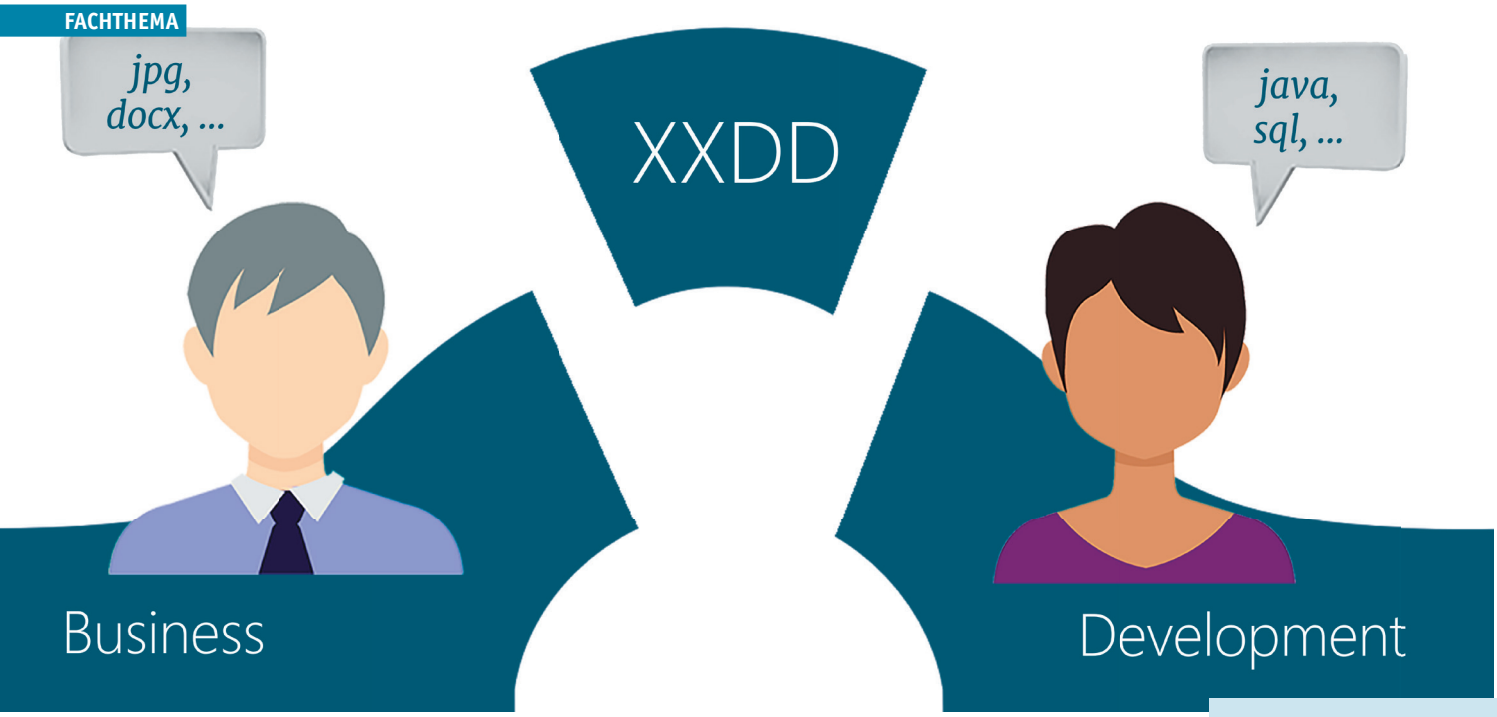


Sonderdruck aus
JavaSPEKTRUM 3/2021

Ausführbare Beispiele als Treiber

XXDD: Fachliche Akzeptanztests der dritten Generation

Reik Oberrath

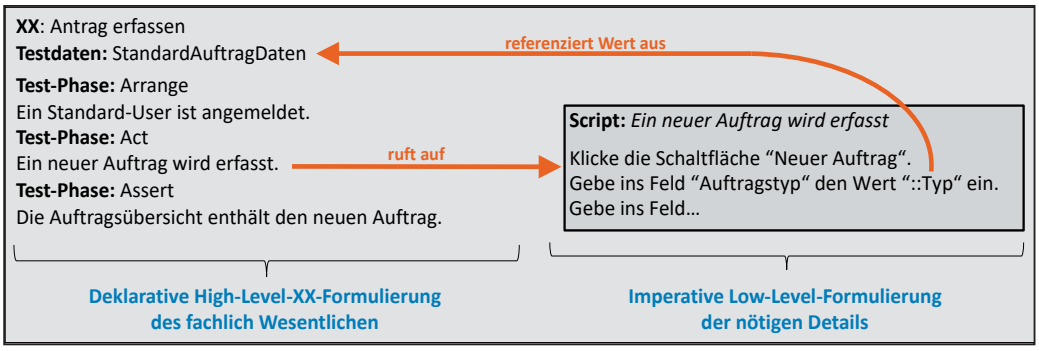
Software wird immer komplexer. Ohne ganzheitliche Betrachtung und viel Automation ist effektive Softwareentwicklung kaum noch möglich. Dieser Artikel stellt die Idee von XXDD vor, die konsequente Weiterentwicklung von ATDD und BDD. XX steht für Ausführbare Beispiele. Als zentrale Drehscheibe und mit der richtigen Automation im Entwicklungsprozess können Ausführbare Beispiele die Softwareentwicklung stark antreiben.

Im Englischen gibt es die schöne Redewendung: *running like a fine-tuned machine*. Die Fragen, was diese Maschine antreibt oder ob sie im Leerlauf rotiert, werden dabei üblicherweise nicht gestellt. Übertragen wir dieses Bild auf unseren Berufsalltag. Als Fachexperte, Anforderungsanalyst, Entwickler, Tester oder Betreiber sind wir alle ein kleines Rädchen in der großen Maschinerie der Softwareentwicklung (SE). Diese mag *fine-tuned* sein, aber was treibt sie an und wie verhindern wir Leerlauf?

Ende der 1990er-Jahre begannen erfahrene Entwickler wie Kent Beck eine neue Programmierweise zu erproben: *Test First* [CCC-a]. Diese Idee ist heute als *Test-Driven Development* (*TDD*) bekannt (s. Abb. 1). Die Tests als Treiber der SE zu verstehen, ist absolut richtig [Wiki]. Bald folgte die Idee des *ATDD* (Acceptance Test-Driven Development) – der Kundenakzeptanztest als neuer Treiber war gefunden. Die SE noch besser antreiben sollte das Behaviour-Driven Development (*BDD*). Dan North beschreibt BDD als *second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology* [Wiki-eng]. Aber was treibt denn im BDD die *fine-tuned machine* der SE an? Antwort: die automatisch ausführbaren Beispiele, im Englischen *Executable Examples*, oder kurz XX.

Wie sich Beispiele ausführen lassen

Beispiele werden in natürlicher Sprache verfasst. Um sie in Programmcode zu übersetzen und auszuführen, gibt es einen ganzen Zoo von ATDD- und BDD-Tools. Die allermeisten funktionieren so: Zuerst definiert ein Entwickler bestimmte Satzschablonen und verknüpft diese mit einem bestimmten Stück Testcode – dann wird das Testtool ausgeführt. Das Tool sucht für jeden Satz im



Listing 1: Die Verwendung von natürlich-sprachlichen Skripten

Reik Oberrath arbeitet seit über 10 Jahren für die IKS GmbH und entwickelt seit über 20 Jahren Geschäftsanwendungen in Java. Seine persönlichen Schwerpunkte dabei sind Softwarequalität, Clean Code, Testautomation und natürlich-sprachliche Oberflächentests.
E-Mail: r.oberrath@iks-gmbh.com

XX eine eindeutig passende Satzschablone und führt den dazugehörenden Testcode aus. Diesen Testcode muss der Entwickler schreiben.

Diese Test-Tools sind also nur Übersetzer, die für bestimmte Sätze bestimmten Testcode aufrufen. Das ist meines Erachtens keine *high-automation*. Ich habe in den letzten sieben Jahren in vier verschiedenen Projekten ein solches Testtool für Geschäftsanwendungen aufgesetzt. Meiner Erfahrungen nach ist viel mehr nötig, um aus natürlicher Sprache ausführbare und wartungsfreundliche Tests zu machen.

Was ein gutes XX-Tool können muss

Schreibt ein Entwickler Testcode für ein ATDD- oder BDD-Tool, dann ist er bei folgenden Punkten allein gelassen:

- Wer passt die Testdaten an, wenn sich diese ändern?
- Wie starte ich die zu testende Anwendung und bediene sie?
- Wie prüfe ich die fachliche Korrektheit?
- Wie erzeuge ich einen Testbericht, der alle fachlich relevanten Details enthält?
- Wie archiviere ich Testberichte?

Ein XXDD-Tool hilft in allen diesen Punkten. Zum Beispiel mit einem Import-Mechanismus für Testdaten. Diese können sich schnell ändern. Manche Testdaten werden von Fachexperten aufwendig zusammengestellt (häufig in Excel-Dateien). Einige Testdaten werden für viele verschiedene Tests gebraucht. Vor diesem Hintergrund sind XX besonders wartungsfreundlich, wenn die Testdaten nicht hart in der Beschreibung der Testausführung stehen, sondern wenn sie dort nur referenziert und bei der eigentlichen Ausführung aus einer externen Quelle gelesen werden können (s. Listing 1).

Ein weiteres Beispiel: Stellen wir uns vor, ein Entwickler hat auf einer Datenmaske ein neues Pflichtfeld programmiert. Einige XX schlagen jetzt fehl, weil der OK-Button auf dieser Datenmaske nicht mehr aktiv wird. Für das neue Pflichtfeld fehlt noch ein Eingabewert. Listing 2 zeigt oben, wie ein klassisches Testtool wie *Cucumber* damit umgeht, und unten, wie es nach XXDD aussehen sollte. Ein Fachexperte oder Anforderungsanalyst hat oben keine Möglichkeit zu sehen, was das Problem ist. Ein Entwickler muss das Problem analysieren und beheben. Unten gibt der Testbericht Auskunft, welcher Wert in welches Feld geschrieben wurde und dass der OK-Button nicht geklickt werden konnte. Mit dieser Information hat ein Fachexperte die Möglichkeit, eigenständig das Problem zu erkennen.

Stellen wir uns weiterhin vor, dass in natürlicher Sprache beschrieben wäre, welches Feld mit welchem Wert zu füllen ist. In diesem Fall könnte der Fachexperte sogar das XX eigenständig anpassen und ohne Entwicklerhilfe das Problem lösen. Geht das? Ja!

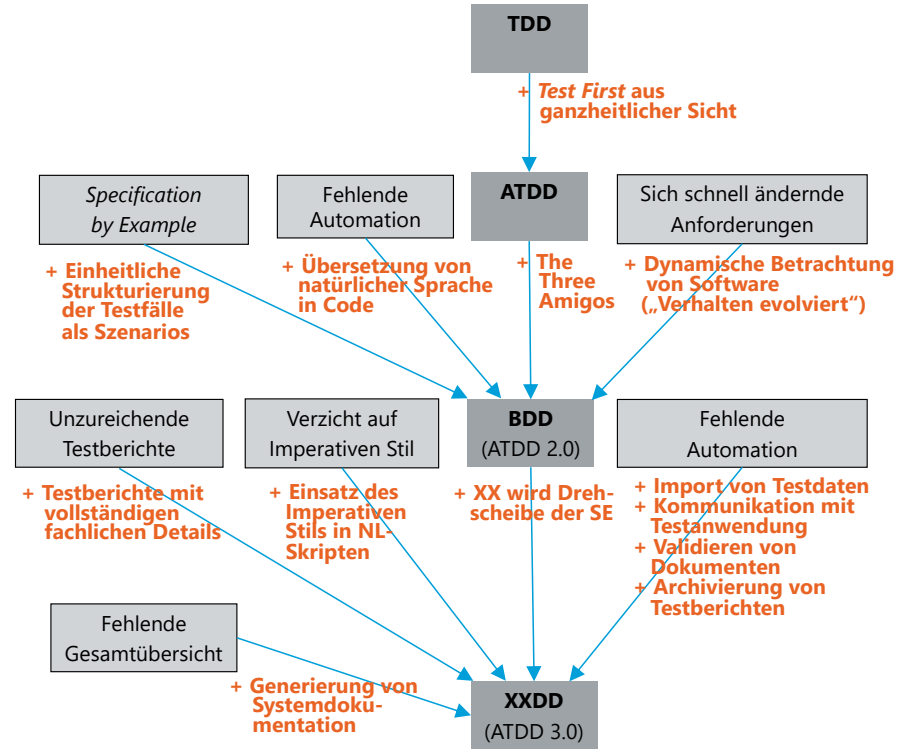
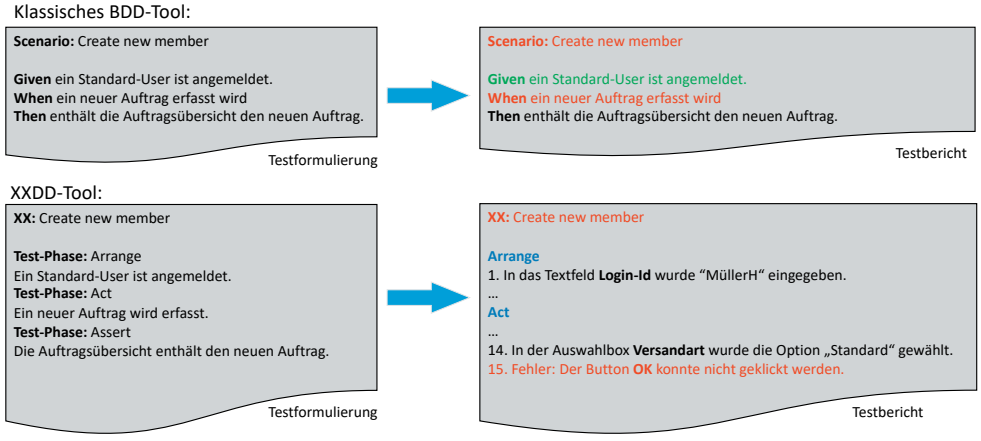


Abb. 1: Generationen der Driven-Development-Methoden



Listing 2: Die Beziehung zwischen Testformulierung und Testbericht

Imperativer Stil vs. Deklarativer Stil

Völlig zu Recht ist es *Best Practice*, Beispiele deklarativ zu beschreiben. Im Unterschied zum imperativen Stil ist die deklarative Formulierung streng auf das fachlich notwendige fokussiert [GitH-b].

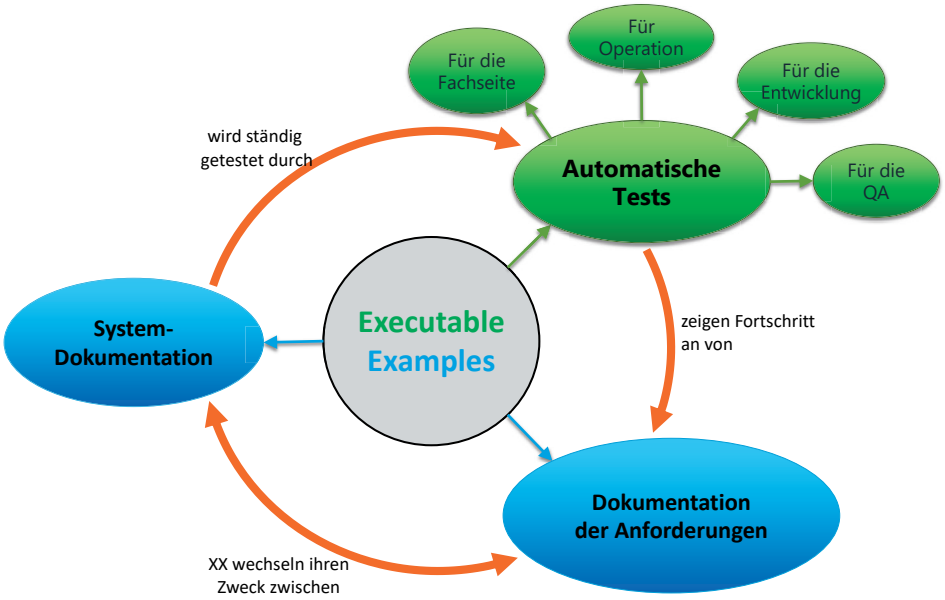


Abb. 2: XX als Drehscheibe der SE

	Deklarativer Stil	Imperativer Stil
Definition	Beschreibung eines Zustands	Liste von Anweisungen, deren Ausführung zu einem bestimmten Zustand führt
Beispiel	User „Hans" ist eingeloggt.	Gebe „Hans" in das Feld „UserId" ein. Gebe „12345" in das Feld „Password" ein. Aktiviere die Schaltfläche „Login".
Eigenschaften	- Kurz und prägnant ☺ - Wenige Details und Einzeldaten ☺ - Abstrakt - Stark kontextspezifisch - Schwer wiederzuverwenden ☹ - Intransparent für Nicht-Entwickler ☹	- Wortreich und ausführlich ☹ - Viele Details und Einzeldaten ☹ - Konkret - Moderat kontextspezifisch - Gut wiederzuverwenden ☺ - Transparent für Nicht-Entwickler ☺

Tabelle 1: Vergleich der Stile für die Formulierung von XXs

Nachteil dabei ist das Fehlen fachlicher Details (s. Tabelle 1). Die Vorteile beider Stile kommen zusammen, wenn natürlich-sprachliche Skripte genutzt werden können. Ein sogenanntes *Natural-Language*-Skript (NLS), stellt eine Sammlung von imperativen Anweisungen dar, die zum Beispiel erklären, wie eine Datenmaske mit Daten zu füllen ist. Jede dieser Anweisung kann aus XX-Sicht als Mini-Schritt betrachtet werden. NLS können in deklarativ formulierten XX aufgerufen werden (s. Listing 1)

Woran kann man ein gutes XXDD-Tool erkennen?

Entwicklern nimmt ein XXDD-Tool beim Programmieren des Testcodes eine Menge Arbeit ab. Für Standardprobleme stehen Lösungen bereit, die der Entwickler *out of the box* nutzen kann und so von vielen technischen Details befreit wird (z. B. das Handling des Testdatenimports, siehe oben). Außerdem können bereits viele Satzschablonen für typische Mini-Schritte zur Verfügung stehen, zum Beispiel „Gebe in Textfeld X den Wert Y ein".

Nicht-technische Anwender versetzt ein XXDD-Tool in die Lage, ohne Entwicklerhilfe viele fachlich bedingte Probleme zu erkennen und selbst zu lösen. Steht eine umfangreiche Bibliothek an Satzschablonen zur Verfügung, kann ein Fachexperte, Anforderungsanalyst oder auch Betreiber seine eigenen XX schreiben und ausführen – ganz ohne Entwicklerhilfe.

All das zusammengenommen nenne ich *high-automation*. Aber ist das nur Theorie? Nein! Das XXDD-Tool *SysNat* [GitH-a] zeigt, dass die genannten Ideen praktisch funktionieren.

XX-zentrierte SE

XX sind immer aktuell. Erst dienen sie als Anforderungen und Tests, die den Fortschritt bei der Umsetzung der Anforderungen deutlich machen (Progressionstests). Dann werden sie zur Systembeschreibung und zu Tests, die prüfen, ob sich das System noch immer wie beschrieben verhält (Regressionstests). Soll sich das bestehende Verhalten einer Anwendung fachlich ändern, werden die XX entsprechend angepasst und aus der/dem Systembeschreibung/Regressionstest wird wieder ein(e) Anforderung/Progressionstest. Die SE ist damit ständig darauf ausgerichtet, die XX und die Software synchron zu halten. Die XX geben dabei den Sollzustand vor. Als zentrale Drehscheibe treiben die XX so die *fine-tuned machine* der SE an (s. Abb. 2).

Fazit

Mit XXDD ist die Softwaredokumentation immer aktuell, weil sie selbst ausführbar ist. Jede Abweichung von Dokumentation und Software ist direkt sichtbar. XX sind natürlich-sprachlich und können ohne technisches Know-how weiterentwickelt und ausgeführt werden. Mit einem XXDD-Tool haben crossfunkionale Teams ein Werkzeug für Systemtests, das jeder Akteur im Team für seine spezifischen Tests nutzen kann. Nur Tests, die mit geringem Aufwand entwickelt, ausgeführt und weiterentwickelt werden, treiben die SE wirklich an. Nur mit einem starken SE-Antrieb können wir die ständig steigende Komplexität in der IT beherrschen [CCC-b].

Links

[CCC-a] http://clean-coding-cosmos.de/testdrivendevelopment-1
[CCC-b] http://clean-coding-cosmos.de/der-entwicklerkosmos/softwarequalitaet-1/softwarequalitaet-6
[GitH-a] https://github.com/iks-gmbh-tools/SysNat/wiki
[GitH-b] https://github.com/iks-gmbh-tools/SysNat/wiki/How-to-formulate-natural-language-instructions%3F
[Wiki] https://en.wikipedia.org/wiki/Test-driven_development
[Wiki-eng] https://en.wikipedia.org/wiki/Behavior-driven_development#History