Autotesting mal anders gedacht

Jan Leßner
S&N Invent GmbH, Zukunftsmeile 2, 33102 Paderborn, jan.lessner@sn-invent.de

Auf die Frage, ob automatisierte Tests eine wichtige Sache in der Software-Entwicklung sind, würden wohl die meisten Entwickler*innen zustimmend nicken. Eine Antwort auf die Frage, was diese Tests denn genau leisten sollen, würde dann vielleicht schon schwerer fallen.

„Ich bin fertig.
ich muss jetzt nur noch die Tests schreiben.“

Dieser Satz begegnet einem in Projekten immer noch recht häufig, und macht gleich mehrere Denkfallen deutlich.

Zum einen ist die Person ganz offensichtlich *nicht* fertig, denn die Tests zu schreiben braucht natürlich auch Zeit. Zum anderen bedeutet dieses *nachträgliche* Schreiben von Tests, das selbige bis zum Abschluss des *produktiven* Codes noch keinerlei Beitrag geleistet haben, um genau diesen schneller zu entwickeln. Es steht zu befürchten, dass die Person bis zu diesem Zeitpunkt in quälend langsamen Edit-Compile-Test-Zyklen vorangeschritten ist und dabei viel Zeit verschwendet hat. Genau die Zeit, die sie besser sofort in die Tests hätte investieren sollen, so dass sie in gleicher Zeit Produktiv- *und* Testcode hätte herstellen können.

Das Dilemma mit isolierten Tests

Zwischen den Zeilen dieses Satzes steckt aber noch ein tiefersitzendes Dilemma: Die Person ist offenbar nicht der Meinung, dass die von ihr noch zu ergänzenden Tests eine wichtige Aussage darüber treffen, ob das implementierte Feature wirklich fertig ist und funktioniert. Das haben wohl vorher schon andere Tests ergeben, wobei das meistens heißt: irgendwie manuell ausprobiert. Was die automatisierten Tests betrifft, ziehen sich Entwickler mit einem Verweis auf die Testpyramide gerne auf stark isolierte „Units under Test“ zurück, die manchmal nur aus einzelnen Klassen bestehen. Das ist im ersten Moment bequem, aber leider vernachlässigen diese Tests die Prüfung einer funktionstüchtigen Kollaboration der Bestandteile, und sie sind auf längere Sicht sogar recht teuer. Das liegt vor allem an Unmengen von Mocks, die zur Umgebungssimulation jeder Testunit implementiert werden müssen. Das kann man mit Mockframeworks vielleicht in wenigen Zeilen Code bewerkstelligen, allerdings nur auf Kosten der langfristigen intuitiven Nachvollziehbarkeit. Und das ist ein echtes Problem, denn diese isolierten Tests sind überdurchschnittlich oft von einer anderen, ebenso wichtigen und permanent stattfindenden Tätigkeit der Qualitätssicherung betroffen: Refactoring.

Kontinuierliches Refactoring ist *der* Schlüssel für dauerhaft wartbare und moderne Software auch über viele Jahre hinweg. Automatisierte Tests wiederum machen Refactoring erst zu einer ungefährlichen Tätigkeit, vorausgesetzt die Tests müssen nicht selbst massiv refactored oder in dem Zuge gar neu geschrieben werden.

Sociable Unittests

Also vielleicht mal anders denken, und weg von der Idee der isolierten Tests als Schwerpunkt. Jay Fields hat in seinem Buch „Working Effectively with Unit Tests“ [1] den Begriff der „Sociable Unittest“ geprägt, die auf größere Units abzielen. Also nicht möglichst *kleine* Units under Test, sondern genau umgekehrt: möglichst *große*. Und da gibt es zumindest eine Art von Units, die eine unbestreitbare Daseinsberechtigung haben, und zwar die System-Usecases, die dem Kunden gegenüber ein Stück verbriefte Funktionalität darstellen. Genau dafür werden Entwickler*innen letztlich bezahlt, und genau für diese sollten die Entwickler selber die Tests herstellen können. Isolation erfolgt im Sinne der XP-Tradition nur dort, wo die Kollaboration mit anderen Bestandteilen sehr hässlich wird oder die Gefahr nicht reproduzierbarer Ergebnisse besteht [2]. Das betrifft vor allem die meisten wirklich externen Systeme, für die man nun also nur noch eine Handvoll Mocks benötigt. Sociable Tests werden zum Hauptaugenmerk und stark isolierte Tests die Ausnahme. Es entsteht das Bild einer Test-Bienenwabe (siehe Abbildung 1), wie André Schaffer sie für eine vergleichbare Problematik erdacht hat [3].

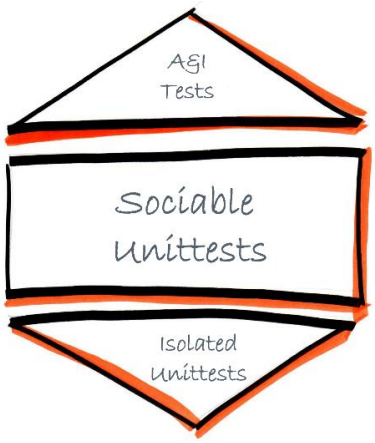

Abbildung 1: Die Testbienenwabe

Leider hat es sich eingebürgert, auch einige zentrale Kollaborationen als „sehr hässlich“ zu betrachten, die mitten im System liegen. Datenbanken werden weggemockt, UI und Backend können nicht miteinander, und Microservices dienen als Entschuldigung dafür, dass mitunter schon Kleinigkeiten nur über „Integrationstests“ prüfbar sind. Die liegen aber meistens außerhalb der Entwicklerverantwortung, sind selten automatisiert und wenn überhaupt, dann oft mit speziellen Werkzeugen wie TestComplete oder ACCELQ. Also nichts, was Entwickler beherrschen und was ihnen hilft, ihre Edit-Compile-Test-Zyklen zu beschleunigen.

Design vor Testing

Um Sociable Testing zu einem Teil der täglichen Arbeit zu machen, gilt es u.U. an einigen Grundfesten der Architektur zu rütteln. Insbesondere bei bestehender Software ist dann fraglich, ob man bis in die letzte Konsequenz vordringt. Aber man kann ja mit kleinen Schritten anfangen, um Motivation für Größeres zu schaffen.

Ein empfehlenswerter erster Schritt zu größeren Tests besteht darin, die Persistenzmedien nutzbar zu machen. Meistens also eine Datenbank, aber z.B. auch Messaging Dienste. Natürlich braucht jedes Teammitglied weiterhin seine eigene Umgebung, d.h. die Persistenzdienste sind lokal installiert. Das mag früher einmal schwierig gewesen sein, aber in Zeiten von Docker sind die meisten gängigen Dienste im Handumdrehen installierbar. In Verbindung mit diesem Schritt ergeben sich aber u.U. noch andere Aspekte, die Neuland darstellen könnten.

Zum einen müssen die Tests für die Nutzung der Datenbank nun einen Großteil des Applikations-Bootstraps durchlaufen, und das muss schnell gehen! Vom Start eines einzelnen Unittests bis zum Erreichen eines Breakpoints im Code sollten es deutlich unter 10 Sekunden sein, sonst ist der Arbeitsfluss gestört. Wenn sich hier nun irgendein Spring Container 30 Sekunden Initialisierungszeit genehmigt, dann heißt es: Ärmel hochkrempeln, analysieren, optimieren, auch wenn man so tief unter der Motorhaube noch nie gearbeitet hat. Nicht einfach schulterzuckend hinnehmen, sonst wächst dem Konzept keine Flügel.

Zum anderen gilt es frühzeitig dafür zu sorgen, dass die persistente Anlage von Geschäftsobjekten leichtgewichtig ist. Bewährt hat sich hier das Builderpattern [4], um die meisten Objekte über Einzeiler anlegen zu können. Und da sich Builder üblicherweise 1:1 aus Entitätenstrukturen ableiten, kann man sich dafür auch mit einfachsten Mitteln Generatoren schreiben oder Plugins wie Lombok verwenden. Wichtig ist, dass das Schreiben von Sociable Tests einfach ist. Es muss leichter sein, einen Test zu schreiben statt die Applikation zu starten, um ein Feature manuell auszuprobieren.

Und wie sieht es Richtung UI aus? Je nach Wahl des Web Frameworks kann es hier ziemlich haarig werden, wenn es darum geht, ohne spezielle Werkzeuge die Tests auf ganze Usecases auszudehnen. Das Prinzip dabei ist der „Headless End-2-End Test“. Die oberste Ebene der Architektur, die zwingend an das Vorhandensein eines Browsers gekoppelt ist, wird möglichst schmal und dumm gehalten. Idealerweise ist sie generiert oder generisch nach den Konzepten einer Naked Objects Architektur [5]. Es reicht aber auch, wenn sie durchgehend kanonischen Mustern folgt, so dass ein unmittelbar unter dieser Schicht ansetzender Test nur wenig an Aussagekraft gegenüber einem UI-Test einbüßt. Die zentrale Frage ist, ob sich ein Sprachbruch vermeiden lässt. Wer ein Java Backend hat und mit GWT, Vaadin oder Apache Wicket arbeitet hat ideale Voraussetzungen für ein Design for Testing bis ins UI hinein. Wer die volle Power von Angular oder Vue.js braucht, sollte schauen, ob er vielleicht Node.js im Backend verwenden kann. Es lohnt sich also vielleicht für das nächste Projekt, im Sinne der Testbarkeit und effizienten Entwicklung auch mal wieder ganz andere Ideen zuzulassen.

Referenzen

[1] Jay Fields, Working Effectively with Unit Tests, Lean Pub, 2015

[2] https://martinfowler.com/bliki/UnitTest.html

[3] https://engineering.atspotify.com/2018/01/testing-of-microservices/

[4] https://www.digicomp.ch/blog/2017/12/07/das-builder-pattern

[5] https://en.wikipedia.org/wiki/Naked_objects