Java magazin™

Java | Architektur | Software-Innovation

SPRING RÄUMT AUF

Spring Boot 4.0 und Spring Framework 7.0

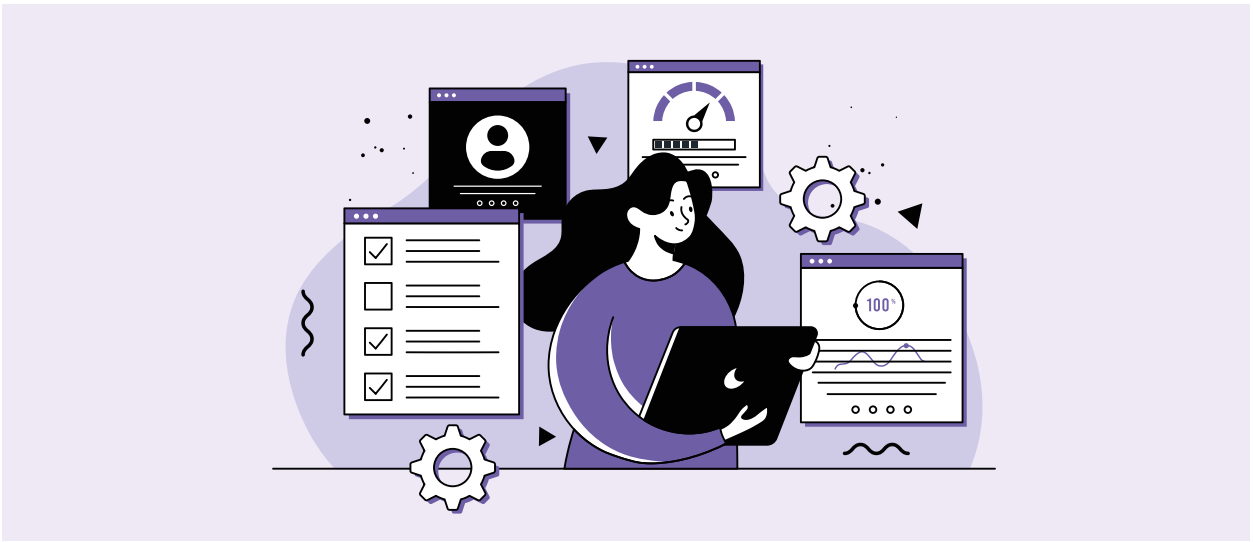
Coverbild erstellt mit ChatGPT (DALL·E)

Ausgabe 2.2026

Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

4 194586 709801 02

entwickler.de



Wie man schnell, wertvoll und kosteneffizient testet

Emergente Testarchitektur

Moderne Softwareentwicklung ist ohne automatisierte Tests nicht vorstellbar. Neu gebaute Software muss schnell, zuverlässig und kosteneffizient validiert werden. Nur – woher weiß ich, was kosteneffizient ist und was nicht? Wie entscheide ich, welche Tests ich schreiben sollte? Mit Scheuklappen eine Testpyramide hochzuziehen, bringt oft nicht die erhoffte Erfolgsgarantie, Gleiches gilt für ungebremste Unit-Test-Euphorie. Zentral ist die Architektur unserer Testsuites: Warum muss sie emergent sein, wie kann man sie bewerten und aus welchen Teilen besteht sie möglicherweise?

von Moritz Tiedje

Warum schreiben wir Tests? Weil effiziente agile Softwareentwicklung viel Feedback braucht, und das nicht nur im Produktmanagement, sondern schon viel früher und häufiger in der täglichen Entwicklerarbeit. Eine agile Methode, die diese Erkenntnis sehr klar verinnerlicht hat, ist das Extreme Programming (XP). Dort gibt es die Planungs-/Feedback-Schleifen, die in **Abbildung 1** zu sehen sind.

Dieses Bild zeigt: Wir bekommen unser erstes Feedback von unserem Pairing-Partner schon beim Schreiben des Codes. Wenige Minuten später erfahren wir mittels Unit-Tests, ob etwas kaputt gegangen ist und ob unser neuer Code das tut, was er tun soll. Einmal am Tag fragen wir uns im Stand-up-Meeting, ob wir noch auf dem richtigen Weg sind, um unsere Ziele zu erreichen. Ist nach einigen Tagen ein Feature fertig gebaut, folgt der Akzeptanztest, in dem Kunden, Fachexperten o. Ä. die fertige Software ausprobieren und prüfen, ob sie wunschgemäß funktioniert. Nach einigen Wochen wird inspiziert und kontrolliert, wie gut der bisherige Iterationsplan eingehalten wurde, ob es relevante Entwicklungen am Markt gegeben hat oder ob andere Fak-

toren aufgetreten sind, die eine Reaktion erfordern. Bei Bedarf werden die Pläne für die nächsten Iterationen und Releases entsprechend angepasst. Wichtig ist hier noch, dass das gesammelte Feedback am Ende zurück in den Code fließen muss und so auf das nächste fertige Software-Inkrement einzahlt.

Seit den Anfängen von XP sind Jahrzehnte vergangen. Dass wir viele Feedbackschleifen nutzen, hat sich aber bewährt. Wenn wir uns nur auf Scrum oder ein anderes agiles Framework beschränken und auf automatisierte Tests und XP-Praktiken verzichten, bekommen wir mit hoher Wahrscheinlichkeit wertvolles Feedback viel zu spät und/oder es kommt uns teurer. Der Hauptzweck der automatisierten Tests ist darum, Feedback zu erhalten. Es gibt weitere Vorteile, die sich daraus ableiten oder nützliche Nebeneffekte bilden:

- Weniger bis kein manueller Testaufwand.
- Tests als lebendige Dokumentation und Anforderungsdefinition, die „rot“ wird, wenn sich der dokumentierte/getestete Systemaspekt nicht so verhält wie beschrieben.
- Deutlich leichtere Wartbarkeit und Erweiterbarkeit des Systems.

Es würde den Rahmen dieses Artikels sprengen, alle Vorzüge der und Gründe für die Testautomatisierung aufzuzählen. Tests zu schreiben hat sich aus guten Gründen fast überall etabliert, wo Software entwickelt wird.

Wie erkenne ich einen guten Test?

Nicht jedes Feedback ist gleich gut. Wir können es sehr konstruktiv äußern oder auch nicht. Ein Beispiel: Wenn mir jemand zu diesem Artikel Feedback geben möchte, dann kann das „Schrott“ lauten oder: „Im Abschnitt X ist das Argument Y missverständlich. Was hältst du von Anpassung Z?“ Ähnlich verhält es sich mit Tests. „Irgendwo in Tausenden Zeilen Code funktioniert etwas nicht so wie es soll“ ist weniger hilfreich als: „In dieser Klasse und in dieser Methode wird bei diesem Aufruf X erwartet, aber Y geliefert“. Testfeedback muss wertvoll sein. Das bedeutet für automatisierte Tests, dass sie drei Anforderungen erfüllen müssen: Erstens darf es keine False Positives oder False Negatives geben. Zweitens muss leicht nachvollziehbar sein, welches Szenario der Test überprüft, und drittens muss transparent werden, was nicht stimmt, wenn der Test fehlschlägt.

Außerdem sind Tests ein integraler Teil des Entwicklerworkflows. Gute Tests sind so schnell, dass sie meine Arbeit nicht behindern. Wenn die Testsuite in Sekunden durchläuft, dann kann ich sie häufig ausführen und damit Defekte schnell entdecken. Wenn sie Minuten braucht, dann werde ich sie immer noch mehrmals am Tag ausführen, beispielsweise, wenn ich eine Kaffeepause machen möchte oder eine Codeänderung integrieren will. Sollte die Testsuite dagegen Stunden brauchen, dann kann ich nicht mehr fokussiert arbeiten. Mein Workflow ist dann: „Arbeite an Thema 1“ -> „starte Tests“ -> „arbeite an Thema 2“ -> „prüfe Testergebnis Thema 1“ -> „zurück zu Thema 1“. Testfeedback muss also schnell verfügbar sein.

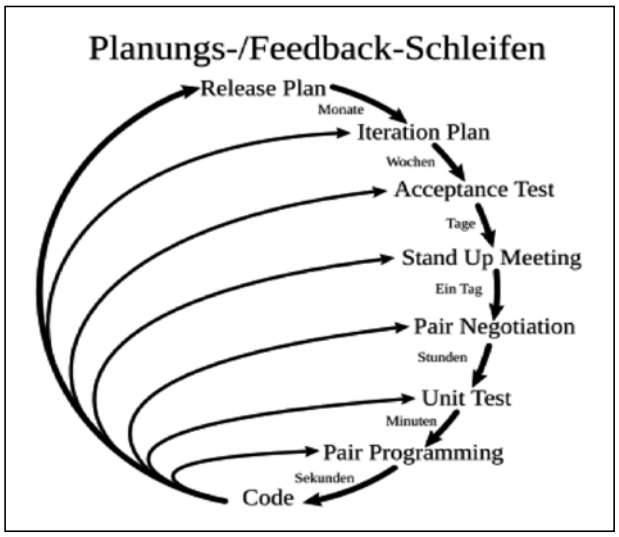


Abb. 1: Planungs-/Feedback-Schleifen in XP (Bildquelle [1])

Schließlich können Tests auch sehr teuer sein. Dem Akronym YAGNI, „You Aren’t Gonna Need It“ [2] sind vier Costs zugeordnet: Cost of Build, Cost of Delay, Cost of Carry und Cost of Repair. In der Testautomatisierung treten diese vier Kostentypen ebenfalls auf:

- *Cost of Build*: Es kostet, einen Test zu bauen.
- *Cost of Delay*: Es kostet, wenn andere Features dafür erst später an den Markt kommen.
- *Cost of Carry*: Es kostet, die zusätzliche Komplexität des Tests im Produkt zu tragen und auch, die Infrastruktur für den Test zu betreiben.
- *Cost of Repair*: Es kostet, einen Test zu warten.

Das bedeutet nicht, dass wir uns jeden Test sparen sollen, weil wir „ihn nicht brauchen werden“. Technische Schulden und Defekte auf Produktivsystemen verursachen Kosten, die exponentiell höher und gefährlicher sind, als es die Kosten einer Testsuite je sein können. Darum empfehle ich auch, im Zweifelsfall eher ein bisschen zu viel als zu wenig in Codequalität und Testautomatisierung zu investieren. Trotzdem besteht eine klare Obergrenze – Testautomatisierung ist kein Selbstzweck: Wir müssen unser Feedback kosteneffizient erhalten können. Zusammengefasst: Ein guter Test gibt schnelles, wertvolles und kosteneffizientes Feedback.

Was hat das Schreiben von Tests mit Softwarearchitektur zu tun?

Es gibt viele verschieden Arten von Tests, weit mehr als nur den herkömmlichen Unit-Test:

- Wir können das funktionale Verhalten eines Systems testen in Unit-Tests, Integrationstests, Akzeptanztests, UI-Tests, Screenshot-Tests, E2E-Tests, Contract-Tests …

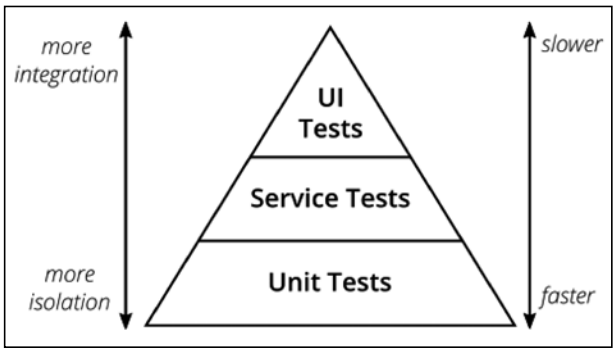


Abb. 2: Die Testpyramide (Bildquelle [3])

- Wir können die innere Qualität testen, indem wir Code Smells, Linting, Test-Coverage, Mutation Coverage … messen (Kasten „Qualitätsmetriken“).
- Wir können Performance und Service-Level-Agreements automatisiert testen.
- Wir können Security mit Vulnerability Scans, automatisierten Pen-Tests etc. testen (Kasten „Qualitätsmetriken“)

Für jede der vielen verschiedenen Arten der Testautomatisierung, die wir bauen können, steht auch eine Vielzahl an Technologien, Frameworks und Third Party Libraries zur Verfügung. Diese Menge an Möglichkeiten bedeutet, dass wir Architekturentscheidungen treffen müssen. Solche Entscheidungen sind zwar nicht in Stein gemeißelt, lassen sich aber oft nur kostenaufwendig korrigieren. Außerdem werden sie berücksichtigt, wenn weitere Entscheidungen anstehen, und wirken sich daher langfristig aus.

Wie sieht die ideale Testarchitektur aus?

Wenn wir in die Literatur schauen, finden wir erst mal die Testpyramide (**Abb. 2**). Sie kommt oft und in verschiedenen Publikationen vor, mit einer variierenden Anzahl von Schichten und deren Bezeichnungen. Grob zusammengefasst besteht eine Testpyramide unten aus kleineren Tests, die zahlreicher, isolierter, schneller, billiger und technischer sind, und oben aus größeren Tests, die weniger zahlreich, integrativer, langsamer, teurer und fachlicher sind. Tests für Security, Performance oder Qualität werden in der Testpyramide meistens nicht berücksichtigt.

Es gibt auch andere Architekturmodelle, zum Beispiel die „Test-Honeycomb“ [4] oder die „Test-Trophy“ [5]. In diesem Artikel können wir nicht auf alle diese Architekturen im Detail eingehen. Jede kann in sich selbst eine wertvolle Hilfestellung sein. Eine Gefahr, die ich in Projekten beobachtet habe, besteht jedoch darin, dass diese Architekturmodelle den eigentlichen Zweck der Testarchitektur überschatten können. Eine Testarchitektur ist dann gut, wenn die Testsuite schnell, wertvoll und kosteneffizient Feedback gibt, nicht, wenn sie sich so nah wie möglich an der Testpyramide orientiert. In meinen Vorträgen werden mir häufig Fragen gestellt wie „Wie viel Prozent meiner Tests sollten Unit-Tests sein?“ (Mehr als 0 %.) oder „Wie viel Code-Coverage sollte eine E2E-Testsuite haben?“ (Optimiert Tests auf fachliche Dokumentation und Aussagekraft und nicht auf Coverage, wenn ihr weit oben in der Testpyramide unterwegs seid.) Meine Antworten auf solche Fragen sind immer eine Version der alten Formel „Es kommt drauf an“. Ein Beispiel: Ein gepanzertes Militärfahrzeug im normalen Straßenverkehr wäre völlig unsinnig, das Gleiche gilt aber umgekehrt für einen Kleinwagen im Kriegsgebiet. Ähnlich ist es mit Testarchitekturen. Was ein Produkt und ein Team an automatisiertem Feedback braucht, kann sehr individuell sein. Zum Beispiel sind die Herausforderungen in einer verteilten Microservices-Landschaft mit harten Performanceanforderungen ganz anders als in einem Legacy-Monolithen, der saniert werden soll.

Emergente Testarchitektur

Die Kunst der emergenten Testarchitektur liegt in der Balance zwischen zwei Planungsextremen. Das eine Extrem ist bekannt als Accidental Architecture. Projekte mit einer Accidental-Testsuite-Architektur beschäftigen

Qualitätsmetriken

Innere Qualität ist etwas inhärent Subjektives. Code ist dann einfach wart- und erweiterbar, wenn Entwickler ihn einfach warten und erweitern können, nicht, wenn Qualitätsmetriken „grün“ sind. Qualitätsmetriken können ein hilfreiches Hinweis- und Verbesserungstool für Entwickler sein. Ich kenne aber eine Myriade grusliger Geschichten von Projekten, in denen Qualitätsmessungen mit Qualität verwechselt wurden und dann entsprechendes Metric Fitting den Code verschlimmbesserte. Die Illusion von Qualität stand dabei dem Inspect-and-Adapt-Prozess erheblich im Weg.
Für Security-Messungen gilt Ähnliches. Bitte verfallt nicht der Illusion von Sicherheit, nur weil ein statisches Analysetool sagt, dass alle Dependencies aktuell seien.

Cargo Cult

„Cargo Cult“ ist eine Metapher und bezieht sich auf einen Mythos angeblicher Kultanhänger im afrikanischen Dschungel, die Cargo-Flugzeuge ohne Motor aus Bambus nachgebaut haben in der Hoffnung, dass sie dann fliegen können. Es gab echte Cargo-Kulte (in Melanesien, nicht in Afrika), die manchmal solche Flugzeug-Nachahmungen gebaut haben. Das war aber eher eine Form des Schamanismus und der religiösen Ahnen-Huldigung, als dass irgendjemand geglaubt hat, dass hier ein funktionsfähiges Flugzeug entsteht. Ich benutze die Metapher trotzdem gern, weil sie in der Softwarewelt vielen geläufig ist: Das Bambus-Flugzeug ohne Motor im Inneren ist wie die voll ausgearbeitete Architektur ohne agile Werte im Inneren. Der Fokus der Arbeit liegt darauf, dass es wie ein Flugzeug aussieht und nicht, dass es fliegen kann.

sich mit dem Thema Testarchitektur ganz einfach, nämlich gar nicht. Symptom einer Accidental Architecture sind auffällige Probleme und Mängel, die über die Jahre immer schlimmer werden statt besser. Die Testsuite, die vier Stunden lang rechnet, braucht irgendwann fünf Stunden, dann sechs. Unzuverlässige Tests, die ohne guten Grund fehlschlagen, werden mittels Copy and Paste zahlreicher, nicht seltener. In dem Projekt findet in der Testarchitektur kein nennenswertes Inspect and Adapt statt, weil das Know-how fehlt oder die Konsequenz im Produktmanagement oder das Verantwortungsbewusstsein. Oft liegt es auch an einer Kombination aller drei Gründe.

Das andere Extrem nenne ich gerne den „Architekturplan-Cargo-Cult“ (Kasten „Cargo Cult“) Es gibt einen Plan, der eingehalten werden muss, die Tests könnten aber schneller, wertvoller und kosteneffizienter sein. Was einen Architekturplan-Cargo-Cult auszeichnet, ist die Tatsache, dass der Plan den eigentlichen Zweck der Testautomatisierung als Ziel verdrängt hat. Ein Symptom davon ist, dass es keine Experimente mehr gibt. Oft findet man hier auch hart durchgesetzte Regeln wie „In den statischen Tests darf kein einziger Code Smell gemessen werden“ oder „Jede Klasse muss eine Test-Coverage von mindestens 80 % haben“. Inspect and Adapt ist hier nur mit vielen Konflikten möglich. Von der bestehenden Architektur und dem etablierten goldenen Pfad zur Softwarequalität abzuweichen, wird als Häresie betrachtet und sofort unterbunden.

Accidental Architecture und Architekturplan-Cargo-Cult sind zwei Extreme. In den meisten Projekten findet man weder das eine noch das andere, sondern irgendein Zwischending. Ich führe die Extreme aber gern auf, um zu illustrieren, das weder absolute Planlosigkeit noch absolut strikte Vorgaben erstrebenswert sind.

In der emergenten Testarchitektur geht es darum, eine angemessene Balance zu halten. Wir wollen Architekturentwürfe wie die Testpyramide als Hilfestellung nutzen, schon allein, um nicht von null an eine eigene Testarchitektur erarbeiten zu müssen. Ziel unserer Arbeit soll aber wertvolles, schnelles und kosteneffizientes Feedback sein, nicht eine perfekte Testpyramide. Ab und zu sollten Experimente möglich sein, in denen wir neue Technologien, Architekturen und Ansätze ausprobieren.

Ziel unserer Arbeit soll wertvolles, schnelles und kosteneffizientes Feedback sein, nicht eine perfekte Testpyramide.

Emergente Testarchitektur: Inspect and Adapt

Wie gut oder schlecht der aktuelle Architekturansatz funktioniert, ist oft schwer zu bewerten. Gute Testarchitektur hat viele Aspekte; sie alle gleichzeitig diskutieren zu wollen führt oft zu Chaos, anstrengenden, fast endlosen Auseinandersetzungen und letztendlich wenig Veränderung. Die Übersicht in Tabelle 1 enthält aus der Praxis entstandene Kriterien, um Bewertungen von Testarchitekturschichten besser zu strukturieren.

Es ist schwer, klare Vorgaben für die Anpassung („Adapt“) einer Testarchitektur zu geben. Wenn im Inspect-Schritt sinnvolle Ergebnisse entstanden sind, sollten aber bereits ein paar offensichtliche Adapt-Schritte sichtbar geworden sein. Eine simple Adapt-Heuristik ist „Accept, Mitigate, Eliminate“:

- *Accept*: Kosten und Nutzen dieser Testsuite stehen in einem angemessenen Verhältnis zueinander.
- *Mitigate:* Wir wollen etwas anpassen, damit die Suite schneller, kosteneffizienter und/oder wertvoller wird. Diese Mitigationen setzen oft Know-how und Expertenwissen voraus. Zum Beispiel könnte man die Cost

Schnell	
Repeatability	Wie oft führen wir die Tests aus?
Time	Wie lange warten wir auf Ergebnisse?
Wertvoll	
Stability	Wie zuverlässig laufen die Tests?
Coverage	Was decken wir mit dieser Suite ab?
Useful Failures	Schlagen die Tests aus hilfreichen Gründen fehl?
Debugability	Wie leicht finden wir den entsprechenden Defekt, wenn ein Test fehlschlägt?
Readability	Wie gut kann das Verhalten des Systems anhand des Tests nachvollzogen werden?
Kosteneffizient	
Cost of Build	Wie viel Implementieraufwand kostet uns diese Testschicht?
Cost of Carry – Cognitive Load	Wie viel komplizierter wird das System mit dieser Testschicht?
Cost of Carry – Infrastructure	Welche Infrastruktur wird benötigt und wie teuer ist sie?
Cost of Repair	Wie oft müssen wir die Tests warten? Wie teuer ist das?

Tabelle 1: Bewertung einer Testarchitektur

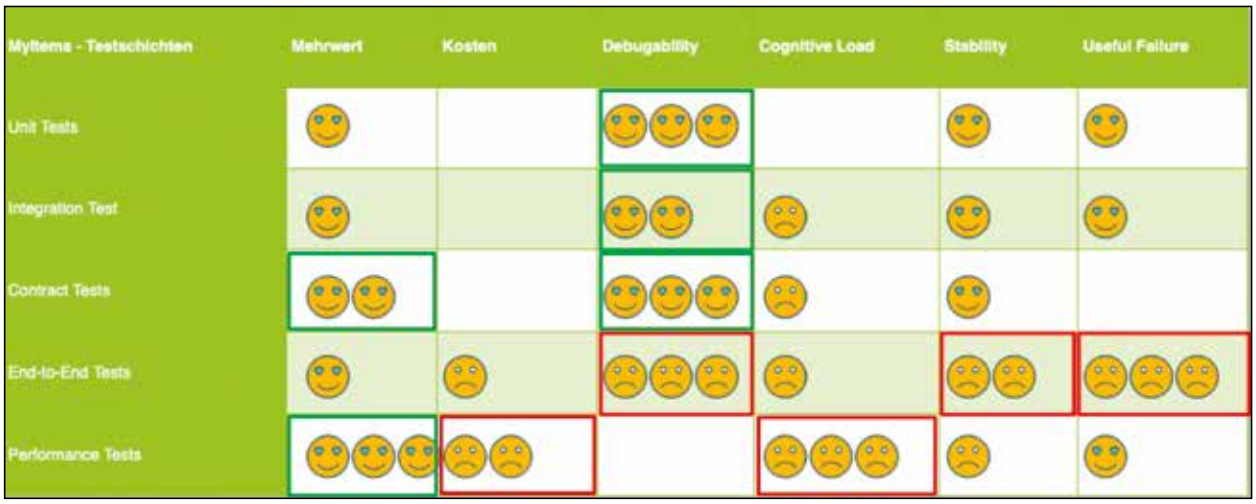


Abb. 3: Beispiel für die Bewertung einer Testarchitektur

of Repair einer UI-Testsuite deutlich reduzieren, indem man Page Objects anstelle von statischen Pfaden benutzt. Oder man könnte die Dauer einer Spring-Integrations-Testsuite reduzieren, indem man den gesamten Spring-Kontext seltener und nicht immer komplett hochfährt.

- *Eliminate*: Die Suite ist zu langsam, zu wertlos und/oder zu teuer, um eine Existenzberechtigung zu haben. Wir werden sie ersetzen, archivieren oder löschen.

Beispiel aus der Praxis

In der Praxis ist es nicht sinnvoll oder pragmatisch, alle Aspekte aus Tabelle 1 ausgiebig zu bewerten. Man kann sich die Fragen aussuchen, die relevant erscheinen, sich bei den Entwicklern für jede Testschicht ein erstes Stimmungsbild abholen und die nicht offensichtlichen Kandidaten im Team durchsprechen. Mit ein bisschen visueller Unterstützung erhält man dann ein Ergebnis, das helfen kann, sinnvolle Entscheidungen zu treffen. **Abbildung 3** zeigt ein anonymisiertes Beispiel einer Testarchitekturbewertung aus einem meiner Projekte.

In diesem Beispiel gab es drei von fünf Testschichten, an die wir schnell ein „Accept“ setzen konnten: Unit-Tests, Integrations- und Contract-Tests. End-to-End-Tests brachten in diesem Projekt zwar einen einzigartigen Mehrwert, waren aber derart aufwendig, dass das Kosteneffizienzkalkül nicht aufging. Daher trafen wir die Entscheidung, die End-to-End-Testsuite nach und nach zu reduzieren und mehr in Contract- und Integration-Tests zu investieren. Die Performance-Tests waren in diesem Projekt unverzichtbar, wie man an der Spalte „Mehrwert“ erkennt. Der Cognitive Load war jedoch so hoch, dass das Problem trotzdem mitigiert werden musste. Wir entschieden, die Performance-Tests von Scala in Java umzuschreiben, da Java ohnehin schon weitläufig im Projekt in Verwendung war.

Fazit

Test sind kein Selbstzweck oder ein Nice-to-Have, sondern ein fundamentaler Feedback-Mechanismus. Unsere Tests müssen dieses Feedback schnell, wertvoll und kosteneffizient liefern. Auf die Frage, welche Testschichten und welche Testarchitektur richtig sind, gibt es keine allgemeingültige Antwort. Wichtig ist, dass die Testarchitektur emergent ist. Das bedeutet, dass wir dann Entscheidungen treffen, wenn wir es müssen, und dabei eine Balance suchen zwischen den beiden Extremen, immer nur das Erstbeste zu nehmen (Accidental Architecture) oder alles im Voraus festzulegen (Architektur-Cargo-Cult).

Um das zu erreichen, müssen wir uns bewusst Zeit für Entscheidungen nehmen und strukturiert vorgehen. In diesem Artikel gebe ich eine Struktur vor, von der man jederzeit abweichen darf, so wie im geschilderten Praxisbeispiel. Mit ein wenig Inspect and Adapt zum richtigen Zeitpunkt lässt sich so gegebenenfalls eine Menge vorhersehbarer Ärger und Stress mit Testautomatisierung vermeiden.

Moritz Tiedje ist seit 2015 professioneller Softwareentwickler bei andrena objects und beschäftigt sich mit allem, was hilft, Software effizienter und wertvoller zu entwickeln. Dabei konzentriert er sich hauptsächlich auf agile Software-Engineering-Praktiken und Softwarearchitektur. Auch verwandten Bereichen wie agile Frameworks, Coaching-Methoden, IT-Security, Moderationstechniken, Teamdynamiken usw. gilt sein Interesse. Seine Projekte haben ihm gezeigt, dass Herausforderungen selten nur technischer Natur sind.

Links & Literatur

[1] Beck, Kent: „Extreme Programming Explained: Embrace Change“, Addison-Wesley, 2004

[2] Fowler, Martin: „YAGNI“: https://martinfowler.com/bliki/Yagni.html

[3] Vocke, Ham: „Test Pyramid“: https://martinfowler.com/articles/practical-test-pyramid.html

[4] Schaffer, André: „Testing Microservices“: https://engineering.atspotify.com/2018/01/testing-of-microservices

[5] Dodds, Kent C.: „Test Trophy“; https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications