit WLAN verbinden     Phone2     GuestWifi
Mobilgerät entsperren
```

Mehr zu diesem Thema findet sich in Abschnitt 2.3 »Data-Driven Testing«.

**Keyword-Struktur:** Ein wichtiger Aspekt bei Keyword-Driven Testing ist, dass Keywords »strukturiert« sein können. Wir meinen damit, dass Keywords aus kleineren Keywords zusammengebaut sein können – oder andersherum: Keywords können zu größeren Keywords zusammengefügt werden. Diese Eigenschaft von Keywords macht man sich zunutze, um Wiederverwendbarkeit und Wartbarkeit zu steigern.

Ein Beispiel: Die Anmeldung eines Benutzers am System besteht aus der Eingabe von Benutzername und Passwort und dem Klicken des »Anmelden«-Knopfes. Nehmen wir also an, dass `Benutzer anmelden` ein strukturiertes Keyword ist und aus drei Einzelschritten besteht:

```
Benutzer anmelden     Benutzer     Passwort
    Benutzername eingeben     Benutzer
    Passwort eingeben         Passwort
    Anmelden klicken
```

Ein Testdesigner braucht nun bei einem Test, der die Anmeldung benötigt, nicht immer wieder die drei Einzelschritte zu nutzen, sondern erfordert nur das eine strukturierte Keyword.

Das Thema der Struktur von Keywords greifen wir in Abschnitt 2.2 im Detail auf.

## 1.5 Evolution der Testautomatisierung

In einem Vortrag über Testautomatisierungsarchitekturen im Rahmen der Veranstaltung »Trends in Testing« im Jahr 2010 [13] wurde eine Evolution von Architekturen zur Testautomatisierung beschrieben.

In dieser »Evolution« hat auch Keyword-Driven Testing seinen Platz.

**Vorsicht Glatteis ...**

Keyword-Driven Testing wird von manchen als Königsweg der Testautomatisierung betrachtet.

An dieser Stelle über Testautomatisierungsarchitekturen zu sprechen könnte den Eindruck erwecken, dass es in diesem Buch auch so gesehen wird. Daher zur Klarstellung:

- Keyword-Driven Testing ist wunderbar geeignet für Testautomatisierung, aber:
- Keyword-Driven Testing hat auch viele Vorteile bei manuellem Test!

Eine Beschränkung des Keyword-Driven Testing auf Testautomatisierung wäre Verschwendung.

Abbildung 1-1 bringt zum Ausdruck, wie mit der Entwicklung der Testautomatisierungsarchitekturen, von Capture & Replay über datengetriebenen Test und Keyword-Driven Testing bis hin zu generierten Tests, ein Reifegrad verbunden wird.

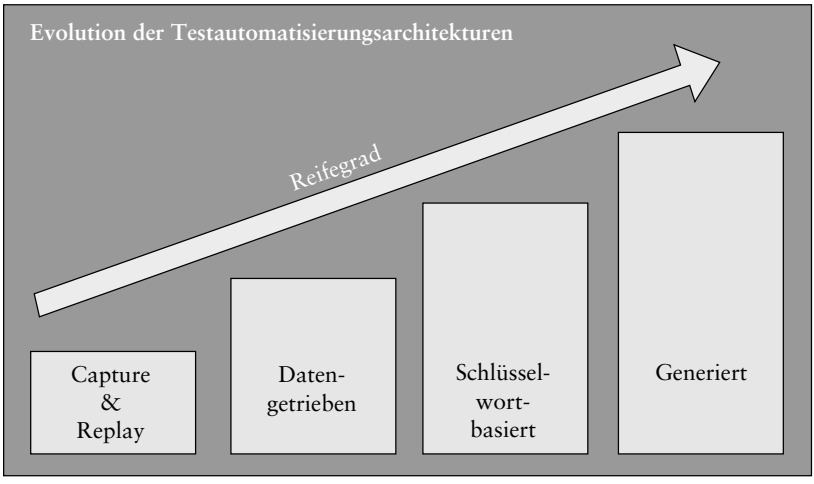



***Abb. 1-1*** *Evolution der Test-automatisierungs-architekturen*

Nach der Einführung von Testautomatisierung in einer Organisation entwickeln sich durch die sukzessive Gewinnung von Erfahrungen der Testerinnen sowohl die Ansprüche an die Umsetzung der Testautomatisierung als auch die Leistungsfähigkeit der Architektur typischerweise in folgenden Phasen – immer vorausgesetzt, externe Einflüsse sorgen nicht dafür, dass diese Phasen direkt übersprungen werden:

*Wartungsproblem*

1. Capture & Replay: Automatisierte Testfälle werden als Skripte aufgezeichnet, dafür wird eine entsprechende »Rekorder«-Funktion des Werkzeugs genutzt. Das Resultat sind automatisierte Testskripte, die einfach strukturiert sind, in der Regel mit der aktuellen Version des Testobjektes funktionieren (aber schon bei kleinen Änderungen am Test Interface Probleme bekommen können), die untereinander redundant sind (gleiche Aktionen werden in jedem Skript erneut erfasst) und, wie sich meist schnell herausstellt, einfach ein Albtraum im Hinblick auf Wartbarkeit darstellen.

   Abbildung 1-2 veranschaulicht einen solchen Fall: Eine Testspezifikation mag vorhanden sein und enthält sowohl Testlogik als auch Daten. Die Verknüpfung zur Testautomatisierung ist nur lose. Ein Testskript liegt individuell für jeden Testfall vor und wird aus einem kaum trennbaren Mix aus Code, Logik und Daten gebildet. Eine Änderung, egal aus welchem Grund, betrifft immer alles, und Fehler passieren leicht.

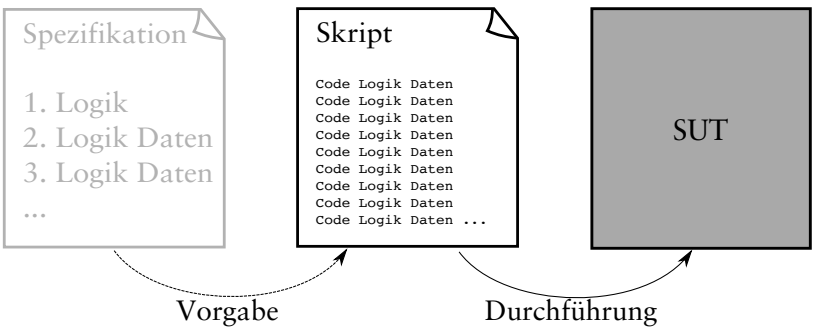



***Abb. 1-2*** *Monolithisches Testskript*

   Also sucht man schnell nach einem besseren Weg.

2. Datengetriebener Test: Während man durch Abkehr vom reinen Aufzeichnen der automatisierten Testskripte schon erste Vorteile in Richtung besserer Wartbarkeit erzielt, kommt die Erkenntnis, dass die Mühe, die man mit der Wartung hat, auch zwischen Testfällen geteilt werden kann. Mehrere strukturell identische Testfälle werden durch ein einziges, verallgemeinertes Skript implementiert und die Unterschiede zwischen den Testfällen, die in den Daten liegen, in

*Trennung von Code und Daten*

von den Testskripten getrennten Datentabellen ausgelagert (mehr dazu in Abschnitt 2.3).

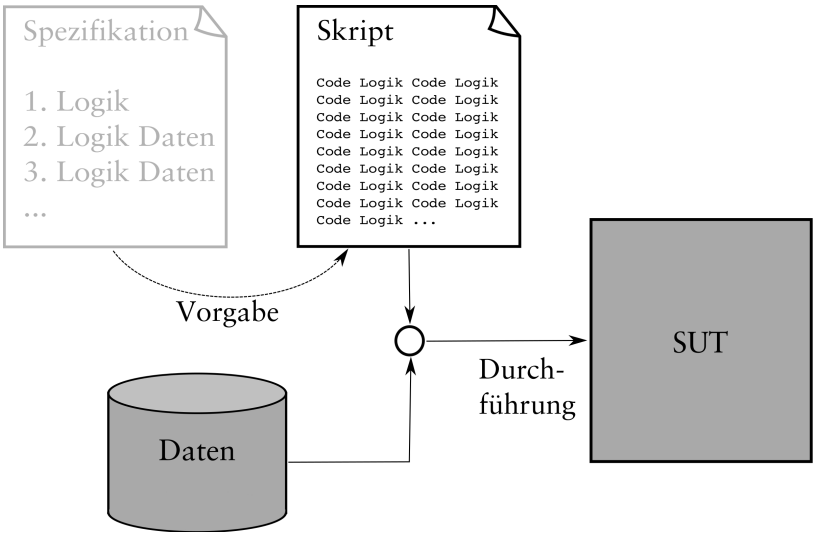



***Abb. 1-3***
*Trennung von Skript und Daten*

In Abbildung 1-3 sind strukturelle Vorteile dieses Ansatzes gegenüber den monolithischen Skripten aus Abbildung 1-2 zu erkennen: Die Kopplung zur Spezifikation ist noch lose (Anmerkung: Daten könnten an dieser Stelle bereits aus der Spezifikation ausgelagert sein), das Chaos in der Testimplementierung ist aber schon geringer, denn es sind nur noch Code und Logik im Skript miteinander vermischt. Änderungen, die nur die Daten betreffen, können unabhängig vom Rest durchgeführt werden, und das kann in vielen Fällen die Änderung sein, die benötigt wird.

Mehr noch: Es muss für eine Anzahl von fünf, zehn oder hundert Testfällen nur noch ein einziges automatisiertes Skript gewartet werden – ein großer Schritt in Richtung eines besseren Kosten-Nutzen-Verhältnisses.

3. Keyword-Driven Testing: Das, was mit datengetriebenem Test begonnen wurde, nämlich die Trennung von Code und Daten, wird beim Keyword-Driven Testing konsequent weitergeführt: Die bisher im Code verankerte Testlogik (der abstrakte Testfall) wird nun getrennt vom (Automatisierungs-)Code abgelegt.

*Trennung von Logik und Code*

Abbildung 1-4 zeigt, dass sich damit massive Änderungen (im Sinne von Verbesserungen) ergeben: Die Spezifikation ist ins Zentrum gerückt und enthält die reine Testlogik. Zur Spezifikation werden Keywords genutzt, denen in einem Repository entsprechende Skripte zugeordnet sind. Hier liegt also der Code. Die drei Aspekte Logik, Code und Daten werden zur Durchführung zusammengeführt.

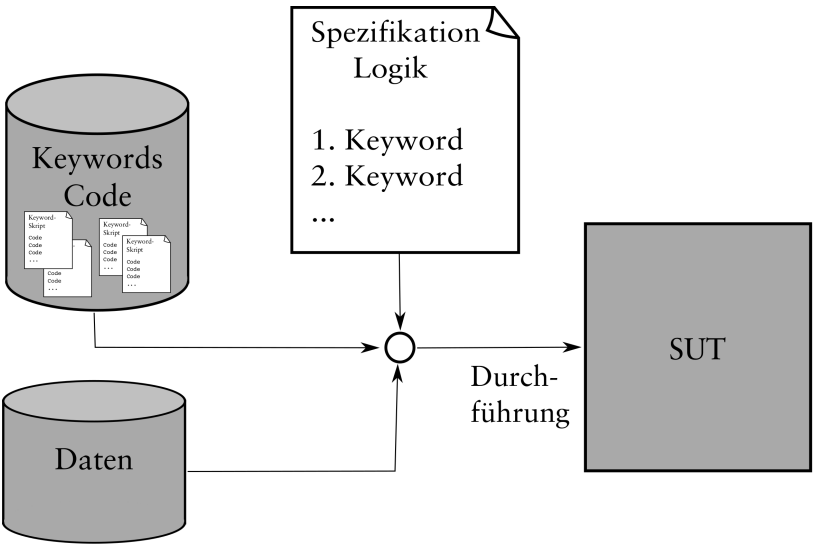



*Abb. 1-4 Trennung von Skript, Daten und Logik*

Jegliche Wartungstätigkeiten können jetzt unabhängig voneinander an Logik, Code und Daten erfolgen. Braucht man eines davon nicht zu ändern, dann muss man es auch nicht anfassen.

> Was hier passiert, die Trennung von Code und Daten, ist ein uraltes Prinzip der Informatik[2] und in der Softwareentwicklung generell ein alter Hut.
>
> Aber zu der Erkenntnis, dass die Entwicklung von Testautomatisierung die gleichen Prinzipien wie Softwareentwicklung erfordert, muss man ja erst gelangen – zunächst sieht alles so einfach aus.

4. Generierte Testfälle: Diese Entwicklungsstufe wird nach heutigem Stand nicht allerorts erreicht und muss auch nicht das Ziel sein. Hier geht es um Model-Based Testing (modellbasierten Test), also darum, Testfälle aus formal beschriebenen Modellen mechanisch abzuleiten. Keywords können dafür eine Grundlage sein.

*Modellbasierter Test*

Was mit diesem Ansatz vorangetrieben wird, ist der Grad der Automatisierung, die nicht mehr nur die Durchführung der Tests,

[2] Und nicht nur dort: In der Rechnerarchitektur ist dieses Prinzip unter dem Begriff »Harvard-Architektur« schon seit dem Mark II bekannt – wenn auch hier eher aus Gründen der Performanz als aus Gründen der Strukturierung und Wiederverwendung.

sondern auch die Erstellung der Testfälle beinhaltet. Die Wartung verlagert sich weg von der Automatisierung und hin zu den Modellen.

Auch wenn Keyword-Driven Testing nach diesem Modell nicht die Spitze der Evolution darstellt – es sieht doch so aus, als wäre im Zuge der Entwicklung von Praktiken zur Testautomatisierung KDT unausweichlich. Aber mehr noch: KDT passt auch ausgezeichnet zu Model-Based Testing; das greifen wir später in Abschnitt 2.6 wieder auf.

## 1.6 Vorteile des Keyword-Driven Testing

Mit dem bisher Gesagten haben wir eine Vorstellung davon vermittelt, was Keyword-Driven Testing ist, aber uns nicht weiter damit beschäftigt, warum es eingesetzt werden sollte. Immerhin liegt es auf der Hand, dass Keyword-Driven Testing mit einem gewissen Aufwand verbunden ist – zumindest müssen Keywords ja definiert und dokumentiert werden. Das Konzept und den damit verbundenen Aufwand können wir nicht alleine mit einem angeblich höheren Reifegrad rechtfertigen.

Im Folgenden werden wir also beleuchten, was die tatsächlichen Vorteile beim Einsatz von Keyword-Driven Testing sind.

### 1.6.1 Klarheit

Das Formulieren von Testfällen aus vorgefertigten Bausteinen führt auf Dauer zu mehr Klarheit, und zwar bei der Interpretation und dem Verständnis dessen, was später bei der Ausführung der Testfälle zu tun ist.

Warum ist das so?

Jede Testerin muss beim Lesen einer Testspezifikation entscheiden, was mit einer bestimmten Formulierung gemeint ist (wir unterstellen, dass sie die Testspezifikation nicht selbst geschrieben hat). Wenn die Testspezifikation herkömmlich, also in Prosa verfasst ist, so sind die Formulierungen nicht einheitlich. Derselbe Sachverhalt kann mit verschiedenen Worten beschrieben sein. Ein Beispiel: Es wird einmal der Begriff »Anmelden« und das nächste Mal der Begriff »Einloggen« verwendet. Durchführende Testerinnen müssen an dieser Stelle entscheiden, ob diese Begriffe dasselbe bedeuten - wenn ja, was, und wenn nein: Was ist der Unterschied?

*Keywords verhindern Ambivalenz.*

Dieselbe Interpretationsleistung muss auch bei Verwendung von Keywords erbracht werden, aber eben nur einmal. Durch die Vereinheitlichung der Begriffe und das Bausteinprinzip gibt es nur ein Keyword für diese Aktivität. Am Anfang muss man einmal verstehen, was damit

*Lerneffekt*

gemeint ist – aber hier findet ein Lernprozess statt und bald kennt man die Bedeutung.

Dieses Beispiel für Vorteile des Keyword-Driven Testing greift insbesondere beim manuellen Test. Also Schluss mit dem Mythos, dass Keyword-Driven Testing nur ein Thema für Testautomatisierer ist!

Klar ist: Das hier beschriebene Mehr an Klarheit wird insbesondere dann seinen Nutzen entfalten, wenn in Teams gearbeitet wird. Aber eine verbesserte Lesbarkeit ist mit Sicherheit ein Gewinn.

### 1.6.2 Wiederverwendbarkeit

Mehr oder weniger selbsterklärend ist die Tatsache, dass Keywords wiederverwendet werden können. Das ist natürlich bei fast allen Testautomaten in Bezug auf die automatisierten Testschritte der Fall. Jedoch können wir diese Funktionalität bei kaum einem Werkzeug für manuelles Testen oder Testmanagement finden. Hier wird in aller Regel mit rein textuellen nicht wiederverwendbaren Testschritten gearbeitet.

Die Wiederverwendbarkeit steht in direktem Zusammenhang mit dem nächsten Abschnitt über Wartbarkeit. Aber es geht nicht nur darum, sondern auch um den ersparten Aufwand bei der Erstellung von Tests und um den Vorteil bezüglich der Klarheit und Lesbarkeit.

*Aufwand sparen*

Ein einmalig erstelltes Keyword, das beschreibt oder implementiert, wie unser Testobjekt zu starten ist, spart bei jeder Nutzung Zeit, verglichen mit einem immer wieder erneuten Beschreiben dieser Tätigkeit.

Wiederverwendbarkeit reduziert repetitive Arbeiten also auf ein Minimum und beschleunigt das Arbeiten entsprechend.

### 1.6.3 Wartbarkeit

Eine verbesserte Wartbarkeit ist wahrscheinlich der am häufigsten beschworene Vorteil von Keyword-Driven Testing.

Er ergibt sich auch bei manuellem Test, ist aber besonders im Zusammenhang mit Testautomatisierung wichtig, da der Nutzen von Testautomatisierung[3] von hohen Wartungsaufwänden zunichtegemacht werden kann.

Wartbarkeit ist ein Qualitätsmerkmal, das beschreibt, mit wie kleinem oder großem Aufwand Änderungen durchgeführt werden können. Ob Änderungen an unseren Testfällen notwendig sind, können wir uns

*Auslöser für Wartung*

[3] Der wirtschaftliche Nutzen wird dadurch festgestellt, dass die Kostenersparnis auf Dauer die Erstellungs- und Wartungskosten aufwiegt. Es gibt noch andere Nutzenaspekte, wie Motivation, Geschwindigkeit etc.

nicht aussuchen. Die Hoffnung sagt: Es wird schon nicht so schlimm. Die Erfahrung zeigt: Änderungen werden unweigerlich kommen.

Die Änderungen können sich, je nach Ursache, auf verschiedene Aspekte beziehen:

**Abstrakter Testfall:** Eine Änderung der Testlogik ist dann nötig, wenn der Testfall falsch war oder sich die zugrunde liegende Anforderung geändert hat. In diesem Fall muss inhaltlich eine Änderung an der Testlogik vorgenommen werden.

**Konkreter Testfall:** Eine Änderung der Testdaten kann aus den gleichen Gründen notwendig sein, zusätzlich aber auch dann, wenn bei gleichem abstraktem Testfall ein weiterer konkreter Testfall hinzugefügt werden soll oder eine Änderung der Testumgebung geänderte Testdaten erfordert.

**Funktionale Veränderung:** Eine Änderung oder das Hinzufügen einer Funktionalität kann bedeuten, dass ein bestehender Test im fachlichen Ablauf identisch ist und von der Testlogik her unverändert bleiben kann, obgleich die Benutzerführung sich ändert.

Ein Beispiel: Stellen wir uns vor, als Neuerung würde nun bei Kundenanlage auf eine Datenschutzerklärung hingewiesen werden. Ein vorhandener abstrakter Testfall kann gleich bleiben, nur das Keyword `Neukunde anlegen` wird technisch etwas angepasst. Anmerkung: Wir sprechen hierbei nicht von dem vermutlich benötigten neuen Testfall, der die neue Datenschutzerklärung prüft!

**Technologische Veränderung:** Eine Änderung der Implementierung wird insbesondere dann erforderlich, wenn sich die Testschnittstelle ändert. Im Fall einer Web-GUI (Graphical User Interface) könnte ein spezifisches Element an einem anderen Ort liegen und aufgrund dessen einen anderen Selektor benötigen, um das Element eindeutig zu identifizieren. Dazu könnte ein einzelnes technisches Keyword verändert werden und alle fachlichen Keywords, die es verwenden, und die Tests, die dieses nutzen, wären wieder lauffähig.

Durch eine konsequente Trennung von Testlogik (Sequenz der Keywords) und Testdaten sowie eine Trennung von Spezifikation und Implementierung in einem Automatisierungswerkzeug können diese Änderungen unabhängig voneinander vorgenommen werden. Das macht die Änderungen überschaubar und damit weniger anfällig dafür, Seiteneffekte und Fehler auszulösen.

Umgekehrt muss im Fall einer monolithischen Implementierung von automatisierten Testfällen ohne Keywords und ohne Trennung von Daten und Logik (Worst Case) bei *jeder* Änderung, egal ob diese die Testlogik, Daten oder Implementierung betrifft, mehrere Testskripte

angepasst werden. Somit würde der Wartungsaufwand direkt proportional mit der Menge von Testfällen skalieren. Dies führt, wie in Abschnitt 3.8 noch gezeigt wird, zu dem Fall, dass ab einem gewissen Wartungsaufwand, beispielsweise einem Technologiewechsel, das Verwerfen der gesamten Tests und eine komplette Neuimplementierung günstiger ist als die Anpassung der Skripte.

Diese verbesserte Wartbarkeit bei Keyword-Driven Testing sorgt also dafür, dass Änderungen und der Wartungsaufwand nicht mit der Anzahl der Tests, sondern lediglich mit der (viel geringeren) Anzahl der implementierten Keywords oder Funktionen steigt.

Häufig ist der steigende Wartungsaufwand die natürliche Grenze dafür, wie viel automatisiert werden kann. Werden die zuständigen Schichten nicht gut designet, ist das Team ab einer gewissen Menge Tests fast vollständig mit Änderungen vorhandener Tests beschäftigt und hat keine Zeit mehr, neue Tests zu schreiben.

### 1.6.4 Kommunikation

Die Verwendung von Keywords zwingt zu einem gewissen Maß an geordneter Kommunikation, wenn andere Vorteile des Keyword-Driven Testing – insbesondere alles, was mit der Wiederverwendbarkeit der Keywords zusammenhängt – erreicht werden sollen. Hier ist es, wie in vielen anderen Bereichen: Klare Zuständigkeiten und Kommunikationsschnittstellen zwingen zu Disziplin. Nichteinhaltung der Kommunikationsschnittstellen andererseits führt zum Scheitern.

Ohne diese klaren Zuständigkeiten könnte im Test jede einzelne Testerin, überspitzt gesagt, tun, was sie will. Ganz für sich alleine und ohne wesentliche Kommunikation mit den anderen Kollegen entsteht Chaos. Das ist sicher einem effizienten Arbeiten nicht zuträglich. Gerade die Wiederverwendbarkeit würde immens leiden unter einem schlecht funktionierenden Team und fehlender Abstimmung.

Abgesehen davon, dass Keyword-Driven Testing alleine schon durch eine Abstimmung über die Keywords das Team zur Kommunikation zwingt und so die Zusammenarbeit fördert, erleichtert es die Kommunikation aber auch.

Nun könnte man argumentieren, dass das ja gerade kein Vorteil, sondern ein Nachteil an Keyword-Driven Testing sei. Es benötigt eine gute Kommunikation zwischen den Beteiligten, sonst scheitert es.

In gewisser Weise müssen wir hier zustimmen, aber: Wenn wir uns auf gleiche Formulierungen einigen, wenn wir von denselben Dingen in derselben Sprache sprechen und wenn alle Prozessbeteiligten sowohl die Tests als auch die Keywords lesen und verstehen können, wächst

automatisch das gemeinsame Verständnis und das Vertrauen bezogen auf den Testprozess.

Und dieser Vorteil ist in der Praxis so enorm wichtig, dass mittelfristig die Beteiligten diese gehobenen Anforderungen keineswegs mehr als Last empfinden.

Die bereits in Abschnitt 1.6.1 beschriebene verbesserte Klarheit der Testfälle wirkt sich nicht nur im Hinblick auf eine beschleunigte und fehlerärmere Ausführung der Testfälle aus, sondern hilft auch bei der Kommunikation über Testfälle, beispielsweise mit Kunden, die ansonsten in der Regel kein Verständnis der Testautomatisierung gewinnen würden. Darüber hinaus erleichtern die wohldefinierten Keywords, die ja auch Schnittstellen zwischen verschiedenen fachlichen Ebenen darstellen, die Verständigung zwischen unterschiedlich spezialisierten Teammitgliedern bei der Arbeitsteiligkeit.

### 1.6.5 Arbeitsteiligkeit

Insbesondere in Projekten[4] mit einer hohen fachlichen Komplexität und einer gewissen Teamgröße gibt es häufig eine gewisse Spezialisierung oder Rollen (in alphabetischer Reihenfolge):

*Rollen*

**Anwender oder Fachexperte, Domain-Experte:** Personen, die einen tiefen Einblick in die fachlichen Zusammenhänge haben und daher die Testfälle aus logischer Sicht hervorragend gestalten können, möglicherweise aber nicht darin ausgebildet sind, in die technischen Untiefen eines Testautomatisierungswerkzeugs einzutauchen.

**Entwicklerin:** Implementiert die eigentliche Anwendung. Es gibt hier häufig weitere Spezialisierungen, etwa nach den Themen Frontend, Backend oder Datenbank.

**Testautomatisierer:** Personen, die Testautomatisierung durch und durch beherrschen, noch vor dem Frühstück Code schreiben, aber die fachlichen Zusammenhänge des Testobjektes ggf. nur sehr bedingt verstehen.

**Testdesigner:** Gestaltet die Testfälle basierend auf den Informationen, die von Fachexperten bereitgestellt werden.

[4] Hier und an anderen Stellen im Buch wird von »Projekt« gesprochen. Das geschieht der Einfachheit und der besseren Lesbarkeit halber, aber:

Es geht hier nicht nur um Tests in Projekten, sondern auch um Tests von Produkten, in Verfahren oder bei der Wartung von Bestandssystemen. Aus Testsicht ist jedes Vorhaben, in dem Software oder Systeme getestet werden wollen, ein »Testprojekt«.

**Testerin:** Eine Person, die Testfälle manuell durchführt, aber auch Testdaten vorbereitet oder Auswertungen vornimmt.

**Testmanagerin:** Zuständig für die Organisation des Testens, Bestimmung der Ziele und der Strategie und somit auch maßgeblich daran beteiligt, eine Entscheidung für die Verwendung von Keyword-Driven Testing herbeizuführen.

Über Rollen im Test ist schon viel geschrieben worden. Widerstehen wir der Versuchung, das hier auch zu tun. Wer aber mehr dazu lesen will, findet weitere Informationen unter anderem in [10] unter »Kompetenzen und Teamzusammensetzung« oder auch in den Lehrplänen des ISTQB®[30].

Einige dieser Rollen können von derselben Person eingenommen werden, es ist aber untypisch, dass dies für Gegenpole wie »Fachexperte« und »Testautomatisierer« zutrifft.

Sicher: Idealerweise wünschen wir uns die legendäre eierlegende Wollmilchsau – also Heldinnen, die einfach alles können. Solche mag es auch geben. Aber leider nie genug. Realistisch betrachtet, kann sich glücklich schätzen, wer ein Team hat, in dem alle erforderlichen Aspekte kompetent abgedeckt werden. Diese unterschiedlichen Ausprägungen der Teammitglieder benötigen wir, und wir sollten keinesfalls eines über das andere stellen. Der beste Testautomatisierer kann durchaus völlig unbelastet von jeglicher Vorkenntnis sein, bezogen auf den Einsatz des Testobjektes. Anders gesagt, er hat absolut kein Konzept davon, was das System tut, für das er automatisieren soll. Das kann funktionieren.

Wichtig ist, dass wir nicht nur Expertinnen haben, die verstehen, wie entwickelt wird, sondern auch hervorragend ausgebildete Fachexperten, die wissen, was entwickelt werden sollte.

*Spezialisten zusammenbringen*

Und hier zeigt sich eine Stärke des Keyword-Driven Testing: Mit Keywords kann man eine Arbeitsteilung zwischen fachlicher Seite und technischer Seite, zwischen Fachexperte und Testautomatisierer erreichen. Während Fachexperten die Testfälle aus Keywords zusammensetzen, können sie die Details der technischen Umsetzung in einem Automatisierungswerkzeug ignorieren. Umgekehrt können sich die Testautomatisierer auf die technische Umsetzung fokussieren und die intimen Kenntnisse über das Automatisierungswerkzeug voll ausspielen. Dabei sind die genauen fachlichen Zusammenhänge für sie nicht wichtig.

Ohne Keyword-Driven Testing ist eine solche Arbeitsteilung, wenn überhaupt, nur wesentlich schwieriger und weniger elegant zu erreichen.

### 1.6.6 Vereinfachte Testautomatisierung

Testautomatisierung ist zwar nicht verpflichtend, aber in vielen Bereichen nicht mehr wegzudenken. Häufig sind Regressionstests ohne Testautomatisierung personell und zeitlich nicht mehr vorstellbar. Das gilt nicht nur, aber ganz besonders im agilen Umfeld, da hier die kurzen Entwicklungszyklen keinen Spielraum für zeitlich ausgedehnte manuelle Testphasen lassen.

Mit Keyword-Driven Testing lässt sich Testautomatisierung sehr effizient umsetzen. Das passende Framework vorausgesetzt, reicht es, für jedes Keyword eine Implementierung in der gewünschten Automatisierungsumgebung bereitzustellen.

Abstrakte Tests werden aus Keywords zusammengesetzt und konkrete Testfälle werden mithilfe von konkreten Daten spezifiziert.

Das Framework übernimmt dann die Ausführung der Automatisierungssequenz basierend auf dem aus Keywords bestehenden Testfall und den zugeordneten Automatisierungsskripten bzw. -funktionen.

Vorteil: Es wird nicht mehr für jeden Testfall eine Implementierung benötigt, sondern nur für jedes Keyword. Das reduziert die Menge des benötigten Automatisierungscodes erheblich. Bei angenommen 1000 Testfällen, bestehend aus 100 Keywords, werden ohne Keyword-Driven Testing 1000 Skripte benötigt[5], mit Keyword-Driven Testing nur 100 automatisierte Funktionen alias Keywords. Kommt man mit weniger Keywords aus, wird das Verhältnis noch günstiger.

*Wenige Keywords reichen aus.*

Im Idealfall lassen sich ab einem gewissen Zeitpunkt Testfälle zusammenstellen, die sofort und ohne weiteres Zutun schon fix und fertig automatisiert sind, da alle benutzten Keywords bereits umgesetzt wurden.

Die bereits genannten Vorteile wie Wartbarkeit und Wiederverwendbarkeit können und sollten natürlich auch mit normalen Programmiersprachen wie JavaScript, Java, C# oder Python erreicht werden. Die Vorteile hinsichtlich Klarheit, Kommunikation und Arbeitsteilung sind jedoch mit der komplexen Syntax solcher Programmiersprachen kaum zu erreichen. Das doch zu schaffen, ist das Ziel von Methoden wie Behavior-Driven Testing (siehe Abschnitt 8.3). Selbst bei der strengen Einhaltung von Clean-Code-Prinzipien und der Optimierung des Testcodes auf Lesbarkeit liegen Welten zwischen automatisierten Tests mit Keyword-Driven Testing und in einer Programmiersprache geschriebenen Tests. Alleine die notwendigen Klammern und andere Steuerzeichen bauen für manche Menschen zu starke Barrieren auf ...

[5] Zur Vereinfachung wird der Aspekt der Testdaten hier ausgeblendet.

Schicken Sie zu diesem Zeitpunkt aber bitte nicht Ihren fähigen Testautomatisierer nach Hause! Auch wenn die Implementierungsaufwände sinken, werden Änderungen an der Implementierung und neue Keywords diese Spezialisten immer benötigen. Zusätzlich werden aufgrund der gesteigerten Geschwindigkeit bei der Testfallerstellung auch entsprechende Zuarbeiten durch die Testautomatisierer notwendig.

Vor allem schafft ein Vorgehen nach Keyword-Driven Testing und die Entlastung der Automatisierer Freiraum, damit sich diese um die Bereitstellung und Betreuung der Testumgebungen und der Test-Pipelines kümmern können.

Angesichts einer klaren Rollentrennung zwischen »Keyword-Implementierenden« und »Test-Spezifizierenden« ist es ein konsequenter Schritt, wenn die Entwicklerinnen, die ohnehin für die Implementierung des Testobjektes zuständig sind, auch die Aufgabe der Implementierung für die Keywords auf technischer Ebene übernehmen.

### 1.6.7 Geschwindigkeit

Zugegeben, Keyword-Driven Testing erfordert in der Startphase einen Zusatzaufwand. Initial wird eine saubere Analyse benötigt. Darauf basierend muss ein Layer-Konzept (siehe Abschnitt 2.2.2) erarbeitet werden und es müssen Keywords definiert und beschrieben werden.

Diese initialen Aufwände werden aber auf Dauer amortisiert. Die Vorarbeit macht sich bezahlt: Sowohl das Spezifizieren geht schneller, wenn man auf fertige Bausteine zurückgreifen kann, als auch später die Ausführung der Tests.

Alle zuvor beschriebenen Vorteile erhöhen die Qualität unserer Tests und Testarchitektur und führen schlussendlich zu einer höheren Geschwindigkeit. Es sei jedoch gesagt, dass das Primärziel der höheren Geschwindigkeit unserer Erfahrung nach noch nie zu einer langfristigen Verbesserung geführt hat. Erhöhung der Qualität sowohl im Testobjekt als auch in der Testspezifikation und -architektur bringt andererseits aber langfristig anhaltend mehr Geschwindigkeit als Nebeneffekt.

## 1.7 Werkzeuge für Keyword-Driven Testing

Welche Tools verwenden wir nun, wenn wir Keyword-Driven Testing einsetzen wollen? Und können wir uns auf die Aussage »unterstützt Keyword-Driven Testing« verlassen?

Leider eher nein – die Werbeaussagen sind oft irreführend. Im besten Fall können diese Werkzeuge als ein Teil eines Frameworks genutzt werden, vollständige Unterstützung bietet keins. Das ist erst einmal auch völlig in Ordnung. Manche Werkzeuge halten aber – vielleicht

aufgrund eines sehr stark vereinfachten Verständnisses von KDT – bei Weitem nicht, was sie versprechen. Machen Sie sich bitte Ihr eigenes Bild (vgl. Kapitel 6).

Wie zuvor in der Begriffserklärung beschrieben, wird ein Framework für Keyword-Driven Testing also aus unterschiedlichen Werkzeugen gebildet. Welche Werkzeugarten Ihnen in diesem Zusammenhang begegnen mögen und was Sie benötigen, möchten wir nachfolgend kurz erläutern.

### 1.7.1 Testmanagementsysteme

Testmanagementsysteme spielen häufig (leider) eine untergeordnete Rolle im Keyword-Driven Testing, da die meisten nicht darauf ausgerichtet sind, aktiv Keyword-Driven Testing zu unterstützen.

Solche Testmanagementsysteme fokussieren sich »nur« auf die Aufgabe, die manuellen Tests zu verwalten und deren Spezifikation und Ergebnisse zu beherbergen; einige werden darüber hinaus dazu verwendet, Ergebnisse externer automatisierter Tests zu archivieren. Die Testspezifikationen der automatisierten Tests sucht man in der Regel vergebens in diesen Werkzeugen.

Um das für Keyword-Driven Testing leisten zu können, müssten die Werkzeuge in der Lage sein, mit wiederverwendbaren Keywords zu arbeiten und die Testdesigner bei modularer und idealerweise datengetriebener Spezifikation der Testfälle zu unterstützen. Geht das nicht, dann können also weder manuelle noch automatisierte Testfälle aus Keywords aufgebaut werden.

Es gibt jedoch wenige Ausnahmen – Testmanagementsysteme, die entweder die Möglichkeit haben, fertig implementierte Keywords aus einem Testautomaten zu importieren, oder mit denen Testdesigner in der Lage sind, neue Keywords zu entwerfen und diese auch im Testmanagementsystem zu strukturieren.

Genau hinzusehen lohnt sich: Einige Produkte auf dem Markt unterstützen zwar die Wiederverwendung einzelner Testschritte, aber nicht Keywords, wie wir sie verstehen.

### 1.7.2 Full-Stack-Testautomaten

Als Full-Stack-Testautomaten bezeichnen wir eine spezifische Gruppe von Testautomaten. Der Begriff »Full-Stack« ist ein Kunstwort, mit dem wir ausdrücken wollen, dass die Werkzeuge dieser Kategorie alle Komponenten, die zu Automatisierung eines Testobjektes notwendig sind, selbst enthalten.

Dadurch grenzen sie sich von den Automatisierungstechnologien und Automatisierungsframeworks ab, die zusätzliche Komponenten zum Betrieb benötigen. Diese Komponenten sind ganz grob gesprochen der Testeditor, die Ausführungs-Engine und die Adaptierungs- oder Automatisierungstechnologie, die das Testobjekt stimuliert und dessen Reaktionen entgegennimmt. Mehr zu diesen Komponenten und den entsprechenden Aufgaben findet sich später in Kapitel 5 »Testautomatisierungsarchitektur«.

Klassische Beispiele für Full-Stack-Automaten sind UFT, Test Complete, SoapUI oder Tools wie QF-Test, Tricentis Tosca, Ranorex oder Eggplant.

All diese Werkzeuge haben eigene Editoren, die teilweise per Drag & Drop oder per Scripting Tests erstellen können. Jedes Testautomatisierungswerkzeug ist dabei auch in der Lage, wiederverwendbare Testschritte zu erstellen und in Tests zu verwenden.

Obwohl einige dieser Automaten für sich in Anspruch nehmen, Keyword-Driven Testing zu unterstützen, möchten wir bei den wenigsten Automaten von Keyword-Driven Testing sprechen. Gerade was die Lesbarkeit, Einfachheit und Strukturierung von Keywords und Tests angeht, setzen diese Tools die Konzepte von Keyword-Driven Testing unserer Auffassung nach nicht zufriedenstellend um.

### 1.7.3 Testautomatisierungsframeworks

In Abgrenzung zu einem Keyword-Driven Testing Framework, das die gesamte konkrete Werkzeugkette vom Editor über die Testausführungs-Engine bis zur eigentlichen Automatisierungstechnologie definiert, kümmern sich die hier genannten Werkzeuge hauptsächlich darum, Testen in einer spezifischen Programmiersprache umzusetzen.

Sie definieren dazu eine Spezifikationssyntax, ohne selbst einen Editor mitzuliefern, und geben an, wie man eine Technologie, wie beispielsweise Selenium, adaptiert, um daraus eine vollständige Automatisierungslösung zu bauen. Diese »unterste« Ebene des Automaten ist entsprechend noch zusätzlich zu implementieren oder zu adaptieren (siehe hierzu das Kapitel 5 »Testautomatisierungsarchitektur«).

Ein (Test-)Automatisierungsframework implementiert im Gegensatz zum Full-Stack-Automaten also nicht alle Komponenten eines fertigen Testautomaten, sondern konzentriert sich hauptsächlich auf die Spezifikationssprache, Ausführung von Tests und Dokumentation von Testergebnissen.

Die gängigsten Testautomatisierungsframeworks lassen sich in eine von zwei Kategorien packen:

**Unit-Testing Frameworks:** Frameworks dieser Kategorie, wie JUnit (Java), Pytest (Python), NUnit (C#), Google Test (C++) oder Mocha (JavaScript), verwenden alle genau die Programmiersprache zur Testspezifikation, in der sie selbst und das zu testende Testobjekt – der zu testende Code – implementiert sind.

**Behaviour-Driven Development Frameworks:** Diese Gruppe von Frameworks bieten dem Anwender die Möglichkeit, Tests als menschenlesbaren Text zu spezifizieren, obwohl der eigentliche Testcode in kompilierter Form vorliegt. In der Regel werden solche Tests in Gherkin-Syntax (Given/When/Then) geschrieben (siehe Abschnitt 8.3 »Behavior-Driven Testing«).

Wie steht es denn nun um Frameworks für Keyword-Driven Testing? Da Keyword-Driven Testing seine Stärken vor allem auf höheren Teststufen zeigt und Entwicklerinnen auf diesen Teststufen selten unterwegs sind, ist die Toolauswahl hier ziemlich begrenzt.

Es gbt aber ein Open Source Framework, das sowohl Behavior-Driven Testing als auch Keyword-Driven Testing hervorragend umsetzt und deswegen auch in diesem Buch besonders beleuchtet werden soll. Robot Framework ist explizit als Keyword-Driven-Automatisierungsframework entwickelt worden und kann somit viele Stärken von Keyword-Driven Testing nutzen. (Robot Framework wird in Kapitel 7 »Praxis mit Robot Framework« detailliert vorgestellt und besprochen.)

### 1.7.4 Testdesignwerkzeuge und Editoren

Ein Großteil von Keyword-Driven Testing spielt sich im Editor bzw. in der Benutzeroberfläche des Testautomaten ab. Leider finden wir auch gerade hier die meisten Unzulänglichkeiten.

Diese Werkzeugkategorie ist als eigenständiges Werkzeug eigentlich nur in Verbindung mit Automatisierungsframeworks relevant. Testmanagementsysteme steuern ja genau diesen Teil als eine Komponente zum Keyword-Driven Testing Framework bei. Full-Stack-Automaten haben ebenfalls einen eigenen Editor, auch wenn die Unterstützung von Keyword-Driven Testing zu wünschen übrig lässt.

Für die meisten Frameworks werden Entwicklungsumgebungen, wie Visual Studio Code, IntelliJ IDEA oder Eclipse, mit entsprechenden Erweiterungen verwendet. Ein Nachteil gegenüber dedizierten Keyword-Driven-Testing-Editoren sind die vielen für die Softwareentwicklung erstellten, jedoch für Keyword-Driven Testing unnötigen

GUI-Komponenten, die solche Tools oft überladen aussehen lassen und Nutzerinnen abschrecken.

Es ist nicht einfach, die richtigen Komponenten zu wählen, und eine »Silver Bullet«[6] gibt es nicht. Wir hoffen aber, in diesem Buch genügend handfeste Entscheidungshilfen zu liefern, damit Sie sich Ihren eigenen Werkzeugkoffer so zusammenstellen, dass es ein erfolgreiches Framework wird.

## 1.8 Beispiele in diesem Buch

Im Laufe des Buches werden wir zur Erläuterung auf Beispiele zurückgreifen. Manche sind sehr spezifisch; wo es aber möglich ist, werden wir ein einfaches, realistisches und für jeden nachvollziehbares Beispiel verwenden, das nicht von dem ablenkt, was erklärt werden soll. Wir haben uns gegen den ewigen Geldautomaten entschieden und nehmen statt dessen etwas anderes, von dem wir sicher sind, dass jede und jeder es kennt: einen Onlineshop.

Weil das Beispiel fiktiv ist, brauchen wir uns nicht zurückzuhalten, sondern können jederzeit Testfälle aus diesem Projekt zitieren. Es dient also als »fachliche« Referenz.

Mit echten Praxisbeispielen wäre das schwieriger, weil jeglicher Inhalt daraus natürlich vertraulich ist. In anonymisierter Form und ohne dass die betreffenden Organisationen erkennbar sind (außer vielleicht für Menschen, die damit ohnehin selbst arbeiten), können wir aber zwei Beispiele zu real in der Praxis eingesetzten Frameworks skizzieren. Das tun wir in Abschnitt 6.4.

Zurück zu unserem fachlichen Beispiel – geben wir ihm ein Gesicht:

In dem Projekt geht es also um einen Onlineshop. Stellen Sie sich vor, es werden Gegenstände verschiedener Künstler und Kunsthandwerker angeboten. Bei den Waren handelt es sich vielleicht um Drechsel- oder Töpferarbeiten, Aquarelle und Ölbilder, Mode eines kleinen Modelabels oder aber auch um handgestrickte Pullis aus Schafwolle.

Der Shop ähnelt vielen gängigen Shops. Es gibt ein Suchfeld, einen Warenkorb, verschiedene Kategorien auf der linken Seite, Bewertungsmöglichkeiten der User zu den einzelnen Produkten sowie einen persönlichen Login-Bereich.

Es gibt eine Desktop-Version für alle gängigen Browser sowie native Apps für iOS und Android. Aktuell plant das Management, für den Shop eine von den verschiedenen Plattformen unabhängige Web-App

[6] Eine »Silver Bullet« (Silberkugel) ist eine der wenigen Waffen die gegen Hexen und Werwölfe hilft und metaphorisch als eine einfache und fast magische Lösung für ein schwieriges Problem genutzt wird.

zu entwickeln. Diese Web-App soll die gleichen Funktionalitäten bieten wie die nativen Apps. Das Management verspricht sich davon eine Kosteneinsparung bei der Wartung und Pflege der einzelnen Apps.

Der Shop ist weltweit zu erreichen und die verkauften Waren werden in alle sechs Kontinente ausgeliefert.

Unser fiktiver Shop ist bereits längere Zeit auf dem Markt, sodass es viele User gibt, hoher Traffic vorhanden ist und falls etwas nicht funktioniert, werden wir schnell ein Problem haben.

Wir denken, ein sorgfältiger Test unseres Shops ist angemessen und die Herausforderungen passen zu Keyword-Driven Testing.

## 1.9 Ressourcen

Einige Beispiele, mit denen Sie selbst die im Buch, insbesondere in Kapitel 7 »Praxis mit Robot Framework«, vorgestellten Konzepte praktisch nachvollziehen können, haben wir für Sie zum Download bereitgestellt. Den Link hierzu haben wir im Literaturverzeichnis als [37] hinterlegt. Ganz einfach erreichen Sie die Seite aber mit diesem QR-Code:

***Abb. 1-5*** *Link zu Ressourcen*

# 6 Keyword-Driven Testing Frameworks

Stand heute ist kein einzelnes Werkzeug mit allen Funktionen ausgestattet, um Keyword-Driven Testing vollständig umzusetzen. Das ist an sich kein Problem – es ist schlicht der Grund, warum wir von »Frameworks« sprechen.

In der Praxis werden also mehrere Werkzeuge zu einem Keyword-Driven Testing Framework zusammengesetzt. Die notwendigen Komponenten haben wir bereits in Kapitel 4 »Keywords und Normen« besprochen.

Werkzeuge, aus denen wir unser Keyword-Driven Testing Framework zusammenstellen, sollten entsprechend der Norm alle Anforderungen abdecken, die an ihren Teilbereich gestellt wurden.

Im Erstellungsprozess des Frameworks und auch in diesem Kapitel besteht unser erster Schritt darin, die Anforderungen zu definieren, die auf jeden Fall von unseren Komponenten erfüllt werden sollen.

Als Nächstes stellen wir uns der Frage, ob wir diese Komponenten selbst entwickeln sollten, uns Open-Source-Komponenten beschaffen oder ob wir nicht doch lieber zu einer kostenpflichtigen Komponente greifen.

Dann werden wir uns kurz drei Bausteine – zwei kommerzielle, einer Open Source – ansehen, die Bestandteile eines Frameworks sein können.

Zuletzt sehen wir uns exemplarisch zwei konkrete Szenarien an und zeigen eine mögliche Framework-Bewertung dieser Lösungen anhand der ISO 29119.

## 6.1 Anforderungen an ein Framework

Was darf man denn von einem Framework für Keyword-Driven Testing erwarten – oder andersherum: Was muss ein solches Framework denn bieten, damit wir damit zufrieden sind?

Bei der Beantwortung dieser Frage orientieren wir uns hier nicht an dem, was die ISO 29119-5 dazu schreibt – wenn Sie den Inhalt dieses Abschnitts gegen die Norm halten, werden Sie Parallelen finden, aber auch Abweichungen. Denn es ist klar: Was jemand wichtig findet, ist subjektiv und situationsabhängig. Wir Autoren haben unsere Erfahrungen mit Keyword-Driven Testing und unterschiedlichen Frameworks dazu, und darauf basieren die folgenden grundsätzlichen Anforderungen, die wir stellen. In einem konkreten Projektkontext, für Sie als Leserin, können ganz andere Anforderungen wichtig sein.

Dies ist unser Beispiel für Minimalanforderungen an ein Framework für Keyword-Driven Testing:

**Editor:** Ein eigener Editor, der Keyword-Driven Testing explizit unterstützt und möglichst auch Fehler vermeiden hilft, ist vorhanden.

**Data-Driven Testing:** Das Framework muss auf jeden Fall Data-Driven Testing unterstützen.

**Manuell/automatisiert:** Sowohl das automatisierte als auch das manuelle Durchführen desselben Testfalls muss möglich sein.

**Hierarchische Keywords:** Für manuelles Testen mag man mit einer Ebene von Keywords auskommen, für Testautomatisierung hingegen sind mehrere Ebenen von Keywords erforderlich.

**Verwaltung:** Unterstützung bei der Verwaltung von Keywords, das Speichern wichtiger Attribute und Auffinden eines gewünschten Keywords gehören dazu.

**Arbeitsteiligkeit:** Arbeit mit mehreren Teammitgliedern an demselben Projekt muss gewährleistet sein. Ein Testdesigner muss mit Keywords in einem Test arbeiten können, während ein Teammitglied sich mit dem Nachbartest beschäftigt oder ein Automatisierer Anpassungen an den Keywords vornimmt.

Aus jeder dieser grundsätzlichen Anforderungen können weitere detaillierte Anforderungen abgeleitet werden. Mit den Checklisten aus Abschnitt 4.4 bewaffnet können wir uns mit unseren konkreten projektspezifischen Anforderungen als Referenz an die Bewertung konkreter Frameworks machen.

Zunächst wenden wir uns aber einem anderen Thema zu, nämlich der Frage, ob wir uns Komponenten aus dem Open-Source-Bereich oder kommerzielle Komponenten beschaffen sollen.

## 6.2 Open Source versus kostenpflichtig

Die Frage, ob man Geld für Software ausgeben sollte oder nicht besser damit bedient ist, Open-Source-Software einzusetzen, ist zwar grundsätzlich unabhängig von Keyword-Driven Testing, aber sie beschäftigt uns hier eben auch. Sie kann nicht unabhängig vom Angebot betrachtet werden: Angenommen, unsere technischen Anforderungen würden ausschließlich von einem Open-Source-Produkt erfüllt, wir wären aber auf eine Gewährleistung durch einen Hersteller angewiesen – dann müssten wir uns schon überlegen, was uns wichtiger ist: die technischen Anforderungen oder die bezahlte Wartung. Oder wir müssten Wege finden, trotz Open Source Gewährleistung zu bekommen. Das Gleiche gilt selbstverständlich genauso für den umgekehrten Fall.

Beschäftigen wir uns doch erst mal mit der Frage, was Open Source denn überhaupt bedeutet.

### 6.2.1 Definition von Open Source

Es gibt unterschiedliche Definitionen von Open Source, doch die Definition der »Open Source Initiative« ist die geläufigste [27].

Die Definition lässt sich auf die folgenden Grundsätze reduzieren:

**Kostenfreie Weitergabe:** Die Software muss kostenfrei zur Verfügung gestellt werden und die Lizenz darf niemanden daran hindern, die Software als Teil eines Softwarepaketes zu verteilen oder sogar zu verkaufen.

**Offener Quellcode:** Die Software muss quelloffen, also in nicht kompilierter und für Menschen lesbarer Form, zur Verfügung gestellt werden.

**Modifikationen und Verwendung:** Die Lizenz muss es ermöglichen, die Software zu verändern und veränderte oder abgeleitete Werke weiter zu verteilen. Auch die Weiterverteilung in kostenpflichtigen Programmpaketen oder als Teil neuer Software muss erlaubt sein.

**Diskriminierungs- & einschränkungsfrei:** Die Softwarelizenz darf weder Personen oder Gruppen diskriminieren noch darf sie Einschränkungen für den Nutzerkreis oder den Verwendungszweck definieren. Auch darf die Lizenz andere Software nicht einschränken und beispielsweise die Open-Source-Lizenzierung einer Software verlangen, die mit der Open-Source-Software gemeinsam auf einem Datenträger oder Computer ist.

**Lizenzweitergabe:** Die Rechte der freien Nutzung, Verbreitung und weitere durch die Lizenz eingeräumte Rechte für die Softwarekomponente müssen für alle weiter gelten, die diese Software erhalten. Ein Abmildern der Lizenz für Nutzer ist nicht erlaubt.

Open Source beinhaltet also mehr als einfach nur die Kostenfreiheit. Für uns als Nutzer sind zwei Aspekte dieser Definition essenziell. Die kostenfreie Nutzung und Weitergabe ist erst mal schon ganz nett. Die garantierte Möglichkeit der Modifikation kann für uns aber richtig wichtig werden.

*Lizenzen*

Für Open-Source-Software gibt es eine Vielzahl unterschiedlicher Lizenzen. Die häufigste Frage und einer der wichtigsten Unterschiede der Lizenzen ist, inwiefern die Lizenz Einschränkungen in der Nutzung bedeutet.

*Copyleft*

Es gibt sogenannte starke »Copyleft«-Lizenzen, zu denen die GPL-Lizenzen gehören. Diese Lizenzen verlangen, dass Software, die auf diesem GPL-lizenzierten Code aufbaut oder daraus modifiziert wurde, bei Weitergabe ebenfalls unter der GPL-Lizenz lizenziert werden muss. Es sagt nicht aus, dass diese Software zu veröffentlichen ist! Solange wir also nicht vorhaben, unsere Automatisierungslösung an jemand Drittes zu verkaufen, ist das völlig irrelevant. Dennoch gibt es in manchen Unternehmen Vorbehalte und Unsicherheit und daraus resultierend ein Verbot zur Verwendung von GPL-Software. Fragen Sie vorsichtig nach, aber versuchen Sie keine schlafenden Hunde zu wecken.

*Freizügige Lizenzen*

Die am häufigsten anzutreffenden Lizenzen im Bereich Testautomatisierung sind sogenannte »Freizügige Open-Source-Lizenzen« (Permissive licence). Diese Lizenzen haben nicht die zuvor beschriebenen Einschränkungen, was die Abwandlung und Weiterverwendung angeht. Die bekanntesten Vertreter dieser Lizenzen sind die »Apache 2.0 License«, die »MIT-License« und die »BSD-License«. Diese Lizenzen stellen eine wichtige Anforderung an die Nutzer: Es ist meist nicht erlaubt, den Lizenztext und den Urheber der Software aus dem Quellcode zu entfernen. Wenn wir Open-Source-Code verwenden, der unter der Apache 2.0 lizenziert ist, müssen wir also in diesem Sourcecode stehen lassen, von wem er ursprünglich unter welcher Lizenz entwickelt wurde.

*LGPL*

Eine abgeschwächte Copyleft-Lizenz ist die LGPL (Lesser General Public License). Sie zählt zwar auch zu den Copyleft-Lizenzen, sie fordert aber nur bei Modifikation des originalen Werks, diese Modifikation erneut unter der LGPL zu lizenzieren, nicht aber bei der Einbettung dieser in eigene Software.

*Haftungsausschluss*

Eines haben eigentlich alle Open-Source-Lizenzen gemein: Sie schließen jegliche Haftung des Urhebers aus. Sollte also die Software nicht wie angekündigt funktionieren oder einen Schaden verursachen, haftet der Nutzer, nicht der Hersteller. Dies ist tatsächlich der Hauptgrund, warum manche Firmen oder Behörden die Nutzung von Open-Source-Software limitieren, solange kein Wartungsvertrag abgeschlossen wurde. Wir sind im Zweifel also selbst dafür verantwortlich, einen Bug zu beheben, wohingegen bei einer gekauften oder sogar gemieteten Software

der Hersteller eine Gewährleistung für ein mängelfreies Produkt geben muss.

### 6.2.2 Nachteile von Open Source

Welche Probleme und Nachteile bringt Open Source denn nun so alles mit sich?

**Unzuverlässigkeit:** Es ist ein weitverbreiteter Glaube, dass Open Source immer von »der Community« entwickelt wird und dass damit irgendwelche Nerds in irgendwelchen Kellern gemeint sind, die die Software zum Spaß schreiben. Gut, zugegeben, auch das ist teilweise der Fall, aber immer häufiger stehen hinter größeren Open-Source-Projekten Firmen, Stiftungen oder Vereine, die die Pflege finanzieren. Dennoch kommt es immer wieder vor, dass Projekte stillgelegt und nicht weiterentwickelt werden.

Das ist aber kein alleiniges Problem im Open-Source-Bereich. Wir erleben gerade, dass es ein paar größere Investoren im Bereich Testwerkzeuge gibt, die fleißig den Markt konsolidieren. Wir beobachteten gerade in den letzten Jahren, dass mehrere Testmanagementsysteme und Testautomaten vom Hersteller abgekündigt wurden und die Kunden mit ihrer jahrelang aufgebauten Automatisierungslösung auf einmal vor dem Abgrund standen. Der schlimmste Fall, der mit Open Source passieren kann, wäre doch, dass wir jetzt eine Entwicklerin bezahlen müssen, damit sie für uns dieses Projekt weiter pflegt. In der Regel haben wir ja vorher lange nichts für die Pflege bezahlt und damit rechnet es sich immer noch.

**Fehlerbehebung:** Wenn wir für eine Software bezahlen, bekommen wir in der Regel auch Wartung. Tritt ein Fehler auf, behebt der Hersteller diesen und wir erhalten ein Update. Auch das passiert in der Regel mit gepflegten Open-Source-Projekten, denn meistens stehen sehr viele Firmen hinter der Nutzung und irgendwer gibt schon Geld, damit ein Bug behoben wird. Im Zweifel sind wir wieder diejenigen, die mal was zahlen müssen.

Aber zugesicherte SLAs (Service Level Agreements) gibt es nicht. Mit einem Hersteller kann ich einen Vertrag schließen, der besagt, dass ein Fehler einer bestimmten Kritikalität in einem zugesicherten Zeitfenster behoben wird. Solche Wartungsverträge gibt es kaum für Open Source. Aber auch das ist Verhandlungssache und ggf. ein lösbares Problem.

**Keine Beratung/Schulung:** Für Open-Source-Software gibt es ja kein Herstellerunternehmen, also niemanden, der Einweisung oder Beratung übernimmt. Das ist zwar richtig, aber die Schlussfolgerung

daraus kann falsch sein. Auch ohne Herstellerfirma gibt es im Open-Source-Bereich viele Firmen, die Schulungen, Weiterentwicklungen und Beratung anbieten.

**Keine Komplettlösung:** Tatsächlich liegt bei vielen Open-Source-Projekten der Fokus auf der Problemlösung und nicht auf dem Aussehen oder dem Rundum-sorglos-Paket mit Doku und Installer. Das heißt, dass wir für die Nutzung mancher Open-Source-Komponenten selbst Anbindungen schreiben müssen oder für die Nutzung Entwicklerinnen oder Experten benötigen. Dennoch lösen gerade diese kleinen Projekte oft genau das Problem, das die bezahlte Software nicht lösen kann.

Am Ende muss jeder selbst entscheiden, wie die Vor- und Nachteile zu bewerten sind. Wir können hier nur ein paar Tipps geben:

- Unterstützen Sie die Open-Source-Projekte finanziell, die im Unternehmen eingesetzt werden, um eine langfristige Wartung zu garantieren.
- Prüfen Sie die Verbreitung der Software. Je mehr Nutzerinnen sie hat, umso sicherer ist der Fortbestand und die verfügbare Unterstützung.
- Prüfen Sie, wie lange es das Projekt bereits gibt und wann der letzte Commit war: Wird es noch gewartet?
- Prüfen Sie, ob es bezahlten Support durch Dritte oder die Entwicklerinnen selbst gibt.
- Prüfen Sie bei kleinen Projekten, ob die Entwicklerinnen an einer Kooperation interessiert sind oder ob der Code qualitativ gut genug ist, um durch Dritte gewartet zu werden.

Oft macht es die gute Mischung. Reinen Open-Source-Lösungen fehlt es oft an dem letzen Schliff. Vollständig gekaufte Lösungen sorgen im Gegenzug für eine starke Abhängigkeit.
Schauen wir uns nun ein paar gute Mischungen an.

## 6.3 Professionelle Bausteine für Frameworks

In Abschnitt 4.3 »Frameworks in der Norm« haben wir das Modell der ISO 29119 kennengelernt, das ein KDT-Framework in verschiedene Elemente gliedert, wohl wissend, dass diese Elemente weder einzeln verfügbar sind noch dass es ein einzelnes fertiges Werkzeug gibt, das diese Elemente vollständig abbildet – jedenfalls bis heute. Also muss ein Keyword-Driven Testing Framework in der Praxis aus verschiedenen Bausteinen zusammengesetzt werden, die jeweils die Teile abdecken,

die für den gewünschten Ansatz von Keyword-Driven Testing nötig sind.

Drei solcher Werkzeuge oder Bausteine stellen wir in diesem Abschnitt vor. Natürlich gibt es viel mehr. Wir erlauben es uns aber, uns auf wenige Beispiele zu beschränken, die uns vertraut sind und von denen wir überzeugt sind, da wir mit ihnen gute Erfahrungen gemacht haben.

### 6.3.1 Robot Framework®

Robot Framework [20] ist ein Open Source (Apache 2.0 License) Keyword-Driven Testing Framework, das von der Robot Framework Foundation [17] weiterentwickelt und gepflegt wird. Es wurde zwar mit Fokus auf Testautomatisierung begonnen, findet heute weltweit aber auch für RPA Anwendung.

Es wurde im Rahmen eines Projektes 2005 bei Nokia Networks in Finnland durch Pekka Klärck (damals Pekka Laukkanen) und Juha Rantanen ins Leben gerufen. Pekka veröffentlichte den von ihm gewählten Keyword-Driven-Ansatz in seiner Master-Thesis 2006 [33]. Er ist heute noch der mit der Pflege des Frameworks beauftragte leitende Entwickler.

Robot Framework wurde in den ersten Jahren mit einem Entwicklungsteam bei Nokia Networks weiterentwickelt. 2008 wurde es auf Bestreben einiger Entwicklerinnen hin unter der Apache-Lizenz 2.0 Open Source gestellt und ist kostenfrei auf GitHub [19] verfügbar.

Im Jahr 2015 wurde die Robot Framework Foundation gegründet und das Projekt von Nokia Networks an die Foundation übertragen. Der Zweck der Foundation besteht darin, die Pflege und Weiterentwicklung von Robot Framework zu ermöglichen sowie dessen Verwendung zu unterstützen. Stand 2021 sind 50 internationale Unternehmen Mitglied der Foundation und sichern somit die Weiterentwicklung von Robot Framework und dessen kostenfreie Verfügbarkeit.

Spezifikationen von Keywords, Variablen und von Tests oder Tasks werden in einer eigens für Keyword-Driven Testing entworfenen Syntax geschrieben und in Textdateien gespeichert. Diese Syntax ist darauf fokussiert, die Spezifikation – ohne unnötige Sonderzeichen – möglichst lesbar und einfach verständlich zu machen.

Leerzeichen sind beim Robot Framework Teil der Syntax. Ein einzelnes Leerzeichen ist ein normal erlaubtes Zeichen, sowohl im Namen und Aufruf der Keywords als auch in den Werten. Zwei oder mehr Leerzeichen werden als Trenner zwischen Keyword und Argumenten interpretiert. Genauer werden wir auf die Syntax in Kapitel 7 »Praxis mit Robot Framework« eingehen.

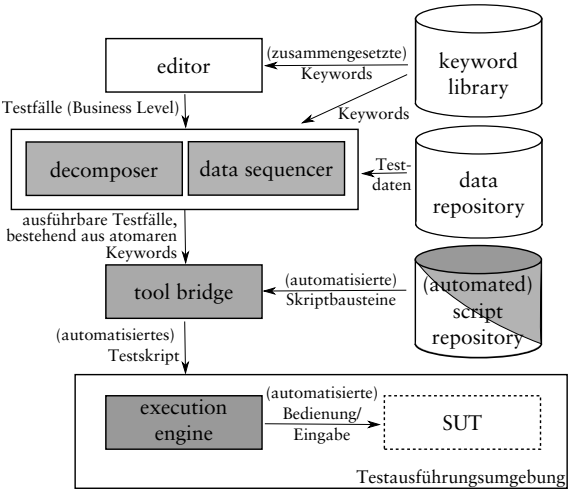
**Abb. 6-1** *Testsuite in Robot Framework*

```
*** Settings ***
Resource        keywords.resource


*** Test Cases ***
Auswahl mit Gutschein
    [Setup]     Starte App
    Login                   user1     123456
    Wähle Produkt           Weste
    Eingabe Gutschein       ${EMPTY}
    Prüfe Preis             87,50
    Bestätige Kauf
    [Teardown]      Beende App
```

Um zu visualisieren, wie Robot Framework in das Framework-Modell der ISO 29119-5 einzuordnen ist, wird das Diagramm in Abbildung 4-4 aus Abschnitt 4.3 auf Robot Framework angepasst dargestellt. In Abbildung 6-2 sind die dunkel eingefärbten Bereiche diejenigen Elemente aus dem ISO-Modell, die von Robot Framework abgedeckt werden.

**Abb. 6-2** *Von Robot Framework abgedeckte Elemente*




**Editor:** Der Editor ist nicht Teil des Frameworks. Da das Robot Framework auf Textdateien aufsetzt, kann als Editor jeder beliebige Texteditor genutzt werden. Für eine volle Unterstützung mit Syntax-Highlighting und Codevervollständigung werden Entwick-

lungsumgebungen wie beispielsweise Microsoft Visual Studio Code mit entsprechenden Erweiterungen verwendet.

**Decomposer:** Robot Framework liest die textbasierten Keywords und Testspezifikationen ein und baut ein ausführbares Datenmodell – das Execution Model – auf. Beim Aufbau des Datenmodells werden zunächst nur die High-Level Keywords aufgelöst. Die darin enthaltenen Intermediate-Level und Low-Level Keywords werden erst zur Laufzeit aufgelöst, wenn sie benötigt werden, da Keyword-Bibliotheken auch dynamisch importiert werden können.

**Keyword Library:** Keywords, die im Robot Framework erstellt werden (exklusive der Low-Level Keywords), werden ähnlich den Tests in Textdateien, den sogenannten Resource Files, spezifiziert und gesichert. Diese werden ebenfalls als erster Schritt der Testdurchführung eingelesen und in das ausführbare Datenmodell eingebaut. Da diese Keywords nicht in einer Datenbank, sondern in nicht katalogisierten Textdateien abgelegt werden, ist die Unterstützung ihrer Verwaltung sehr stark von externen Werkzeugen, wie Entwicklungsumgebungen, abhängig.

**Data Sequencer:** Testdaten können sowohl zu Beginn der Ausführung als auch zur Laufzeit aus verschiedensten Quellen vom Robot Framework gelesen werden. Datengetriebene Tests werden standardmäßig in Textform spezifiziert und vom »Sequencer« ausgerollt. Aufgrund der offenen APIs kann der interne »Data Sequencer« erweitert werden (z.B. durch DataDriver) und somit auch Daten aus beliebigen Quellen, unter anderem Datenbanken, Excel-Dokumenten oder Datengeneratoren, genutzt werden.

**Data Repository:** Robot Framework kann direkt auf Variablen und deren Werte zugreifen, die wie die Keywords in Resource Files gespeichert sind. Zusätzlich kann YAML als Datenquelle dienen, oder Python-Code kann dynamisch Daten bereitstellen und somit live aus beliebigen Quellen beziehen. Das »Data Repository« ist also genau wie die »Keyword Library« nicht katalogisiert und werkzeuggestützt verwaltet.

**Tool Bridge:** Robot Framework hat ein sehr flexibles Konzept, beliebige Programmiersprachen anzubinden. Über das »Remote Library API« können – zusätzlich zu nativen Keywords auf Basis von Python – auch Keywords anderer Sprachen direkt oder über das Netzwerk entfernt ausgeführt werden.

**Script Repository:** Robot Framework bringt zehn Keyword-Bibliotheken mit, die sich auf generische Funktionen konzentrieren. Weitere externe Bibliotheken können selbst geschrieben oder aus der Community bezogen werden. Da es sich in der Regel um externe Python-Bibliotheken handelt, ist die Verwaltung, Installation

und Distribution dieser Bibliotheken nicht Aufgabe und Teil von Robot Framework, sondern wird über die Python-Paketverwaltung »PIP« organisiert. Da aber auch andere Programmiersprachen über die »Tool Bridge« angesprochen werden können, ist das spezifische »Script Repository« sehr stark abhängig von der eingesetzten Programmiersprache. Robot Framework definiert nur die Schnittstelle zum Repository.

**Execution Engine:** Der eigentliche Kern von Robot Framework ist die »Execution Engine« selbst. Dieser Teil führt das Execution Model aus, ruft die Low-Level Keywords auf und erzeugt die Berichte. Auch hier kann über offene APIs eingegriffen und der Ablauf manipuliert werden.

Robot Framework enthält zwar keinen eigenen Editor, mit »Ride« [14] ist allerdings ein eigens für das Robot Framework entwickelter Editor verfügbar. Häufiger genutzt werden aber Entwicklungsumgebungen, wie Visual Studio Code oder PyCharm, die für die Unterstützung von Robot Framework durch Plug-ins erweitert wurden.

Bezüglich der unterstützten bzw. enthaltenen Automatisierungstechnologien kann Robot Framework oder genauer gesagt die Robot Framework Community besonders punkten. Robot Framework unterstützt, im Gegensatz zu fast allen anderen Open Source Test Frameworks, das Verschachteln von Keywords und somit eine Architektur mit verschiedenen Layern. Aufgrund dieser Architektur entstanden über die Jahre in der Robot Framework Community einige Hundert Keyword Libraries für die Technologieschicht.

Einige Beispiele:

- Die SeleniumLibrary bietet über 170 fertige Keywords wie `Open Browser`, `Click Element` und `Input Text` und ermöglicht es, komplett ohne zu programmieren, eigene Keywords zu bauen und eine Webautomatisierung zu realisieren.
- Die RequestsLibrary bietet mit ihren ca. 30 Keywords alles, was man zum Testen einer HTTP-Schnittstelle benötigt.
- Die RoboSAPiens Library wurde entworfen, um SAP-GUIs in natürlicher Sprache zu bedienen.
- Die DataBaseLibrary spricht fast alle SQL-Dialekte und kann direkt mit Datenbanken kommunizieren.

Die Libraries sind entweder über die Webseite von Robot Framework, GitHub.com oder PyPI.org zu finden. Da all diese Libraries von der Robot Framework Community bereitgestellt und gepflegt werden, kann man sie als Teil des Frameworks betrachten, obwohl sie im Sinne der ISO nicht dazugerechnet werden.

Natürlich ist es wie bei den Komponenten bereits erwähnt auch möglich, mit beliebigen Programmiersprachen eigene Low-Level Keywords zu implementieren und diese als Library in der eigenen Automatisierung zu nutzen.

Robot Framework deckt zwar nicht alleine, aber in Kooperation mit weiteren Open-Source-Komponenten die vollständige Architektur für Testautomatisierung mit Keyword-Driven Testing ab. Verglichen mit den meist kostenpflichtigen »Full-Stack-Automaten« hat es dabei den Vorteil, dass die Automatisierungstechnologie von der Testspezifikation vollständig unabhängig und damit austauschbar ist. Wir haben bereits in mehreren Projekten erfolgreich die »SeleniumLibrary« durch die modernere und effizientere Playwright-basierte Library »Robot Framework Browser« abgelöst, ohne die Testfälle anpassen zu müssen.

Robot Framework bietet, verglichen mit den beiden folgenden Beispielen, keine Unterstützung für manuellen Test, keinen eigenen Editor und keine Komponente zur Verwaltung der Artefakte. Bei sehr großen Projekten kann daher der Überblick leiden und der Wunsch nach einem Managementwerkzeug wachsen.

### 6.3.2 imbus Test*Bench* Enterprise Edition

Mit der Test*Bench* Enterprise Edition[1] (kurz: »Test*Bench* EE«) steht ein sehr ausgereiftes Werkzeug zur Verfügung, bei dessen Entwicklung von Anfang an Keyword-Driven Testing ein zentraler Aspekt war. Zum Zeitpunkt der initialen Entwicklung, Anfang der 2000er-Jahre, war der Begriff des Keyword-Driven Testing noch nicht gesetzt und so sprach man auch von »Action Words« und anderen Bezeichnungen für Keywords. In der Test*Bench* EE wurde das, was wir heute als Keywords kennen, »Interaktionen« genannt, weil sie eine definierte Interaktion zwischen Testerin und Testobjekt repräsentieren.

Die Test*Bench* EE unterstützt hier nicht nur die Definition dieser Keywords, sondern arbeitet auch mit typisierten Daten und ist in der

[1] DISCLAIMER: An dieser Stelle sei nochmals darauf hingewiesen, dass beide Autoren bei der imbus AG angestellt sind. Wir sind seit Jahren mit dem Thema Keyword-Driven Testing befasst und haben beide die imbus Test*Bench* mitgeprägt. Daher sind die Erfahrungen mit der Test*Bench* zwar aus allererster Hand, jedoch nicht unabhängig. Aber: Keiner von uns würde jemals, weder für einen Arbeitgeber bei einem Kunden noch in diesem Buch ein Werkzeug empfehlen, hinter dem er nicht voll stehen könnte. Bei der Gelegenheit wollen wir festhalten, dass dieses Buch und die Zeit der Ausarbeitung in keiner Weise von der imbus AG finanziert oder subventioniert werden und wir hier ausdrücklich nur unsere eigene Meinung und Überzeugung ausdrücken.

Lage, Datenklassen aufzubauen und die entsprechenden Datenobjekte zu speichern.

Verglichen mit Robot Framework legt die Test*Bench* konzeptionell einen wesentlich stärkeren Fokus auf das Management der Daten, Interaktionen und der Testfälle.

Da in Deutschland Banken, Versicherungen, Medizintechnikkonzerne und weitere komplexe Branchen zum Alltag der Test*Bench* EE gehören, wurde über die Jahre ein besonderer Wert auf Data-Driven Testing und die Versionierung von Tests zum Zwecke der Nachweissicherheit gelegt.

Die Test*Bench* EE konzentriert sich ausschließlich auf den Bereich des Testens. Daher werden andere prozessrelevante Werkzeuge, wie Anforderungsmanagement und Fehlermanagement, über offene Schnittstellen angebunden.

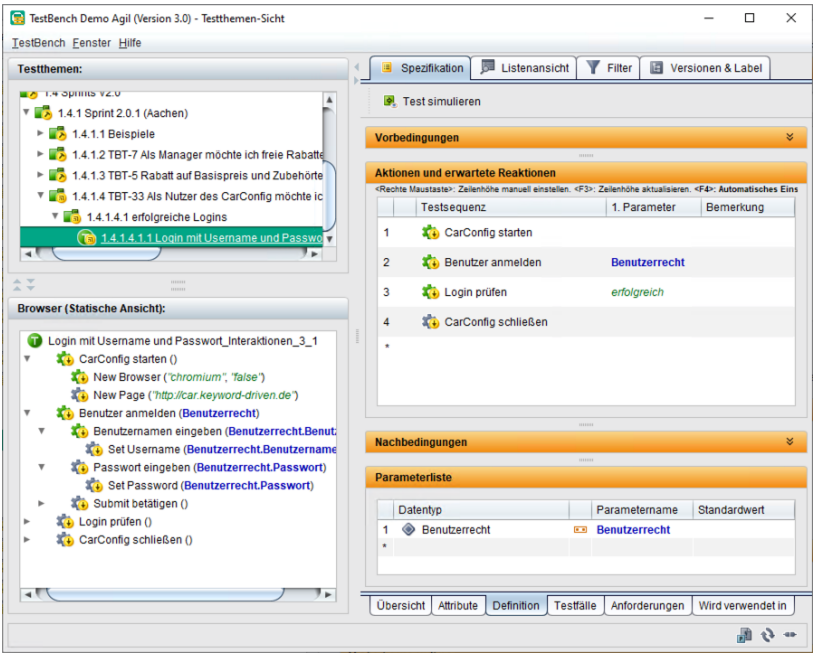



***Abb. 6-3*** *TestBench EE*

In Abbildung 6-3 wird exemplarisch ein Bildschirm der Test*Bench* EE gezeigt. Oben links im Bild ist der Testthemenbaum zu erkennen, in dem Tests strukturiert abgelegt werden, mit dem selektierten Test. Im rechten Teil befindet sich die »Testsequenz« dieses Tests, die aus strukturierten High-Level Keywords zusammengesetzt ist. Im linken unteren Bereich ist der Browser zu sehen, mit dessen Hilfe ein Testdesigner die eigentliche Struktur der Testspezifikation – mit den Intermediate-Level Keywords bis hin zu den Low-Level Keywords – analysieren kann.

Wir betrachten Test*Bench* im Folgenden bezogen auf Keyword-Driven Testing, also: Wie ordnet sich Test*Bench* EE in die Architektur des ISO-Modells ein?

Einen Fokus legt die Test*Bench* auf die Bereiche des Test- und Keyword-Designs. Wie aus dem Screenshot ersichtlich ist der Editor zentraler Bestandteil der Benutzeroberfläche und darauf ausgelegt, auch Testdesigner und Fachleute abzuholen, die nicht aus der Softwareentwicklung kommen. Über Drag & Drop können Interaktionen zu Testsequenzen zusammengestellt werden.

In Abbildung 6-4 (Framework für Testautomatisierung) und 6-5 (Framework für manuellen Test) haben wir diesmal die Elemente aus dem ISO-Modell dunkel markiert, die Test*Bench* EE abdeckt.

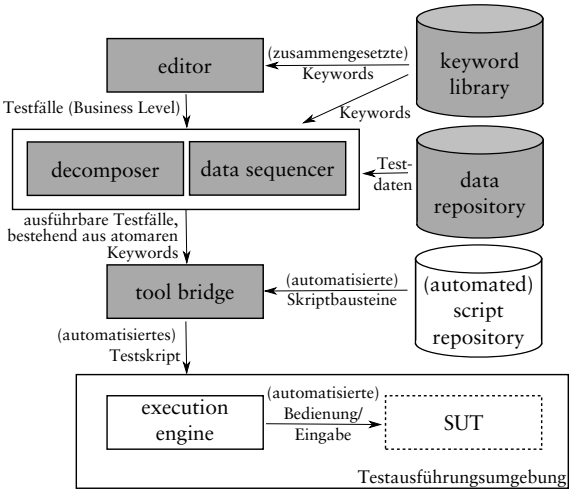



***Abb. 6-4***
*Von TestBench CS & EE abgedeckte Elemente (automatisiert)*

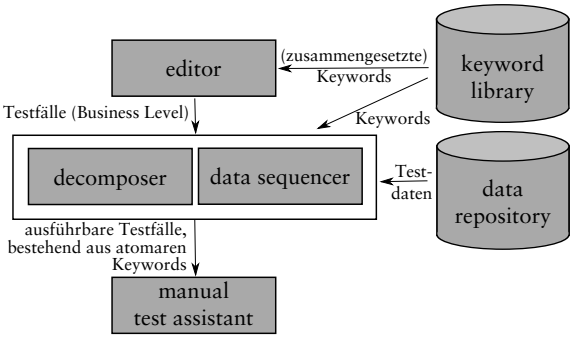



***Abb. 6-5***
*Von TestBench CS & EE abgedeckte Elemente (manuell)*

Im Einzelnen:

**Editor:** Der Editor, mit dem auch »nicht durch Keyword getriebene« Testfälle erstellt werden können, erlaubt das Zusammenstellen neuer Tests aus Keywords – oder Keywords aus anderen Keywords – per Drag & Drop. Die Eingabe und Verwaltung von Testdaten wird besonders unterstützt, da die Test*Bench* den Bezug zwischen Interaktionen und typisiert abgelegten Daten verwaltet und dementsprechend den Testdesignern nur die Daten anbietet, die zu der aktuellen Interaktion passen.

**Keyword Library:** Keywords werden über den Test*Bench* -Server in einer Datenbank gespeichert. Die Verwendung von Keywords in Tests oder in anderen Keywords sowie die Verknüpfung von Datentypen zu Argumenten werden in dieser relationalen Datenbank über Referenzen gespeichert und sind unabhängig von Namensänderungen oder Umstrukturierungen der Ablagesysteme sicher verknüpft. Die Keywords können mit weiteren Informationen wie Beschreibung und Status versehen werden, was die Nutzung im manuellen Test erleichtert.

**Data Repository:** Daten werden in der Test*Bench* typisiert verwaltet. Diese Datentypen enthalten in Äquivalenzklassen aufgeteilte Werte, die in Test*Bench* Repräsentanten genannt werden. Im Falle von strukturierten Datentypen (Datenklassen) handelt es sich bei den in Datentypen gespeicherten Repräsentanten um strukturierte Datenobjekte, die eine Kombination von Werten niedrigerer Komplexität enthalten. Da auch diese Repräsentanten und Datentypen in der relationalen Datenbank der Test*Bench* gespeichert sind, sind Verwendungsnachweise oder selbst Veränderungen von Werten an einer zentralen Stelle möglich.

**Data Sequencer:** Daten werden für die Ausführung von High-Level Keywords zu Low-Level Keywords weitergeleitet. Dazu werden komplexe Datenobjekte soweit nötig zerlegt und ihre Bestandteile passend den nutzenden Keywords zugeordnet.

Datengetriebene Test werden entsprechend der Menge der Datensätze vervielfältigt und abstrakte Testfälle werden zu mehreren konkreten Testfällen umgewandelt.

**Decomposer:** Test*Bench* exportiert die Tests in einer XML- oder JSON-Datenstruktur. Damit diese Datenstrukturen automatisiert sequenziell ausgeführt werden können, bereitet die Test*Bench* die Testfälle für die manuelle und automatisierte Durchführung so auf, dass High-Level Keywords bis auf die Keywords der niedrigsten Ebene aufgelöst werden. Der Decomposer und der Data Sequencer

sind Teil einer Komponente der Test*Bench* namens »iTEP« (imbus Test Execution Plugin).

**Tool Bridge:** Abhängig von der verwendeten Automatisierungstechnologie können unterschiedliche Implementierungen der Tool Bridge zum Einsatz kommen. Im Test*Bench* -Kontext werden diese »iTEP-Wrapper« genannt. Die Verknüpfung zwischen dem nutzungsspezifischen Script Repository und den vom Decomposer sequenzialisierten Keyword-Aufrufen kann auf unterschiedlichste Weise implementiert werden.

**Execution Engine:** Nutzt man das Robot Framework als Execution Engine unterhalb der Test*Bench* EE, bringt die Test*Bench* einen Skriptgenerator mit, der Testfallspezifikationen für das Robot Framework erzeugt. Nach der Testausführung durch Robot Framework werden die Testprotokolle ausgelesen und in Test*Bench* zurückgeschrieben.

In anderen Szenarien ist die Execution Engine ebenfalls Teil des iTEP-Wrappers und Interaktionen werden sofort als Funktionsaufrufe ausgeführt und deren Ergebnis als Testergebnis protokolliert. Fertige Wrapper, um beispielsweise in QF-Test oder Ranorex implementierte Script Repositories (Automatisierungstechnologie) zu nutzen, können bezogen werden. In speziellen Anwendungsfällen können sie auch vollständig spezifisch implementiert werden.

**Manual Test Assistant:** Die manuelle Testdurchführung wird von der Test*Bench* EE durch die Komponente »iTORX« (imbus Tool for Online and Remote eXecution) unterstützt. Im iTORX sieht eine Testerin jeden einzelnen Schritt, der sich aus der Auflösung der Keywords ergibt, und kann während der Durchführung die Ergebnisse je Schritt erfassen.

Lediglich die Komponente »Script Repository« und je nach Situation die »Execution Engine« sind nicht Teil des Scope der Test*Bench* EE, sie bleiben Domäne des Testautomatisierungswerkzeugs. Anders gesagt: Test*Bench* ist vollständig technologieagnostisch und unterstützt mit ihren APIs die Anbindung beliebiger Automaten.

Wir haben jetzt schon zwei Komponenten kennengelernt, aus denen wir sogar bereits ein komplettes Keyword-Driven Testing Framework zusammensetzen können, wie in Abbildung 6-6 zu sehen ist. Test*Bench* (hellgrau) kümmert sich um die Spezifikation der High-Level und Intermediate-Level Keyword-Bibliotheken sowie den manuellen Test und Robot Framework (dunkelgrau) deckt den Teil der Automatisierung ab. Hinzu kommen noch die notwendigen Low-Level Keywords aus den Open Source Libraries und fertig ist ein Keyword-Driven Testing Framework.

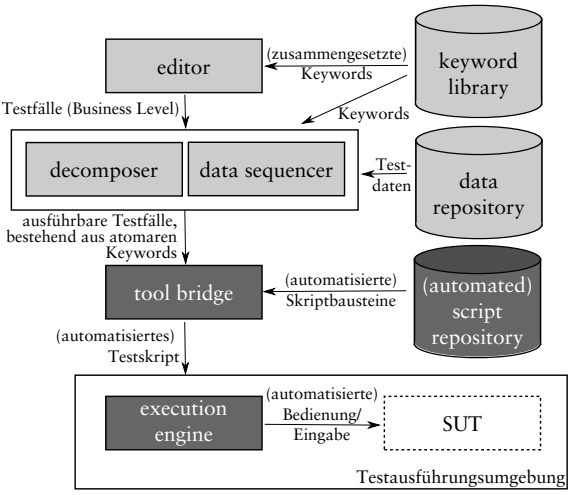



***Abb. 6-6***
*TestBench mit Robot Framework*

### 6.3.3 imbus TestBench Cloud Services

Die TestBench CS[2] ist ein modernes Testmanagementwerkzeug, das über die Cloud agilen Entwicklungsteams eine Arbeitsplattform bietet, selbst wenn die Teammitglieder weltweit verteilt sind. Wie bei der Test*Bench* EE liegt auch hier der Schwerpunkt auf Testmanagement und Testdesign.

Via API kann die TestBench CS von beliebigen Fremdsystemen angesprochen werden, sofern diese nur in der Lage sind, ein REST-API anzusteuern. Für Werkzeuge zur Testautomatisierung ist dies zwar im Allgemeinen kein Problem, wir werden weiter unten aber noch einen anderen Weg zur Anbindung einer Testautomatisierung zeigen.

Auch hier einige Hauptmerkmale:

- TestBench CS ist als Cloud-System von überall auf der Welt per Internet erreichbar. Alternativ ist auch eine On-Premises-Installation im eigenen Netzwerk möglich – dann ist die Cloud privat.
- Es wird sowohl spezifikationsbasierter (hierzu zählt Keyword-Driven Testing) Test als auch explorativer Test unterstützt.
- Ein Schwerpunkt liegt auf Kollaboration: Teammitglieder können sich per internem Chat austauschen, Dokumente zeigen (»follow-me«) und beim explorativen Test im Canvas den gemeinsamen Fortschritt verfolgen.

[2]Hinweis: Die Cloud-Variante der TestBench (TestBench CC) schreibt sich nicht kursiv – Test*Bench* EE schon.

- Anforderungsbasierter Test ist ein Basiskonzept: Requirements (Epics, User Stories) können mit Testfällen und Defects in sehr transparenter Weise verknüpft werden.
- Sowohl Data-Driven Testing als auch Keyword-Driven Testing werden unterstützt; Daten können beinahe an beliebiger Stelle im abstrakten Testfall eingesetzt werden, die Wirkung wird im konkreten Testfall unmittelbar sichtbar.
- Das vollständig offene API erlaubt nicht nur Anbindungen an beinahe beliebige andere Werkzeuge, sondern auch Ansteuerungen durch Eigenentwicklungen (beispielsweise Import von Testdaten).

Einen Eindruck davon, wie Testfälle in der TestBench CS unter Verwendung von Keywords aussehen können, vermittelt Abbildung 6-7. Auf der linken Seite ist in der Beschreibung der grobe Prozess illustriert. Rechts findet sich der eigentliche Testablauf. Darin ist ein Keyword `Preis Prüfen` selektiert, und der im Testfall zu verwendende Wert wird gerade eingetragen.

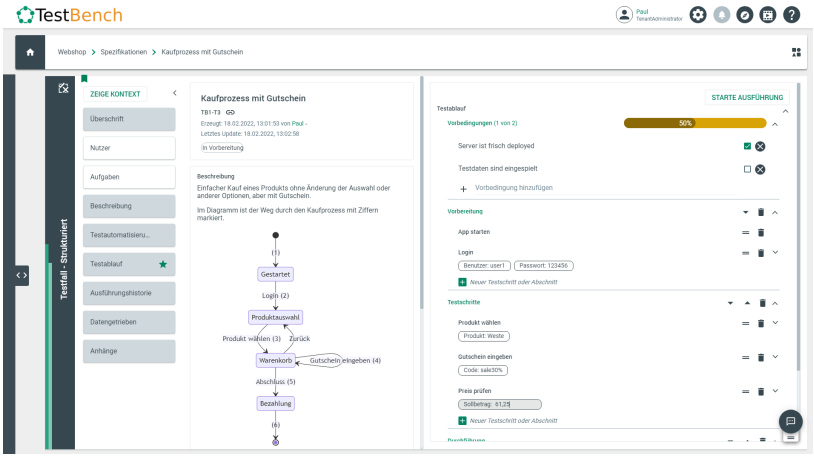



*Abb. 6-7*
*TestBench CS*

Bezüglich der Abdeckung der Elemente aus der ISO-Norm 29119-5 [49] können wir auf die Abbildungen 6-4 und 6-5 zurückgreifen. Auf dieser groben Ebene und in Bezug auf das Framework haben beide Werkzeuge der TestBench-Familie einen ähnlichen Ansatz. Ein Blick auf die Details lohnt sich:

**Editor:** Der integrierte Editor ermöglicht das Hinzufügen von Keywords wahlweise per Tastatur und Auto-Complete oder per Maus. Fehlende Keywords können »on-the-fly« definiert werden.

**Decomposer:** Hierarchische Keywords werden für manuelle Tests in der Durchführung auf jedem Detailgrad angezeigt und können je nach Belieben auf jeder Ebene mit »PASS« oder »FAIL« bewertet werden.

**Data Sequencer:** Data-Driven Testing wird auch in Keyword-Parametern vollständig umgesetzt – es können also dem Parameter eines Keywords entweder feste Werte (Literale) oder Referenzen an eine verknüpfte Datentabelle zugewiesen werden.

**Tool Bridge:** Die Verbindung zwischen TestBench CS und Testautomatisierungswerkzeugen ist aufgrund der Cloud-Architektur etwas anders gelöst, als es ohne den Cloud-Aspekt der Fall wäre – darauf kommen wir gleich noch mal zurück.

**Keyword Library:** Keywords werden in einem eigenen Repository abgelegt und enthalten Zusatzinformationen wie Beschreibung, Status, Implementierungsstatus und Verwendung.

**Data Repository:** Datentabellen können in TestBench definiert oder importiert und vom Testfall verwendet werden.

Ähnlich wie bei der Test*Bench* EE macht sich die TestBench CS die Vielfalt verfügbarer Testautomatisierungswerkzeuge zunutze und deckt daher die Elemente »Script Repository« und »Execution Engine« nicht selbst ab.

Da die TestBench CS ja als Cloud-System angelegt ist und der Client konsequent als Browseranwendung läuft, ist eine direkte Ansprache anderer Tools, etwa zur Testautomatisierung, nicht möglich. Abbildung 6-8 zeigt, wie dies stattdessen umgesetzt wird.

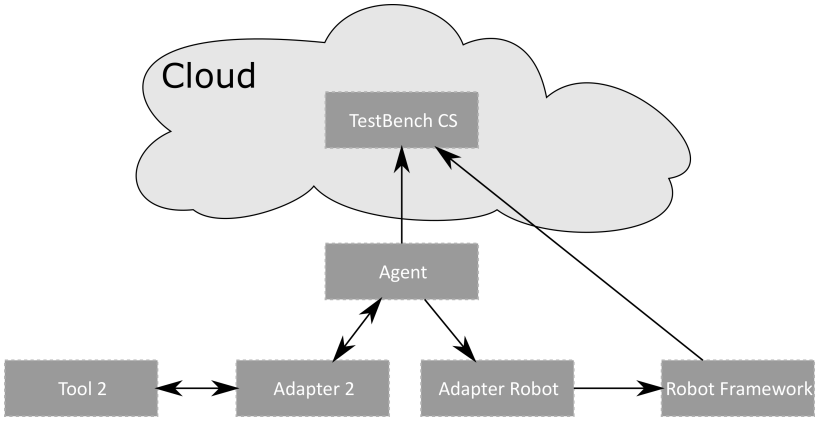



***Abb. 6-8*** *TestBench-CS-Automatisierungsarchitektur*

Eine lokal (in derselben Umgebung wie das anzusprechende Testautomatisierungswerkzeug) laufende Komponente, der »Agent«, prüft via API, ob in TestBench CS automatisiert durchzuführende Testfälle

vorhanden sind. Sobald das der Fall ist, wird abhängig davon, für welche durchzuführende Instanz der Testfall vorgesehen ist, das passende Framework zur Ausführung angestoßen. Dazu gibt es für jede ausführende Instanz einen sogenannten Adapter, der die spezifischen Eigenheiten jedes Tools kapselt.

Abbildung 6-8 zeigt zwei angebundene Frameworks, das »Robot Framework« und ein gewisses »Tool 2«. Gehen wir zunächst davon aus, dass das Robot Framework den Testfall durchführen soll: Dann wird über einen spezifischen »Adapter« für das Robot Framework auf Basis der Keywords aus dem Testfall ein Skript generiert und anschließend dem Robot Framework zur Ausführung übergeben.

Dank seiner sehr guten Erweiterbarkeit konnte das Robot-Framework so angepasst werden, dass es die Ergebnisse direkt zur TestBench zurückmeldet, ohne erneut den Weg über Adapter → Broker gehen zu müssen.

Falls ein Tool angesprochen werden soll, das nicht so leicht davon zu überzeugen ist, via REST-API mit TestBench CS zu sprechen – im Beispiel »Tool 2« –, ist es über den dafür passenden Adapter möglich, nicht nur die Testausführung anzustoßen, sondern auch die Ergebnisse (beispielsweise im JUnit-XML-Format) einzusammeln und der Testausführung in TestBench CS zuzuordnen.

## 6.4 Beispiele für Frameworks mit Bewertung

Im Folgenden werden zwei Frameworks beispielhaft genauer betrachtet und anhand der Merkmale aus Abschnitt 4.4 bewertet. Es sind echte Praxisbeispiele, die wir genau so in der Realität erlebt haben.

*Subjektivität der Bewertung*

Die Bewertung wurde jeweils im Interview mit einer einzelnen Person durchgeführt. Die persönlichen Erfahrungen mit anderen Frameworks oder Komponenten sowie die konkreten Projektanforderungen spielen bei der konkreten Bewertung und Priorisierung der einzelnen Anforderungen der Norm eine wichtige Rolle.

Dieselben Frameworks kommen in vielen unterschiedlichen Kontexten zum Einsatz. Ein Framework kann in einem Umfeld exzellent funktionieren, erfüllt aber die Anforderungen eines anderen Umfelds nicht genauso gut. Ein Beispiel können hierarchische Schlüsselwörter sein, die aufgrund der Struktur der Testobjekte hier als absolut notwendig und dort als überflüssig angesehen werden. Bei diesen Anforderungen kann es sich möglicherweise auch um ganz andere handeln als diejenigen, die wir der Norm entnommen haben. Dennoch prägen sie die Wahrnehmung ihrer Benutzer und somit die Bewertung. Daher

bezieht sich die folgende Bewertung jeweils auf exakt das beschriebene Einsatzszenario.

Wenn das alles so subjektiv und daher nicht übertragbar ist, weshalb haben wir uns doch entschieden, solche Bewertungen hier aufzuführen?

*Vergleichbarkeit herbeiführen*

Zum einen wollen wir einen Eindruck vermitteln, wie unterschiedlich Frameworks sein können. Auf den ersten Blick sind verschiedene Frameworks nicht vergleichbar. Mittels eines klaren Kriterienkatalogs wie der Anforderungen aus der ISO 29119-5 geht es aber doch.

Zum anderen wollten wir Sie, liebe Leser, motivieren, Ihre eigenen Anforderungen (und die Ihrer Kolleginnen, Ihres Teams) zu hinterfragen, zu erfassen und herauszufinden, was Sie eigentlich benötigen. Das ist wichtig für die Auswahl des für Sie richtigen Frameworks. Damit finden Sie heraus, ob Sie gut versorgt sind und zufrieden sein dürfen (falls Sie es gerade nicht sind – das Gras ist bekanntlich immer auf der anderen Seite viel grüner) oder ob es angebracht wäre, über Verbesserungen nachzudenken.

### 6.4.1 Framework 1: Test*Bench*

#### Steckbrief

Das Projektumfeld für das erste hier vorgestellte Framework ist komplex. Tatsächlich handelt es sich hier nicht um ein einzelnes »Projekt« – die gesamte QS eines Unternehmens ist involviert – und zum Zeitpunkt der Betrachtung sind mehr als 40 Einzelprojekte enthalten. Darin arbeiten rund 150 Personen an vier Standorten auf drei Kontinenten.

Im Rahmen dieser Projekte werden Embedded Systems (also Hardware) erstellt, die umfangreiche Schnittstellen zu jeweils anderen Systemen aufweisen.

Diese Schnittstellen sind einerseits Kommunikationsschnittstellen wie Ethernet, CAN-Bus oder HF-Funk, andererseits Sensoren und Aktoren wie Spannungsmessungen, Kontakte oder Relais.

Das Umfeld ist streng reguliert und durch Fehler an den Produkten könnten Menschen zu Schaden kommen. Daher besteht ein hoher Bedarf an gründlichen Tests und lückenloser Dokumentation.

Automatisierte Tests werden auf speziell für diesen Einsatz entwickelten HiL-Systemen ausgeführt. HiL steht dabei für »Hardware-in-the-Loop«. Das Testobjekt wird an allen relevanten Schnittstellen mit dem HiL verbunden und somit bildet der HiL zusammen mit dem Testobjekt eine sogenannte Regelschleife (engl. loop). Der HiL simuliert auch das spätere Gesamtsystem, in dem später das entwickelte Produkt als Komponente eingebettet werden wird.

Während eines Tests werden nun verschiedene Systemverhalten simuliert. Eingangssignale oder Befehle werden an das Testobjekt gesendet und die zu erwartenden Reaktionen werden überprüft. Manche dieser Tests müssen in Echtzeit durchgeführt werden, da im Test wirklich exakte Zeitspannen einzuhalten sind.

In diesen Projekten verwenden alle Tests, sowohl manuelle als auch automatisierte Tests, Keyword-Driven Testing. Als zentrales Werkzeug wird bei diesem Unternehmen die imbus Test*Bench* Enterprise eingesetzt. Sie wird sowohl als Testmanagement- und Testdesignwerkzeug als auch im manuellen Test als Durchführungsassistent und im Reviewprozess verwendet. Darüber hinaus werden selbst entwickelte Softwarekomponenten und die kundenspezifischen HiL-Systeme eingesetzt.

Da die Test*Bench* Enterprise bereits in Abschnitt 6.3.2 eingehend beschrieben wurde, verzichten wir hier auf weitere Details.

## Mapping der Tools

Die folgende Tabelle zeigt, wie die verwendeten Tools auf die Elemente aus der Struktur laut ISO 29119-5 abgebildet werden können.

Man sieht, dass bereits die Test*Bench* aus mehreren Komponenten besteht. Obwohl sie um das Thema Keyword-Driven Testing herum konzipiert ist und daher sich ein großer Teil der Elemente auf die Struktur abbilden lassen, werden weitere Komponenten benötigt, um Keyword-Driven Testing laut Norm vollständig abzudecken.

***Tabelle 6-1*** *Framework 1: Abdeckung der Komponenten aus ISO 29119-5*

| Element aus ISO 29119-5 | Tool/Element im Framework |
|---|---|
| Kernbestandteile | |
| Editor | imbus Test*Bench* (iTB) |
| Decomposer | imbus Test*Bench* iTEP (iTEP) |
| Data Sequencer | imbus Test*Bench* iTEP (iTEP) |
| Manual Test Assistant | imbus Test*Bench* iTORX (iTORX) |
| Tool Bridge | HiL-Code Generator |
| Execution Engine | HiL-System |
| Ablage/Repositories | |
| Keyword Library | imbus Test*Bench* (iTB) |
| Data Repository | imbus Test*Bench* (iTB) |
| Script Repository | HiL-Code in Git (Git) |

## Bewertung

In diesem Abschnitt wird in einer Reihe von Tabellen eine Bewertung des Frameworks aus diesem Praxisbeispiel vorgenommen. Diese Bewertung wurde im Hinblick auf genau diese Konstellation von Tools in genau diesem Kontext durchgeführt, und wir haben bewusst Subjektivität zugelassen – sie ist ohnehin nicht auszuschließen. Sie könnten also selbst in demselben Kontext zu einer anderen Bewertung kommen. Die Zahlen in dieser Tabelle sind für Sie, gerade in Ihrem Kontext, also nicht übertragbar.

Worum es uns hier geht, ist zu zeigen, wie eine solche Bewertung in der Realität aussehen kann.

Das Bewertungsschema ist folgendermaßen zu verstehen:

| Bewertung: | | Priorität: | |
|---|---|---|---|
| **0:** | nicht vorhanden | **0:** | irrelevant/unnötig |
| **1:** | teilweise erfüllt | **1:** | wünschenswert |
| **2:** | vollständig erfüllt | **2:** | unverzichtbar |

Anhand der Bewertung könnte man sagen: Die bewertende Person scheint ganz zufrieden mit der vorliegenden Lösung – und das, obwohl durchaus nicht alle Normelemente abgedeckt sind. Offenbar wurden sie aber nicht als fehlend empfunden.

*Tabelle 6-2 Framework 1: Basisanforderungen, Teil 1*

| Allgemein/Dokumentation | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.2 a) | keywords | 2 | Test*Bench* | 2 |
| 7.3.2 b) | parameters | 2 | Test*Bench* | 2 |
| 7.3.2 c) | default value | 2 | Test*Bench* | 2 |
| 7.3.2 d) | hierarchies | 1 | Test*Bench* | 2 |
| 7.3.2 e) | data | 1 | Test*Bench* | 2 |
| **Editor** | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.3 a) | actions in keywords | 2 | Test*Bench* | 2 |
| 7.3.3 b) | go to low-level keywords | 2 | Test*Bench* | 2 |
| 7.3.3 c) | parameters | 2 | Test*Bench* | 2 |
| 7.3.3 d) | comments | 2 | Test*Bench* | 2 |
| 7.3.3 e) | data sources | 2 | Test*Bench* | 2 |
| 7.3.3 f) | references | 2 | Test*Bench* | 2 |
| 7.3.3 g) | order | 2 | Test*Bench* | 2 |
| **Decomposer und Data Sequencer** | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.4 a) | Decomposer – parameters | 2 | iTEP | 2 |
| 7.3.4 b) | Data Sequencer – parameters | 2 | iTEP | 2 |
| **Manual Test Assistant** | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.5 a) | manual execution | 2 | iTORX | 2 |
| 7.3.5 b) | defect tracking | 2 | iTORX | 2 |

***Tabelle 6-3*** *Framework 1: Basisanforderungen, Teil 2*

| Tool Bridge | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.6 a) | execution code | 2 | HiL-Code Generator | 2 |

| Execution Engine | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.7 a) | sequential execution | 2 | HiL | 2 |
| 7.3.7 b) | identify missing impl. | 2 | HiL | 2 |
| 7.3.7 c) | literals and variables | 2 | HiL | 2 |
| 7.3.7 d) | execution results | 2 | HiL | 2 |
| 7.3.7 e) | timestamps | 2 | HiL | 2 |
| 7.3.7 f) | error recognition | 2 | HiL | 2 |
| 7.3.7 g) | PASS/FAIL | 2 | HiL | 2 |
| 7.3.7 h) | ID for execution | 2 | HiL | 2 |
| 7.3.7 i) | ID for environment | 1 | HiL | 2 |
| 7.3.7 j) | ID for test item | 1 | HiL | 2 |
| 7.3.7 k) | multiple implementation | 0 | HiL | 2 |
| 7.3.7 l) | test results | 2 | HiL | 2 |

| Keyword Library | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.8 a) | basic attributes | 2 | Test*Bench* | 2 |

| Script Repository | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.9 a) | execution code | 2 | Git | 2 |
| 7.3.9 b) | references | 0 | Git | 2 |

| Allgemein/Dokumentation | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.2 a) | composition rules | 1 | Test*Bench* | 2 |
| 7.4.2 b) | parameter description | 1 | Test*Bench* | 2 |
| 7.4.2 c) | parameter passing | 1 | Test*Bench* | 2 |
| 7.4.2 d) | keyword definition | 1 | Test*Bench* | 2 |
| Editor | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.3 a) | syntax checking | 2 | Test*Bench* | 2 |
| 7.4.3 b) | track usage | 2 | Test*Bench* | 2 |
| 7.4.3 c) | check for existance | 2 | Test*Bench* | 2 |
| 7.4.3 d) | mark missing keywords | 2 | Test*Bench* | 2 |
| 7.4.3 e) | exception handling | 2 | Test*Bench* | 2 |
| 7.4.3 f) | auto completion | 2 | Test*Bench* | 2 |
| 7.4.3 g) | versioning | 2 | Test*Bench* | 2 |
| Decomposer und Data Sequencer | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.4 a) | hierarchical keywords | 2 | iTEP | 2 |
| 7.4.4 b) | hierarchical data | 1 | iTEP | 2 |
| Manual Test Assistant | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.5 a) | attachments | 2 | iTORX | 2 |

***Tabelle 6-4*** *Framework 1: erweiterte Anforderungen, Teil 1*

***Tabelle 6-5*** *Framework 1: erweiterte Anforderungen, Teil 2*

| Execution Engine | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.7 a) | access test item | 2 | HiL | 2 |
| 7.4.7 b) | name space | 0 | HiL | 2 |
| 7.4.7 c) | context switching | 1 | HiL | 2 |
| 7.4.7 d) | name space switching | 1 | HiL | 2 |
| 7.4.7 e) | parallel testing | 1 | HiL | 0 |
| 7.4.7 f) | handle blocking | 1 | HiL | 1 |
| 7.4.7 g) | »blocking« attributes | 0 | HiL | 0 |
| 7.4.7 h) | data driven tests | 2 | HiL | 2 |
| 7.4.7 i) | conditional testing | 2 | HiL | 2 |
| 7.4.7 j) | configuration file | 0 | HiL | 2 |
| 7.4.7 k) | multiple configurations | 0 | HiL | 2 |
| 7.4.7 l) | cleanup | 2 | HiL | 2 |
| 7.4.7 m) | allocate information | 2 | HiL | 2 |
| 7.4.7 n) | integration check | 2 | HiL | 2 |
| 7.4.7 o) | finite loops | 2 | HiL | 2 |
| 7.4.7 p) | loops by count | 0 | HiL | 0 |
| 7.4.7 q) | loops by time | 0 | HiL | 0 |
| Keyword Library | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.8 a) | composite keywords | 2 | Test*Bench* | 2 |
| 7.4.8 b) | versioning of keywords | 2 | Test*Bench* | 2 |
| 7.4.8 c) | aliasing | 0 | Test*Bench* | 0 |
| Test Data Support | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.9 a) | versioning of data | 2 | Test*Bench* | 2 |
| 7.4.9 b) | hierarchical data | 2 | Test*Bench* | 2 |
| Script Repository | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.10 a) | versioning of code | 2 | Git | 2 |

### 6.4.2 Framework 2: Robot Framework

#### Steckbrief

Im Projektbeispiel 2 handelt es sich um ein Projekt im Finanzsektor. Die Testobjekte werden durch zwei Teams betreut: Eines ist zuständig für eine Webanwendung für Onlinebanking und die dazugehörige Webanwendung für den Berater in der Bank, und das andere Team kümmert sich um eine REST-API, die genutzt wird, um Konto- und Transaktionsdaten mit Buchhaltungsapplikationen auszutauschen.

Zielstellung war hier, eine ausreichend hohe automatisierte Testabdeckung auch auf der Systemteststufe zu erreichen, um zum Zeitpunkt eines internen Releases eine aus Sicht der Entwicklung fehlerfreie Software an den Abnahmetest liefern zu können. Im Gegensatz zum Projektbeispiel 1 spielt der manuelle Test hier in Bezug auf Keyword-Driven Testing keine Rolle: Die manuellen Abnahmetests wurden von unabhängigen Fachanwendern in einem anderen Bereich durchgeführt. Dennoch sollten die manuellen Testerinnen die Automatisierung verstehen können.

In der Vergangenheit war es in den Projekten immer häufiger zu Kommunikationsproblemen zwischen Entwicklung und Abnahmetest gekommen. Aufgrund der auf JUnit und Java basierenden Testautomatisierung, die von den Abnahmetesterinnen und anderen Stakeholdern nicht verstanden wurde, gab es wenig Kenntnis und entsprechend kaum Vertrauen in die bestehende Testautomatisierung. Zudem wurde wegen eines Technologiewechsels im Testobjekt eine erneute Evaluierung der Automatisierung angestoßen.

Eine detailliertere Beschreibung der Projektsituation und der Umstellung auf Robot Framework wird in Abschnitt 7.2 »Praxisbeispiel« gegeben.

Beide Teams wurden nun aus Testerinnen, Automatisierern und Entwicklerinnen zusammengesetzt. Die Fachtesterinnen sollten mit Robot Framework Tests spezifizieren und die Automatisierer/Entwicklerinnen sollten auf Ebene der Low-Level Keywords bei der Implementierung helfen.

Das Ziel ist es, mit Robot Framework ein ATTD- (Acceptance Test-Driven Development) oder »System Test-Driven Development«-Vorgehen zu etablieren und somit das Wissen der Fachtesterinnen möglichst früh in den Entwicklungsprozess einzubringen.

Als Werkzeuge wird fast ausschließlich Open-Source-Software im Test eingesetzt: Git zur Versionsverwaltung, Jenkins als Continuous Integration System, Robot Framework als Automatisierungsframework, Visual Studio Code als Editor, Playwright für die Browser-

Automatisierung und Python Requests als Automatisierungstechnologie des REST-API.

## Mapping der Tools

Die folgende Tabelle zeigt, wie die verwendeten Tools auf die Elemente aus der Struktur laut ISO 29119-5 abgebildet werden können.

Robot Framework deckt hier einiges ab, hat aber beispielsweise keinen eigenen Editor. Auch hier gilt also, dass erst das Zusammenspiel mehrerer Werkzeuge alle praxisrelevanten Aspekte erfüllen kann. In Abschnitt 6.3.1 »Robot Framework®« kam das schon zur Sprache. Im folgenden Kapitel 7 »Praxis mit Robot Framework« werden wir detailliert auf die einzelnen Komponenten eingehen.

Eine Lücke bleibt beim Robot Framework: das manuelle Testen. Braucht man es, anders als in diesem Beispiel, dann kann man diese Lücke mit Testmanagementwerkzeugen wie zum Beispiel der Test*Bench* schließen.

***Tabelle 6-6*** *Framework 2: Abdeckung der Komponenten aus ISO 29119-5*

| Element aus ISO 29119-5 | Tool/Element im Framework |
|---|---|
| Kernbestandteile | |
| Editor | Visual Studio Code mit RobotCode |
| Decomposer | Robot Framework |
| Data Sequencer | DataDriver |
| Manual Test Assistant | — |
| Tool Bridge | Robot Framework Library API (Python) |
| Execution Engine | Robot Framework |
| Ablage/Repositories | |
| Keyword Library | Resource Files (*.resource) |
| Data Repository | Datentabellen, z.B. CSV, XLSX |
| Script Repository | Python-Code in Git (Git) |

## Bewertung

Bei der nun folgenden Bewertung wird das gleiche Bewertungsschema wie im letzten Abschnitt verwendet und auch hier ist die Bewertung ausschließlich auf den Kontext der beiden Testteams bezogen zu verstehen.

Ziemlich sicher sähen die Tabellen anders aus, wenn dieselbe Person das Framework des zweiten Beispiels im Kontext des ersten Beispiels beurteilt hätte.

*Tabelle 6-7 Framework 2: Basisanforderungen, Teil 1*

| Allgemein/Dokumentation | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.2 a) | keywords | 2 | VS Code | 2 |
| 7.3.2 b) | parameters | 1 | VS Code | 2 |
| 7.3.2 c) | default value | 1 | VS Code | 2 |
| 7.3.2 d) | hierarchies | 1 | VS Code | 2 |
| 7.3.2 e) | data | 1 | VS Code | 2 |
| Editor | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.3 a) | actions in keywords | 2 | VS Code | 2 |
| 7.3.3 b) | go to low-level keywords | 2 | VS Code | 2 |
| 7.3.3 c) | parameters | 2 | VS Code | 2 |
| 7.3.3 d) | comments | 0 | VS Code | 2 |
| 7.3.3 e) | data sources | 1 | VS Code | 2 |
| 7.3.3 f) | references | 2 | VS Code | 1 |
| 7.3.3 g) | order | 1 | VS Code | 2 |
| Decomposer und Data Sequencer | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.4 a) | Decomposer – parameters | 2 | RF | 2 |
| 7.3.4 b) | Data Sequencer – parameters | 2 | DataDriver | 2 |
| Manual Test Assistant | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.5 a) | manual execution | 0 | – | 0 |
| 7.3.5 b) | defect tracking | 0 | – | 0 |

**Tabelle 6-8** *Framework 2: Basisanforderungen, Teil 2*

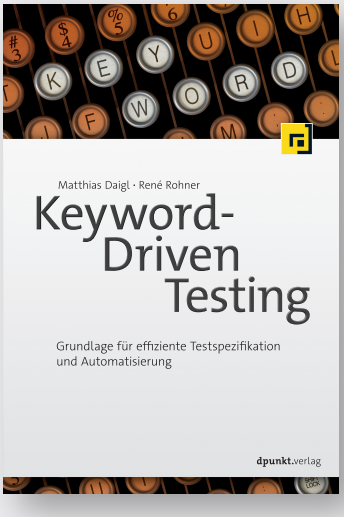
| Tool Bridge | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.6 a) | execution code | 2 | RF | 2 |
| **Execution Engine** | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.7 a) | sequential execution | 2 | RF | 2 |
| 7.3.7 b) | identify missing impl. | 2 | RF | 2 |
| 7.3.7 c) | literals and variables | 2 | RF | 2 |
| 7.3.7 d) | execution results | 2 | RF | 2 |
| 7.3.7 e) | timestamps | 2 | RF | 2 |
| 7.3.7 f) | error recognition | 2 | RF | 2 |
| 7.3.7 g) | PASS/FAIL | 2 | RF | 2 |
| 7.3.7 h) | ID for execution | 1 | RF | 1 |
| 7.3.7 i) | ID for environment | 1 | RF | 1 |
| 7.3.7 j) | ID for test item | 1 | RF | 1 |
| 7.3.7 k) | multiple implementation | 1 | RF | 2 |
| 7.3.7 l) | test results | 2 | RF | 2 |
| **Keyword Library** | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.8 a) | basic attributes | 2 | *.resource | 2 |
| **Script Repository** | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.3.9 a) | execution code | 2 | Python-Code | 2 |
| 7.3.9 b) | references | 2 | Python-Code | 2 |

*Tabelle 6-9 Framework 2: erweiterte Anforderungen, Teil 1*

| Allgemein/Dokumentation | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.2 a) | composition rules | 1 | UserGuide | 2 |
| 7.4.2 b) | parameter description | 1 | UserGuide | 2 |
| 7.4.2 c) | parameter passing | 0 | UserGuide | 2 |
| 7.4.2 d) | keyword definition | 1 | UserGuide | 2 |
| Editor | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.3 a) | syntax checking | 2 | VS Code | 2 |
| 7.4.3 b) | track usage | 2 | VS Code | 1 |
| 7.4.3 c) | check for existence | 2 | VS Code | 2 |
| 7.4.3 d) | mark missing keywords | 1 | VS Code | 2 |
| 7.4.3 e) | exception handling | 2 | VS Code | 2 |
| 7.4.3 f) | auto completion | 2 | VS Code | 2 |
| 7.4.3 g) | versioning | 1 | VS Code | 2 |
| Decomposer und Data Sequencer | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.4 a) | hierarchical keywords | 2 | RF | 2 |
| 7.4.4 b) | hierarchical data | 1 | DataDriver | 2 |
| Manual Test Assistant | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.5 a) | attachments | 0 | — | 0 |

**Tabelle 6-10** *Framework 2: erweiterte Anforderungen, Teil 2*

| Execution Engine | | | | |
|---|---|---|---|---|
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.7 a) | access test item | 1 | RF | 2 |
| 7.4.7 b) | name space | 1 | RF | 2 |
| 7.4.7 c) | context switching | 1 | RF | 2 |
| 7.4.7 d) | name space switching | 1 | RF | 2 |
| 7.4.7 e) | parallel testing | 1 | RF, Pabot | 2 |
| 7.4.7 f) | handle blocking | 1 | RF | 2 |
| 7.4.7 g) | »blocking« attributes | 1 | RF | 0 |
| 7.4.7 h) | data driven tests | 2 | RF | 2 |
| 7.4.7 i) | conditional testing | 1 | RF | 2 |
| 7.4.7 j) | configuration file | 1 | RF | 2 |
| 7.4.7 k) | multiple configurations | 1 | RF | 2 |
| 7.4.7 l) | cleanup | 2 | RF | 2 |
| 7.4.7 m) | allocate information | 1 | RF | 2 |
| 7.4.7 n) | integration check | 1 | RF | 2 |
| 7.4.7 o) | prevent infinite loops | 0 | RF | 2 |
| 7.4.7 p) | loops by count | 1 | RF | 2 |
| 7.4.7 q) | loops by time | 1 | RF | 2 |
| Keyword Library | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.8 a) | composite keywords | 2 | *.resource | 2 |
| 7.4.8 b) | versioning of keywords | 1 | Git | 1 |
| 7.4.8 c) | aliasing | 0 | *.resource | 0 |
| Test Data Support | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.9 a) | versioning of data | 1 | Git | 1 |
| 7.4.9 b) | hierarchical data | 2 | RF, DataDriver | 2 |
| Script Repository | | | | |
| Abschnitt | Attribut | Prio. | Tool | Bewertung |
| 7.4.10 a) | versioning of code | 2 | Git | 2 |




Matthias Daigl • René Rohner

# Keyword-Driven Testing

## Grundlage für effiziente Testspezifikation und Automatisierung

*2022*
*260 Seiten, Festeinband*
*€ 34,90 (D)*

*ISBN:*
*Print 978-3-86490-570-4*
*PDF 978-3-96088-482-8*
*ePub 978-3-96088-483-5*
*mobi 978-3-96088-484-2*

Auf *dpunkt.de* auch als Bundle (Print & E-Book) erhältlich.

Die Autoren bieten einen fundierten Überblick über die technischen und organisatorischen Aspekte des Keyword-Driven Testing und vermitteln das notwendige Praxiswissen, um Keyword-gesteuerte Tests zu erstellen sowie Keywords auszuwählen und zu strukturieren. Auch auf die Herausforderungen und Werkzeuge für das Keyword-Driven Testing wird eingegangen.

Im Einzelnen werden behandelt:

- Unterschiedliche Ansätze für Keyword-Driven Testing
- Auswahl und Strukturierung von Keywords sowie Qualitätssicherung
- Normen im Testen und speziell zu Keywords
- Testautomatisierungsarchitektur
- Keyword-Driven Testing Frameworks
- Praxis mit Robot Framework
- Verbindung mit Testpraktiken wie Test-Driven, Behavior-Driven oder Acceptance Test-Driven Development

dpunkt.verlag
www.dpunkt.de