Ausgabe 04/2020

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 04

Java aktuell

iJUG Verbund
www.ijug.eu

Java 14
Einblick in die neuen Features

Testing
Neue Impulse für das effektivere Testen von Software

Testautomatisierung
Sind automatisierte Tests eine Illusion?

Liebe Leser der Java aktuell,

in dieser Ausgabe widmen wir uns dem Themenschwerpunkt „Testing". Das Testen einer Anwendung ist ein elementarer Bestandteil der Softwareentwicklung und stellt sicher, dass die vorgesehenen Anforderungen funktionieren. Dafür stehen dem Entwickler eine Vielzahl von Methoden, Tools und Ansätzen zur Verfügung. In dieser Ausgabe der Java aktuell beleuchten wir einige davon näher.

Den Anfang macht Sebastian Daschner mit seinem Artikel „effizientes Testen mit Enterprise Java" auf Seite 14. Darin gibt er einen Überblick darüber, welche Arten von Tests es gibt und wie diese effektiv eingesetzt werden können. Danach beschäftigt sich Thomas Papendieck mit Testautomatisierung und erläutert seine These, dass Testautomatisierung eine Illusion sei. Der Frage, ob eine Testabdeckung von einhundert Prozent erstrebenswert und sinnvoll ist, widmet sich Dr. Roger Butenuth ab Seite 26 in seinem gleichnamigen Artikel. Weiterhin finden Sie in dieser Ausgabe spannende Artikel zu Testcontainers, Ansible Playbooks und Side-by-Side-Refactoring. Außerdem zeigt Ihnen Armin Schubert, wie Sie für alle Teilnehmer sinnvolle Meetings gestalten können – auch aus dem aufgrund der Corona-Krise momentan weit verbreiteten Home-Office. Zum Abschluss dieser Ausgabe zeigt Ihnen Sabine Wojcieszak, warum Sie in gewissen Situationen im Berufsalltag auch einfach Mal „Nein" sagen sollten, und erläutert die teils schwerwiegenden Auswirkungen eines unüberlegten „Ja" anhand eines Beispiels.

Im März 2020 gab es ein neues Java-Release. Werfen Sie gemeinsam mit Falk Sippach ab Seite 9 einen Blick auf die neuen Features! Im Java-Tagebuch und in der Eclipse Corner halten wir Sie wie gewohnt auf dem Laufenden, was aktuell in der Java- und Eclipse-Community passiert.

Wir wünschen Ihnen viel Spaß beim Lesen!

Ihre

Lisa Damerow
Redaktionsleitung Java aktuell

Inhalt

Ansätze und Technologien für effektiveres Testing

Die Grenzen der Testautomatisierung

Fortgeschrittenes Refactoring von Legacy Code

Tipps und Tricks für sinnvolle Meetings – auch aus dem Home-Office

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java.

12. Februar 2020

Vorschlag für Jakarta-/MicroProfile-Zusammenarbeit

Die Wege der Zusammenarbeit (oder: die Abgrenzung) zwischen MicroProfile (MP) und Jakarta EE (jetzt beide bei der Eclipse Foundation zuhause) sind weiterhin nicht geklärt. Ein neuer Vorstoß kommt von Steve Millidge von Payara in der MP-Mailingliste. „MicroProfile specs become Jakarta EE specs; MicroProfile becomes a profile within Jakarta EE; MicroProfile APIs move to the Jakarta namespace when MicroProfile adopts Jakarta EE 9 as a baseline; Jakarta EE will need to change some to support this."

Speziell das („unnötige") Umbenennen von Package-Namen stößt auf wenig Gegenliebe bei den Vertretern von Red Hat, IBM und Co. Mark Little weist zusätzlich darauf hin, dass für MP von Beginn an vorgesehen war, die Java-/Jakarta-EE-Spezifikationen zu nutzen, aber Innovation und Standardisierung darüber hinaus zu betreiben (beispielsweise mit OASIS oder inzwischen der CNCF). Reza Rahman (früher Oracle, jetzt Microsoft) und Oracles Rudy Busscher sehen insbesondere in der vorgeschlagenen Zusammenfassung der Projekte in einer einzigen Working Group eine große Chance. Schwer zu sagen, ob das persönliche Meinungen sind oder sich darin Firmenpolitik widerspiegelt. Die Diskussion ist erst einmal abgeflaut, aber sie kommt wieder... Wer sich für die Details interessiert, kann die Diskussion nachlesen *[1]* (und sich aktiv beteiligen).

19. Februar 2020

Eclipse MicroProfile 3.3 aktualisiert fünf APIs

MicroProfile 3.3 ist gestern freigegeben worden. Das Release bietet Aktualisierungen von fünf APIs: Config (1.4), Fault Tolerance (2.1), Health (2.2), Metrics (2.3) und Rest Client (1.4). Wie die Versionsnummern suggerieren: keine Quantensprünge, aber viele kleinere Verbesserungen und Fixes.

20. Februar 2020

GraalVM 20.0

Die GraalVM hat mit der Umstellung der Versionierung einen ganz anderen Sprung (von 1 auf 19) hinter sich und ist nun im Jahr 2020 angekommen. Die vorgestern freigegebene Version 20.0.0 bietet jetzt unter anderem vollwertige Unterstützung für Windows, inklusive des Werkzeugs „gu" zur Installation von Komponenten, sowie der neuen JavaScript Engine. Node.js-Unterstützung auf Windows soll allerdings erst mit 20.1 folgen. Update (11.3.): Zur Abkündigung der Nashorn-Engine im OpenJDK (die mit Java 15 im Herbst umgesetzt werden soll) und dazu, wie die GraalVM als Rettung dienen kann, gibt es einen Artikel auf „Medium" *[2]*.

22. Februar 2020

Java 8, und nun?

Migration ist wichtig, wenn auch gleichzeitig eine Herausforderung. Viele Projekte nutzen weiterhin Java 8, weil sowohl Risiko und Aufwand als auch die tatsächlichen Vorteile neuerer Versionen nicht ganz klar sind. Ein neues Slide Deck von IBMs Migrations-Expertin Dalia Abo Sheasha (genauer Titel: „Migration Tools Dev Lead") hilft, die Dinge etwas klarer zu sehen *[3]*.

25. Februar 2020

GraalVM Project Adivsory Board gegründet

Das GraalVM-Projekt hat nun ein Leitungsgremium („Project Adivsory Board"), in dem die wichtigsten das Projekt unterstützenden Organisationen vertreten sind. Logischerweise ist Oracle durch Thomas Würthinger in der Zwölferrunde vertreten, aber auch Amazon, Gluon, Pivotal oder auch das Hasso-Plattner-Institut sind repräsentiert. Details und die Entstehungsgeschichte hat „Developer Advocate" Alina Yurenko auf Medium beschrieben *[4]*. Ein erstes Meeting hat auch schon stattgefunden, in dem es unter anderem um eine verbesserte, schnellere Behandlung von Pull Requests und den Build-Prozess sowie die Gründung einer „Security Collaboration Group" in Anlehnung an das OpenJDK ging.

26. Februar 2020

MicroProfile und GraphQL

Aus dem MicroProfile-Projekt (MP) ist jetzt eine GraphQL-Spezifikation inklusive dazugehöriger API-Klassen in Version 1.0 veröffentlicht worden. Das Projekt stellt (wie bei MP üblich) keine eigene Implementierung bereit, einige Hersteller arbeiten jedoch zumindest daran. Offizieller Bestandteil der MP-Releases ist das GraphQL-API allerdings (noch) nicht, bis auf Weiteres wird sie eine eigenständige Spezifikation bleiben.

11. März 2020

Absage der JavaLand

Es hatte sich bereits angebahnt, nun ist es leider „amtlich": die JavaLand 2020 wird wegen der Corona-Krise nicht stattfinden. Damit sind auch der JavaLand-Schulungstag, der Microservices-Thementag und das Pre-Event für die neue CloudLand-Konferenz abgesagt. Es gibt jedoch schon ein paar alternative Ideen, wie zum Beispiel eine „Small JavaLand" im Juni (wenn das Virus es zulässt) und eine Serie von Online-Vorträgen. Es gab vor der Bekanntmachung Kritik am Informationsfluss – aber es war auch für die Konferenzleitung nicht einfach, sich durch das Minenfeld aus gesellschaftlicher, juristischer und wirtschaftlicher Verantwortung zu navigieren, bei gleichzeitig „ausbaufähiger" Unterstützung durch Politik und Behörden (die es selbst sicher auch nicht leicht hatten).

12. März 2020

CyberLand statt JavaLand

Einige der JavaLand-Organisatoren haben sich blitzschnell zusammengetan, um mit der CyberJug ein Web-basiertes Event am 17. März durchzuführen – dem ersten Tag der JavaLand. Es ist kein offizieller Ersatz für die JavaLand, sondern eine offene, kostenlose Online-Konferenz mit zwei bis drei Streams. Speaker zu finden war kein großes Problem – es haben ja jetzt eine ganze Reihe erstklassiger Referenten am Dienstag noch nichts vor. Die spannende Frage ist, ob es mit der technischen Umsetzung funktioniert – aber da hat die CyberJug zumindest im kleineren Rahmen schon Erfahrungen sammeln können. Die Vorbereitung der Konferenz erfolgt zu großen Teilen über den #cyberland-Channel *[5]*. Im jvm-german-Workspace kann sich jeder frei registrieren und es gibt darin auch noch andere spannende Channels; daher sei er sowieso allen ans Herz gelegt, die in der deutschsprachigen Java Community mitmachen (wollen).

17. März 2020

Tag der CyberLand

Es ist soweit: Die CyberLand startet. Nach einer kurzen Begrüßung und der Keynote von Carola Lilienthal geht es, bis zur Abschlussrunde abends um 18 Uhr, in zwei parallelen Streams weiter. Als Plattform wird BigMarker genutzt. Es haben sich zirka 1.600 Personen registriert und, wenn ich mich nicht irre, sind zeitweise auch über 800 gleichzeitig online. Durch das Speaker-Moderator-Duo und die eingebauten Plattformfunktionen wie Chat, Fragen und Umfragen funktioniert das organisatorisch sehr gut und hat ein hohes Maß an Interaktion. Die Vorträge werden mitgeschnitten und sollen im YouTube-Kanal der CyberJUG veröffentlicht werden. Leider werden dort wohl aus Datenschutzgründen die QA-Runden nach den Vorträgen fehlen; dabei sind die häufig der interessanteste Teil. Das Feedback ist sehr positiv und eine große Mehrheit kann sich eine weitere CyberLand vorstellen.

18. März 2020

Java 14

Java 14 ist da! Ich zähle die Features (JEPs) nicht noch mal alle auf, das hat Sharat Chander von Oracle bereits übernommen *[6]*. Auch Falk Sippach gibt uns ab Seite 9 in dieser Ausgabe der Java aktuell noch einmal einen umfassenden Einblick in das Update. Insgesamt sind es 16, einige mehr als in den Vor-Releases – wobei es sich teilweise um Previews oder Inkubator-Features handelt, die dann bei positivem Feedback in einem Folge-Release produktionsreif werden. Wir haben jetzt „Halbzeit" zwischen zwei Long-Term-Releases, da kann noch einiges getestet werden, was für die langfristige Nutzung reif werden soll. Das nächste LTS – Java 17 – kommt laut Fahrplan in 18 Monaten (und dieser wird seit der Einführung des Sechs-Monats-Zyklus mit schweizerischer Pünktlichkeit eingehalten).

20. März 2020

Cross-Spec Coordination/Work

Soll Eclipse MicroProfile eine übergreifende Spezifikation erhalten, die mehr regelt als nur die Versionen der Einzel-Spezifikationen, so wie es die „Umbrella"-Spezifikationen von Java EE schon immer machen? Diese Diskussion ist von Red Hats Heiko Rupp in der MicroProfile-Mailingliste neu angeregt worden, da es immer wieder auch strategische Fragestellungen gibt, die das Zusammenspiel zwischen verschiedenen Einzel-Spezifikationen betreffen. Statt einer „Plattform-Spezifikation" kommt wohl der Begriff „Architektur-Spezifikation" besser an. Die Diskussion ist nicht neu, ist jedoch bislang immer vertagt worden, weil es andere Prioritäten gab. Bis es soweit ist, könnte aber schon mal die Zusammenarbeit zwischen den einzelnen Teams verbessert werden. Ein Vorschlag von Ondro Mihalyi von Payara zielt darauf, lieber die „Main Specs" zu identifizieren, die das Zusammenspiel mit anderen Spezifikationen definieren.

Referenzen:

[1] *https://groups.google.com/forum/#!forum/microprofile*
[2] *https://medium.com/graalvm/nashorn-removal-graalvm-to-the-rescue-d4da3605b6cb*
[3] *https://www.slideshare.net/DaliaAboSheasha/migratingbeyond-java-8-228829333*
[4] *https://medium.com/graalvm/announcing-the-graalvm-project-advisory-board-282223cde700*
[5] *jvm-german.slack.com*
[6] *blogs.oracle.com*

Andreas Badelt
stellv. Leiter der DOAG Java Community
andreas.badelt@doag.org

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich im DOAG e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit 2015 ist er stellvertretender Leiter der neugegründeten Java Community innerhalb der DOAG. Beruflich hat er seit dem Jahr 1999 als Entwickler und Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet und ist seit dem Jahr 2016 als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Markus' eclipse-Corner

Eigentlich wollte ich an dieser Stelle freudig über die Fortschritte von Jakarta EE berichten. Doch dann kam Corona. Und da Viren an Grenzen nun mal nicht Halt machen, verschonte SARS-CoV-2, also das Virus, das die neue Lungenkrankheit COVID-19 auslöst, leider eben auch die JavaLand *[1]* nicht – weder die gleichnamige Konferenz, deren Durchführung aus diesem Grunde von Staats wegen verboten wurde, noch die beliebte Entwicklungsplattform. Denn dank Corona geht nach einem kurzen Aufflammen auch bei Jakarta wieder relativ wenig voran.

Doch zunächst die guten Nachrichten! Die Eclipse Foundation hat ja, wie bereits in der letzten Ausgabe der Java aktuell berichtet, einen Plan *[2]* veröffentlicht, wie die Jakarta-Entwicklergemeinde glaubt, bis zum Sommer Jakarta EE 9 veröffentlichen zu können. Wir erinnern uns: Mit weitgehend gleichem Inhalt wie in Jakarta EE 8, the artist formerly known as Java EE 8 (abzüglich einiger veralteter APIs, zuzüglich der aus Java SE verbannten Grundlagen wie JAXB und Activation API), aber mittels eines „Big Bang" umgezogen in den Package-Namespace "jakarta.*". Tatsächlich hatte sich hier im Laufe des Februars deutlicher Aktivismus gezeigt, und zwar ausnahmsweise auch in echtem Code und weniger nur in Forenbeiträgen. Explizit sei hierbei, dass nun auch jenseits Oracle und der einschlägig bekannten Volontäre deutliche Beiträge geleistet wurden. Es ging also klar voran, und die Meilensteine wurden auch mehr oder minder erfüllt. Für Außenstehende am besten zu erkennen ist dies anhand der „M1"-Meilensteine der APIs in Maven Central (oder „The Central Repository", wie es offiziell heißt). Beispielsweise ist dort seit dem 28. Februar JAX-RS in Version 3.0.0-M1 *[3]* zu finden. Diese Meilensteine dienen nicht nur dem Zweck, dass sich Hersteller dieser JARs zur Implementierung bedienen können (zum Beispiel als Grundlage für Jersey 3.0.0 *[4]*, was wiederum selbst zentraler Baustein von Eclipse GlassFish ist), sondern mithilfe dieser kann bereits heute der Umbau bestehender Anwendungen auf die neuen Package-Namen und die nun nicht mehr automatisch durch Java SE bereitgestellten APIs wie JAXB gestartet werden. Der Build auf Java SE 14 sollte damit problemlos möglich sein. Lediglich die Ausführung des Kompilats ist noch nicht möglich, da es noch nicht von jedem API eine Implementierung mit neuen Package-Namen gibt. Diese werden in den nächsten Wochen nach und nach folgen.

Wie schnell das – und nun kommen wir zu den schlechten Nachrichten – wird auch durch die Corona-Krise bestimmt. So ist anhand der GitHub-Statistikfunktionen deutlich abzulesen, dass im März die Arbeitsgeschwindigkeit an einigen Projekten wie auch bei einigen zentralen Committern deutlich nachgelassen hat. Wer selbst, wie ich, ins Home-Office umziehen musste, dem wird klar sein, was die Ursachen dafür sind. Je schneller die beteiligten Teams Lösungen finden, die wegen der Krise notwendigen Einschnitte auszugleichen, desto weniger dramatisch wird sich die Krise auf Jakarta EE auswirken. Doch selbst wenn nicht: Ich denke, es gibt in diesen Tagen wirklich Schlimmeres als einen gerissenen Release-Termin.

Referenzen

[1] *https://www.javaland.eu/*

[2] *https://eclipse-ee4j.github.io/jakartaee-platform/jakartaee9/JakartaEE9ReleasePlan*

[3] *https://search.maven.org/artifact/jakarta.ws.rs/jakarta.ws.rs-api/3.0.0-M1/bundle*

[4] *https://github.com/eclipse-ee4j/jersey/wiki/Road-Map*

Markus Karg

markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

War denn schon wieder Java-Release-Zeit?

Falk Sippach, OIO - die Java-Experten der Trivadis

An die halbjährlichen Updates von Java haben wir uns mittlerweile gewöhnt – sie tun Sprache und Plattform gut. Zuletzt waren die Änderungen jedoch überschaubar. Mit dem Mitte März 2020 erschienenen Java 14 gab es nun wieder ein regelrechtes Feuerwerk an neuen spannenden Features. In diesem Artikel wollen wir die aus Entwicklersicht besonders relevanten Themen unter die Lupe nehmen. Schließlich steht im September mit Java 15 bereits die nächste Version vor der Tür.

Die folgenden Java Enhancement Proposals (JEP) wurden umgesetzt *[1]*:

- JEP 305: Pattern Matching for instanceof (Preview)
- JEP 343: Packaging Tool (Incubator)
- JEP 345: NUMA-Aware Memory Allocation for G1
- JEP 349: JFR Event Streaming
- JEP 352: Non-Volatile Mapped Byte Buffers
- JEP 358: Helpful NullPointerExceptions
- JEP 359: Records (Preview)
- JEP 361: Switch Expressions (Standard)
- JEP 362: Deprecate the Solaris and SPARC Ports
- JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector
- JEP 364: ZGC on macOS
- JEP 365: ZGC on Windows
- JEP 366: Deprecate the ParallelScavenge + SerialOld GC Combination
- JEP 367: Remove the Pack200 Tools and API
- JEP 368: Text Blocks (Second Preview)
- JEP 370: Foreign-Memory Access API (Incubator)

JEP 359: Records

Die wahrscheinlich spannendste und gleichzeitig auch überraschendste Neuerung dürfte die Einführung der Record-Typen sein. Sie wurden noch relativ spät in das Release von Java 14 aufgenommen. Dabei handelt es sich um eine eingeschränkte Form der Klas-

sendeklaration, ähnlich zu den Enums. Entwickelt wurden Records im Rahmen des Projekts Valhalla. Es gibt gewisse Ähnlichkeiten zu Data Classes in Kotlin und Case Classes in Scala. Die kompakte Syntax könnte Bibliotheken wie Lombok in Zukunft zum Teil überflüssig machen. Kevlin Henney sieht außerdem noch folgenden Vorteil *[2]*: „I think one of the interesting side effects of the Java record feature is that, in practice, it will help expose how much Java code really is getter/setter-oriented rather than object-oriented."

Die einfache Definition einer Person mit zwei Feldern wird in *Listing 1* gezeigt. Eine erweiterte Variante mit einem zusätzlichen Konstruktor, sodass nur das Feld `name` Pflicht ist, lässt sich ebenfalls realisieren *(siehe Listing 2)*. Vom Compiler wird eine unveränderbare („immutable") Klasse erzeugt, die neben den beiden Attributen und den eigenen Methoden natürlich auch noch die Implementierungen für die Accessors (allerdings keine Getter!), den Konstruktor sowie `equals/hashcode` und `toString` enthält *(siehe Listing 3)*. Die Verwendung sieht erwartungsgemäß aus. Man sieht dem Aufrufer dabei nicht an, dass Record-Typen instanziiert werden *(siehe Listing 4)*.

Records sind übrigens keine klassischen Java Beans, da sie keine echten Getter enthalten. Man kann aber über die gleichnamigen Methoden auf die Membervariablen zugreifen. Records kann man auch mit Annotationen und JavaDocs erweitern. Im Body dürfen zudem statische Felder sowie Methoden, Konstruktoren oder Instanzmethoden deklariert werden. Die Definition von weiteren Instanzfeldern außerhalb des Record Header ist nicht erlaubt.

JEP 305: Pattern Matching for `instanceof`

Das Konzept des Pattern Matching kommt bereits seit den 1960er Jahren bei diversen Programmiersprachen zum Einsatz. Zu den modernen Vertretern zählen Haskell und Scala. Ein Pattern ist eine Kombination aus einem Prädikat, das auf eine Zielstruktur passt, und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen. Die Intention ist demnach die Destrukturierung von Objekten, also das Aufspalten in seine Bestandteile.

In Java kann man für solche Anwendungsfälle das Switch-Statement benutzen, ist dabei aber auf die Datentypen Integer, String und Enum beschränkt. Es wird noch einige Zeit dauern, bis es echtes Pattern Matching in Java gibt. Durch die Einführung der Switch Expression in Java 12 wurde jedoch bereits der erste Schritt dahin vollzogen. Mit Java 14 können wir nun zusätzlich Pattern Matching beim `instanceof`-Operator nutzen. Dabei werden unnötige Casts vermieden, zudem erhöht sich durch die verringerte Redundanz die Lesbarkeit.

Vorher musste man beispielsweise für das Prüfen auf einen leeren String beziehungsweise eine leere Collection wie in *Listing 5* gezeigt vorgehen. Jetzt kann man beim Check mit `instanceof` den Wert direkt einer Variablen zuweisen und darauf weitere Operationen ausführen *(siehe Listing 6)*. Der Unterschied mag auf den ersten Blick marginal erscheinen. Für die Puristen unter den Java-Entwicklern spart das allerdings eine kleine, aber dennoch lästige Redundanz ein.

Übrigens, die Switch Expression wurde zunächst in Java 12 und 13 jeweils als Preview-Feature eingeführt. Sie wurde nun im JEP 361

```
public record Person(String name, Person partner) {}
```

Listing 1

```
public record Person(String name, Person partner) {
  public Person(String name) {
    this(name, null);
  }
  public String getNameInUppercase() {
    return name.toUpperCase();
  }
}
```

Listing 2

```
public final class Person extends Record {
  private final String name;
  private final Person partner;

  public Person(String name) {
    this(name, null);
  }
  public Person(String name, Person partner){
    this.name = name; this.partner = partner;
  }

  public String getNameInUppercase(){
    return name.toUpperCase();
  }
  public String toString(){ /* ... */ }
  public final int hashCode(){ /* ... */ }
  public final boolean equals(Object o){ /* ... */ }
  public String name(){ return name; }
  public Person partner(){ return partner; }
}
```

Listing 3

```
var man = new Person("Adam");
var woman = new Person("Eve", man);
woman.toString(); // ==> "Person[name=Eve, partner=…]"

woman.partner().name(); // ==> "Adam"
woman.getNameInUppercase(); // ==> "EVE"

// Deep equals
// ==> true
new Person("Eve", new Person("Adam")).equals(woman);
```

Listing 4

```
boolean isNullOrEmpty(Object o){
  return == null ||
    instanceof String && ((String) o).isBlank() ||
    instanceof Collection && ((Collection) o).isEmpty();
}
```

Listing 5

```
boolean isNullOrEmpty(Object o){
  return o == null ||
    o instanceof String s && s.isBlank() ||
    o instanceof Collection c && c.isEmpty();
}
```

Listing 6

```
String developerRating(int numberOfChildren){
  return switch(numberOfChildren){
    case 0 -> "open source contributor";
    case 1, 2 -> "junior";
    case 3 -> "senior";
    default -> {
      if(numberOfChildren < 0){
        throw new IndexOutOfBoundsException(numberOfChildren);
      }
      yield "manager";
    }
  };
}
```

Listing 7

```
man.partner().name()

Result: java.lang.NullPointerException: Cannot invoke "Person.name()" because the return value of "Person.partner()" is null
```

Listing 8

finalisiert. Dadurch stehen Entwicklern zwei neue Syntaxvarianten mit einer kürzeren und klareren sowie weniger fehleranfälligen Semantik zur Verfügung. Das Ergebnis der Expression kann einer Variablen zugewiesen oder als Wert aus einer Methode zurückgegeben werden *(siehe Listing 7)*. Weitere Details können dem Artikel zu Java 13 *[3]* entnommen werden.

JEP 358: Helpful NullPointerExceptions

Der unbeabsichtigte Zugriff auf leere Referenzen ist auch bei Java-Entwicklern gefürchtet. Nach eigener Aussage von Sir Tony Hoare war seine Erfindung der Null-Referenz ein Fehler mit Folgen in Höhe von vielen Milliarden Dollar. Und das nur, weil es bei der Entwicklung der Sprache Algol in den 1960er Jahren einfach so leicht zu implementieren war.

In Java gibt es weder vom Compiler noch von der Laufzeitumgebung Unterstützung beim Umgang mit Null-Referenzen. Mit diversen Mustern und Vorgehensweisen lassen sich diese leidigen Exceptions allerdings vermeiden. Den einfachsten Weg stellt die Prüfung auf `null` dar. Leider ist dieses Vorgehen sehr mühselig und wird immer genau dann vergessen, wenn man den Check gebraucht hätte. Mit der seit dem JDK 8 enthaltenen Wrapper-Klasse „optional" kann man über das API den Aufrufer darauf hinweisen, dass ein Wert `null` sein kann und er darauf reagieren muss. Somit kann man nicht mehr aus Versehen in eine Null-Referenz hineinlaufen, sondern muss explizit mit dem möglicherweise leeren Wert umgehen. Dieses Vorgehen bietet sich unter anderem bei Rückgabewerten von öffentlichen Schnittstellen an, kostet jedoch auch eine extra Indirektionsschicht, da man den eigentlichen Wert immer auspacken muss.

In anderen Sprachen wurden längst Hilfsmittel in die Syntax und den Compiler eingebaut, wie zum Beispiel in Groovy das NullObjectPattern und der Safe Navigation Operator (*some?.method()*). Bei Kotlin kann man explizit zwischen Typen, die nicht leer sein dürfen, und solchen, bei deren Referenz auch `null` erlaubt ist, unterscheiden. Lange Rede, kurzer Sinn: Mit den NullPointerExceptions werden wir auch zukünftig in Java leben müssen. Aber immerhin erleichtern uns die als Preview-Feature eingeführten „Helpful NullPointerExceptions" nun die Fehlersuche im Ausnahmefall. Damit beim Werfen einer NullPointerException die notwendigen Informationen eingefügt werden, muss man beim Starten die folgende Option aktivieren: `-XX:+ShowCodeDetailsInExceptionMessages`. Wenn dann in einer Aufrufkette ein Wert `null` ist, bekommt man die in *Listing 8* gezeigte aussagekräftige Meldung.

Lambda-Ausdrücke benötigen eine Spezialbehandlung. Ist zum Beispiel der Parameter einer Lambda-Funktion `null`, bekommt man standardmäßig eine unzureichende Fehlermeldung *(siehe Listing 9)*. Damit der korrekte Parametername angezeigt wird, muss der Quellcode mit der Option `-g:vars` kompiliert werden. Das Resultat ist in *Listing 10* zu sehen. Bei Methodenreferenzen gibt es aktuell leider noch keinen Hinweis im Fall eines leeren Parameters *(siehe Listing 11)*.

Wenn man aber wie hier im Beispiel jeden Stream-Methodenaufruf auf eine neue Zeile setzt, lässt sich die problematische Codezeile schnell eingrenzen. Herausfordernd waren NullPointerExceptions bisher beim automatischen Boxing/Unboxing. Wird hier nun auch der Compiler-Parameter `-g:vars` aktiviert, bekommt man ebenfalls eine neue hilfreiche Fehlermeldung *(siehe Listing 12)*.

```
Stream.of(man, woman)
  .map(p -> p.partner())
  .map(p -> p.name())
  .collect(Collectors.toUnmodifiableList());

Result: java.lang.NullPointerException: Cannot invoke "Person.name()" because "<parameter1>" is null
```

Listing 9

```
java.lang.NullPointerException: Cannot invoke "Person.name()" because "p" is null
```

Listing 10

```
Stream.of(man, woman)
  .map(Person::partner)
  .map(Person::name)
  .collect(Collectors.toUnmodifiableList())
Result: java.lang.NullPointerException
```

Listing 11

JEP 368: Text Blocks

Ursprünglich als Raw String Literals bereits für Java 12 geplant, wurde in Java 13 zunächst eine abgespeckte Variante in Form von mehrzeiligen Strings namens Text Blocks eingeführt. Insbesondere für HTML-Templates und SQL-Skripte erhöht sich dadurch die Lesbarkeit enorm *(siehe Listing 13)*.

Neu hinzugekommen sind jetzt zwei Escape-Sequenzen, mit denen man die Formatierung eines Text Block anpassen kann. Um zum Beispiel einen Zeilenumbruch zu verwenden, der aber nicht explizit in der Ausgabe erscheinen soll, kann man am Zeilenende einfach ein „\" (Backslash) einfügen. Im in *Listing 14* Gezeigten bekommt man trotz Umbrüchen einen String mit einer langen Zeile.

Die zweite neue Escape-Sequenz „\s" wird zu einem Leerzeichen übersetzt. Dadurch kann man erreichen, dass Leerzeichen am Zeilenende nicht automatisch abgeschnitten (getrimmt) werden und man beispielsweise eine feste Zeichenbreite je Zeile erhält *(siehe Listing 15)*.

Was gibt es noch Neues?

Neben den gerade beschriebenen Features, die hauptsächlich für Entwickler interessant sind, gibt es natürlich diverse andere Änderungen. Im JEP 352 wurde das FileChannel-API erweitert, um die Erzeugung von MappedByteBuffer-Instanzen zu ermöglichen. Die arbeiten auf nichtflüchtigen Datenspeichern (NVM: non-volatile memory), im Gegensatz zum volatilen Speicher, dem RAM. Die Zielplattform ist allerdings auf Linux x64 beschränkt. Auch bei den Garbage Collectors hat sich wieder etwas getan. So wurde der Concurrent Mark Sweep (CMS) Garbage Collector entfernt und den ZGC gibt es jetzt auch für macOS und Windows.

Bei kritischen Java-Anwendung wird empfohlen, die Flight-Recording-Funktion in Produktion zu aktivieren. Der in *Listing 16* gezeigte Befehl startet eine Java-Anwendung mit Flight Recording und schreibt die Informationen in die recording.jfr, wobei immer die Daten eines Tages aufgehoben werden.

Normalerweise liest man die Daten dann mit dem Tool JDK Mission Control (JMC) aus, um sie zu analysieren. Neu im JDK 14 ist, dass man auch aus der Anwendung heraus asynchron die Events abfragen kann *(siehe Listing 17)*.

```
int calculate(){
  Integer a = 2, b = 4, x = null;
  return a + b * x;
}
calculate();
Result: java.lang.NullPointerException: Cannot invoke "java.lang.Integer.intValue()" because "x" is null
```

Listing 12

```
// Ohne Text Blocks
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, Escapes</p>\n" +
              "    </body>\n" +
              "</html>\n";

// Mit Text Blocks
String html = """
              <html>
                  <body>
                      <p>Hello, Text Blocks</p>
                  </body>
              </html>""";
```

Listing 13

```
String text = """
  Lorem ipsum dolor sit amet, consectetur adipiscing \
  elit, sed do eiusmod tempor incididunt ut labore \
  et dolore magna aliqua.\
""";
// statt
String literal = "Lorem ipsum dolor sit amet, consectetur adipiscing " +
  "elit, sed do eiusmod tempor incididunt ut labore " +
  "et dolore magna aliqua.";
```

Listing 14

```
String colors = """
    red  \s
    green\s
    blue \s
    """;
```

Listing 15

```
java \
-XX:+FlightRecorder \
-XX:StartFlightRecording=disk=true, \
filename=recording.jfr,dumponexit=true,maxage=1d \
-jar application.jar
```

Listing 16

```
import jdk.jfr.consumer.RecordingStream;
import java.time.Duration;

try(var rs = new RecordingStream()){
  rs.enable("jdk.CPULoad").withPeriod(Duration.ofSeconds(1));
  rs.onEvent("jdk.CPULoad", event -> {
    System.out.printf("%.1f %% %n",
      event.getFloat("machineTotal") * 100);
  });
  rs.start();
}
```

Listing 17

Im JDK 8 gab es das Tool „javapackager", das leider mitsamt JavaFX in den neueren Java-Versionen entfernt wurde. In Java 14 wird nun der Nachfolger „jpackage" eingeführt (JEP 343: Packaging Tool), mit dem wieder eigenständige Java-Installationsdateien erstellt werden können. Der Inhalt ist die Java-Anwendung mitsamt einer Laufzeit-Umgebung. Das Tool baut aus diesem Input ein lauffähiges Binärartefakt, das sämtliche Abhängigkeiten enthält (Formate: msi, exe, pkg als dmg, app als dmg, deb und rpm).

Ausblick

Vor anderthalb Jahren ist im Herbst 2019 mit Java 11 die letzte LTS-Version erschienen. Seitdem gab es bei den beiden folgenden Major-Releases jeweils nur eine überschaubare Menge an neuen Features. In den JDK-Inkubator-Projekten (Amber, Valhalla, Loom etc.) wird jedoch bereits an vielen neuen Ideen gearbeitet und so verwundert es nicht, dass der Funktionsumfang beim gerade veröffentlichten JDK 14 wieder deutlich größer ausfällt. Und auch, wenn nur wenige die neue Version in Produktion einsetzen werden, sollte man trotzdem frühzeitig einen Blick auf die Neuerungen werfen und gegebenenfalls zu den Preview-Funktionen Feedback geben. Nur so ist sichergestellt, dass sie bis zur Finalisierung im nächsten LTS-Release, das als Java 17 im Herbst 2021 erscheinen wird, gerüstet sind.

„Languages must evolve, or they risk becoming irrelevant", sagte Brian Goetz von Oracle im November 2019 in seiner Präsentation „Java Language Futures" bei der Devoxx in Belgien. Er ist als Language Architect maßgeblich daran beteiligt, dass Java trotz seiner 25 Jahre noch lange nicht zum alten Eisen gehört. Oracle hat dazu in den vergangenen Jahren einige wegweisende Entscheidungen getroffen. Die halbjährlichen OpenJDK-Releases mit den Preview-Features wurden gut angenommen. Zudem hat Oracle sein Lizenzmodell geändert. Das Oracle JDK wird jetzt zwar nicht mehr kostenfrei angeboten, das hat den Wettbewerb allerdings angekurbelt. Und so bekommt man nun von diversen Anbietern, auch noch von Oracle, freie Distributionen des OpenJDK. Das ist seit Java 11 binärkompatibel zum Oracle JDK und steht unter einer Open-Source-Lizenz.

Wir können also festhalten: Java ist noch lange nicht tot. Außerdem sind aktuell noch viele Features in Arbeit, die in zukünftigen Versionen auf ihren Einsatz warten. Uns Java-Entwicklern wird also nicht langweilig werden, die Zukunft sieht weiterhin rosig aus. Und im September 2020 erwartet uns bereits Java 15!

Referenzen:

[1] JDK 14 Projektseite: *https://openjdk.java.net/projects/jdk/14/*

[2] Zitat Kevlin Henney: *https://twitter.com/KevlinHenney/status/1226094836404670464?s=03*

[3] Rückblick Java 13: *https://jaxenter.de/java-13-neue-features-87053*

[4] Codebeispiele: *https://github.com/jonatan-kazmierczak/java-new-features/blob/master/java14.md*

Falk Sippach

JUG Darmstadt

falk@jug-da.de

Falk Sippach hat 20 Jahre Erfahrung mit Java und ist bei der OIO - den Java-Experten der Trivadis - als Trainer, Software-Entwickler und -Architekt tätig. Er publiziert regelmäßig in Blogs, Fachartikeln und auf Konferenzen. In seiner Wahlheimat Darmstadt organisiert er mit Anderen die örtliche Java User Group. Falk twittert unter @sippsack.

Effizientes Testen mit Enterprise Java

Sebastian Daschner, IBM

Testing in Enterprise-Anwendungen ist leider immer noch ein Thema, das nicht mit der Wichtigkeit behandelt wird, die es verdient. Erstellen und vor allem Maintainen von Tests beansprucht wertvolle Zeit, das Vernachlässigen von Tests ist jedoch auch keine Lösung. Wie also, mit welchen Scopes, Ansätzen und Technologien, können wir effektiver testen?

Vor allem für Anwendungen, die deutlich komplexer als „Hello World" sind, ist es sehr wichtig, welchen Ansätzen wir folgen. Ich werde mich hauptsächlich auf das Testen der funktionalen Anforderungen unserer Applikationen fokussieren, das heißt darauf, wie gut sie die Businesslogik umsetzen. Im Folgenden werde ich anhand von Best Practices aus Erfahrungen in Projekten erklären, wie man Testing für verschiedene Scopes und Ansätze effizienter gestalten kann.

Einführung

Der Sinn einer Testsuite, unabhängig von den verschiedenen Scopes, ist es, dass wir verifizieren können, ob unsere Applikationen in Produktion aus Benutzersicht genauso funktioniert wie erwartet. Da die Aufmerksamkeitsspanne des Menschen kurz ist, müssen wir dann dafür sorgen, dass die Tests schnell laufen und Turnaround-Zeiten kurz bleiben, um einen effizienten Workflow zu behalten, während wir entwickeln.

Weiterhin müssen unsere Tests auf längere Sicht wartbar bleiben. Software ändert sich und mit ausreichender Testabdeckung wird jede funktionale Änderung im Produktionscode auch eine Anpassung des Testcodes erfordern. Idealerweise ändert sich der Testcode nur dann, wenn sich die Funktionalität ändert, und nicht für Clean-up und Refactoring.

Für Microservices-Anwendungen oder jedes verteilte System wird es wichtiger, dass wir die korrekte Integration der Anwendungen sicherstellen. Das heißt, wir brauchen einen effizienten Weg, um zu verifizieren, dass die gesamte Anwendung während unserer Entwicklungsphase funktioniert.

Unittests

Unittests verifizieren das Verhalten einer einzelnen Einheit, normalerweise einer Klasse, während alle anderen Komponenten ignoriert oder simuliert werden. Meiner Erfahrung nach haben die meisten Entwickler ein sehr gutes Verständnis, wie Unittests aufgebaut sind. Im verlinkten Projekt ist ein Beispiel zu finden *[1]*. Die meisten Projekte nutzen JUnit in Kombination mit Mockito, um Dependencies zu mocken, und empfehlungsgemäß AssertJ, um lesbare Assertions zu schreiben. Worauf ich in Projekten immer Wert lege, ist, dass wir Unittests ohne weitere Extensions oder Runner ausführen, das heißt, dass wir sie mit reinem Java und JUnit laufen lassen. Der Grund dafür ist die Ausführungszeit; wir sollten in der Lage sein, Hunderte von Tests in ein paar Millisekunden laufen zu lassen.

Ein Manko von zu vielen Unittests ist, dass sie sich sehr stark an die Implementierung koppeln, vor allem an die gewählten Klassen, Strukturen und Methoden, was es schwierig macht, unseren Code zu „refactoren". In anderen Worten, für jede Refactoring-Aktion, die wir im Produktionscode vollziehen, müssen wir auch den Testcode anpassen. Im schlimmsten Fall führt dies dazu, dass die Entwickler weniger Refactorings machen, weil jene zu umständlich werden, was schnell dazu führt, dass die Qualität unseres Codes abnimmt. Idealerweise sollten die Entwickler in der Lage sein, zu „refactoren" und Dinge hin- und herzuschieben, solange die Funktionalität der Applikation sich nicht ändert.

Erfahrungsgemäß sind Unittests sehr effektiv beim Testen von Code, der eine hohe Dichte von Logik oder Funktionalität beinhaltet, wie zum Beispiel die Implementierung eines speziellen Algorithmus, und der gleichzeitig nicht sehr stark mit anderen Komponenten interagiert. Je weniger komplex der Code einer spezifischen Klasse ist, desto geringer die zyklomatische Komplexität, oder je höher die Interaktion mit anderen Komponenten ist, desto weniger effektiv ist es, diese Klasse mit Unittests zu verifizieren. Vor allem in Microservice-Szenarien mit verhältnismäßig wenig Businesslogik und einem hohen Grad an Interaktion mit externen Systemen ergibt es wenig Sinn, viele Unittests zu haben. Die einzelnen Units dieser Systeme haben typischerweise wenig komplexe Logik, von ein paar Ausnahmen abgesehen.

Use-Case-Tests

Um die Herausforderung der Kopplung von Tests zur Implementierung anzugehen, können wir einen anderen Ansatz wählen. In meinem Buch *[2]* habe ich das Konzept von Komponententests oder für einen besseren Ausdruck Use-Case-Tests beschrieben.

Use-Case-Tests sind Code-Level-Tests, die keinen Container und kein Reflection-Scanning verwenden, um die Testlaufzeit zu optimieren. Sie verifizieren einen einzelnen Use Case, inklusive aller in-

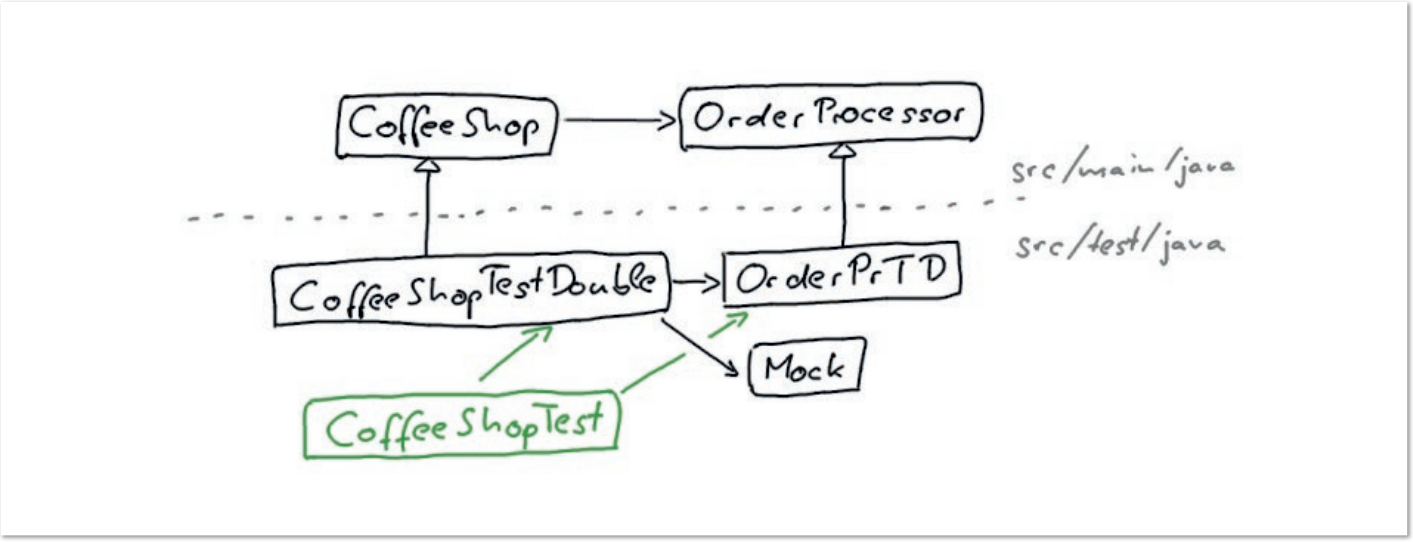


Abbildung 1: Use-Case-Test-Klassen im Projekt- und Testscope (© Sebastian Daschner)

volvierter Komponenten, exklusive Datenbankzugriffe und externer Gateways; diese externen Systeme sind „weggemockt".

Solche Szenarien ohne spezifische Testtechnologie, die unsere Komponenten schon injiziert, aufzubauen, resultiert in viel Aufwand. Wir definieren jedoch wiederverwendbare Testkomponenten oder Test-Doubles, die die Komponenten um Mocking, Wiring und Testkonfiguration erweitern, um den generellen Aufwand zu minimieren. Ziel ist es, Single Responsibilities zu erstellen, die den Impact eines Code-Change auf einzelne oder wenige Klassen reduzieren. Dies resultiert in weniger Gesamtaufwand und zahlt sich aus, sobald das Projekt größer wird, weil wir die Konfigurationskosten nur einmal zahlen.

Angenommen wir testen den Use Case zum Bestellen eines Kaffees, der zwei Klassen, `CoffeeShop` und `OrderProcessor`, beinhaltet *(siehe Abbildung 1)*. Die Test-Double-Klassen `CoffeeShopTestDouble` und `OrderProcessorTestDouble` liegen im Testscope des Projekts, während sie `CoffeeShop` und `OrderProcessor` des Main-Scopes erweitern. Die Test-Doubles können erforderliches Mocking beinhalten, die Komponenten zusammenschließen und die Public-Interfaces der Klassen mit businessspezifischen Methoden erweitern. In *Listing 1* sehen wir die Test-Double-Klasse für die `CoffeeShop`-Komponente.

Die Test-Double-Klasse kann Felder und Konstruktoren der `CoffeeShop`-Basisklasse erweitern, um die Abhängigkeiten vorzukonfigurieren. Sie nutzt abhängige Komponenten als Test-Double, zum Beispiel `OrderProcessorTestDouble`, um potenzielle Verifikationsmethoden aufrufen zu können. Die Test-Double-Klassen sind wiederverwendbare Komponenten, die einmal im Projekt-Scope geschrieben werden und in verschiedenen Use-Case-Tests Verwendung finden.

Die Use-Case-Tests verifizieren die Ausführung eines einzelnen Use Case, der auf einer Boundary ausgeführt wird, hier `CoffeeShop`. Diese Tests sind kurz und sehr lesbar, da die Konfigurations- und Mocking-Logik in den einzelnen Test-Doubles passiert. Die Use-Case-Tests können spezifische Verifikationsmethoden wie zum Beispiel `verifyProcessOrders()` aufrufen.

Je größer und komplexer unser Projekt wird, desto mehr zahlt sich dieser Ansatz aus, vor allem, da wir die einzelnen Komponenten nur einmal erstellen und so *„Single Points of Responsibility"* haben. Zum anderen sorgen wir dafür, dass die Testlaufzeit gering bleibt, da wir die Tests ausschließlich mit JUnit ausführen. Das ist der Hauptvorteil dieses Ansatzes, da wir die Use-Case-Tests genauso schnell wie normale Unittests ausführen können, es jedoch möglich machen, Produktionscode zu „refactoren", nachdem da die meisten Änderungen nur an einzelnen oder wenigen Stellen angepasst werden müssen. Zudem macht das Delegieren der Konfiguration den Code lesbarer, da unsere Testmethoden nur beinhalten, was für das einzelne Testszenario von Relevanz ist.

Code-Level-Integrationstests

Der Ausdruck Integrationstest wird in verschiedenen Kontexten unterschiedlich verwendet. Auf was ich mich beziehe, sind Tests, die die Interaktion mehrerer Komponenten verifizieren. Typischerweise nutzen Integrationstests Embedded Container oder andere simulierte Umgebungen, um ein Subset der Applikation zu testen. Test-Technologien wie Spring-Tests, Arquillian, CDI-Unit und andere machen es einfach, Tests zu schreiben und einzelne Klassen in die Testklassen zu „injecten".

In *Abbildung 2* sehen wir ein Pseudocode-Beispiel eines Integrationstests, das CDI-Unit verwendet. Das Testszenario kann einfach die Abhängigkeiten „injecten", mocken und während der Testmethoden verwenden.

Die Testtechnologien brauchen ein paar Momente, um zu starten. Erfahrungsgemäß haben Integrationstests den größten negativen Einfluss auf die Build-Zeit. Ich sehe in Projekten sehr oft, dass Entwickler existierende Testszenarien per Copy-and-paste kopieren und in einer Konfiguration laufen lassen, in dem die Applikation immer wieder aufs Neue gestartet wird. Das führt zu einer erhöhten Testlaufzeit und die Entwickler bekommen kein schnelles Feedback.

Während diese Tests die Korrektheit unserer Konfiguration verifizieren, zum Beispiel ob APIs und Annotationen korrekt verwendet wurden, sind sie nicht der effizienteste Weg, um Businesslogik zu

```
public class CoffeeShopTestDouble extends CoffeeShop {

    public CoffeeShopTestDouble(OrderProcessorTestDouble orderProcessorTestDouble) {
        entityManager = mock(EntityManager.class);
        orderProcessor = orderProcessorTestDouble;
    }

    public void verifyCreateOrder(Order order) {
        verify(entityManager).merge(order);
    }

    public void verifyProcessUnfinishedOrders() {
        verify(entityManager).createNamedQuery(Order.FIND_UNFINISHED, Order.class);
    }

    public void answerForUnfinishedOrders(List<Order> orders) {
        // setup entity manager mock behavior
    }
}
```

Listing 1

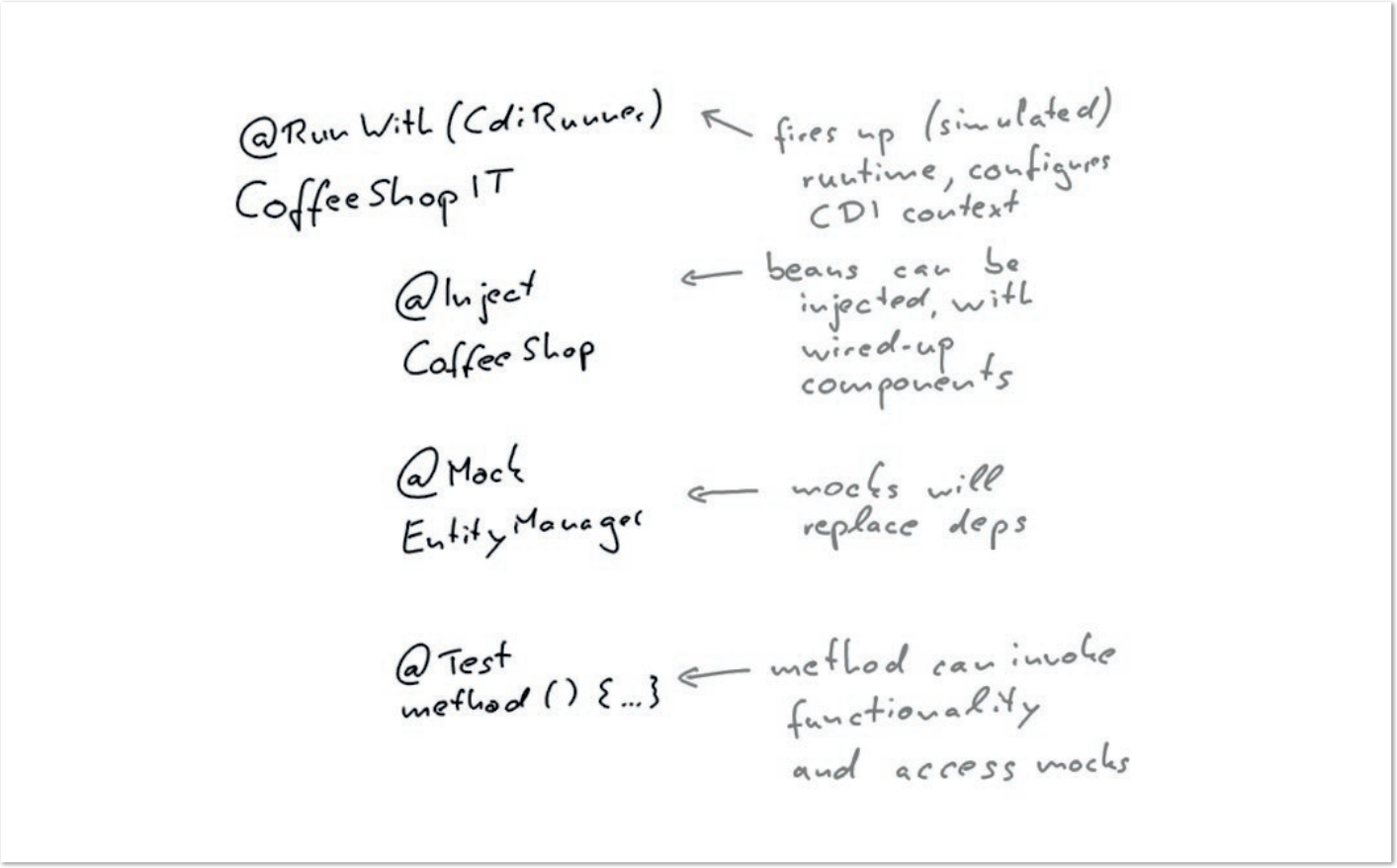


Abbildung 2: Pseudo-Code-Level-Integrationstest (© Sebastian Daschner)

testen. Vor allem in Microservices-Applikationen liefern Integrationstests nicht endgültige Sicherheit, ob die Integration der Endpunkte und Persistenz sich genauso verhält, wie später in Produktion. Im Endeffekt können immer noch kleine Unterschiede auftreten, zum Beispiel beim Mapping von JSON-Objekten.

Da Integrationstest-Technologien typischerweise einen eigenen Container starten, definieren sie einen bestimmten Lifecycle im Test und erschweren es, in ein größeres Gesamtbild integriert zu werden. Entwickler, die einen optimierten Workflow erstellen wollen, zum Beispiel indem eine Applikation in einem Hot-Reload-Modus läuft und Änderungen sehr schnell abgebildet werden, erstellen wir idealerweise integrative Tests, die sich direkt und unmittelbar gegen die schon laufende Applikation verbinden. Moderne Technologien wie Quarkus ermöglichen und vereinfachen das. Es ist sinnvoll, im Sinne der Flexibilität den Lebenszyklus der Testszenarien von dem der Testumgebung zu trennen. Das beschreiben wir im folgenden Kapitel.

Systemtests

In einer Microservices-Welt sind unsere Applikationen mehr und mehr mit anderen Ressourcen wie externen Systemen, Datenbanken oder Message-Brokern verbunden und beinhalten im Schnitt immer weniger komplexe Businesslogik. Daher ist es unabdingbar, das Verhalten unserer Systeme aus End-to-End-Perspektive zu verifizieren, das heißt, dass wir mit unserer Applikation auf die gleiche Art und Weise wie in Produktion interagieren.

Systemtests verifizieren das Verhalten einer deployten Anwendung, indem sie auf die gleichen Schnittstellen zugreifen wie später die Benutzer. Sie werden gegen eine Umgebung ausgeführt, in der die Applikation auf dieselbe Art und Weise wie in Produktion deployt und konfiguriert ist, während externe Systeme typischerweise weggemockt oder simuliert sind. Testszenarien können mit den gemockten externen Systemen interagieren und so das Szenario kontrollieren *(siehe Abbildung 3)*. Sie verifizieren, ob mit den Mocks korrekt interagiert wurde. Container-Technologien, Mock-Server und Embedded Datenbanken können hier sehr helfen.

Generell können Systemtests in allen möglichen Technologien erstellt werden, da sie unabhängig von der Implementierung sind. Generell ergibt es jedoch Sinn, auch die gleiche Technologie wie im Applikationsprojekt zu verwenden, nachdem die Entwickler damit vertraut sind, zum Beispiel Java und JUnit mit JAX-RS als HTTP-Client.

Idealerweise trennen wir das Systemtest- von dem Applikationsprojekt, um nicht aus Versehen gleiche Klassen zu verwenden und sicherzustellen, dass wir jede Änderung in den API-Contracts mitbekommen. Zudem können wir in Systemtest-Projekten vereinfachte Versionen, beispielsweise von Mapping-Klassen, verwenden.

Da Systemtests sehr schnell recht komplex werden können, müssen wir uns auch um die Wartbarkeit und Testcodequalität Gedanken machen. Generell ist es sinnvoll, spezielle Delegates zu erstellen, um mit den einzelnen Systemen zu interagieren.

Für komplexe Setups ist es auch zweckmäßig, dafür zu sorgen, dass unsere Systemtests idempotent sind, also ein bestimmtes Verhal-

ten unabhängig von dem aktuellen Zustand der Applikation verifizieren. Wir sollten es vermeiden, Testszenarien zu erstellen, die nur gegen ein frisches System oder ein System in einem bestimmten Zustand funktionieren. Use-Cases in der Realität werden ebenfalls simultan gegen länger laufende Applikationen ausgeführt.

Um Umgebungen möglichst gleich zu halten, sollten wir die Applikationen auf die gleiche Art und Weise wie in Produktion laufen lassen. Container-Technologien machen das sehr einfach. Lokal haben wir die Möglichkeit, Container auf verschiedene Arten zu starten, entweder per Shell-Skript, Docker Compose, Testcontainers oder sogar per lokalem Kubernetes- oder OpenShift-Cluster. In einer Continous-Delivery-Pipeline deployen wir normalerweise die Applikation auf eine Testumgebung, auf dieselbe Art und Weise wie in Produktion.

Abhängig von der Komplexität des Systems und des lokalen Entwicklungsworkflows können wir den Lifecycle der Applikation im Systemtest, oder außerhalb des Tests, verwalten. Erfahrungsgemäß ist es am einfachsten, die Applikation extern, zum Beispiel per Skript, zu starten und im Test zu erwarten, dass die Applikation bereits vorhanden ist. Damit ermöglichen wir, dass die Systemtests sehr schnell laufen können, da sie einen unabhängigen Lebenszyklus haben und direkt gegen eine schon deployte Anwendung ausgeführt werden. Das funktioniert sowohl für lokale Entwicklungsumgebungen als auch für spezielle Testumgebungen. In Kombination mit modernen Entwicklungstools mit Hot-Deploy-Ansätzen können wir damit einen Workflow erstellen, der sowohl Änderungen als auch Systemtests extrem schnell ausführt. Das sorgt dafür, dass der Feedback-Loop der Tests außerordentlich kurz wird und die Entwickler sehr gut im Flow bleiben. Jedoch müssen wir dafür sorgen, dass unser Systemtestcode wartbar bleibt, um die Komplexität der Systemtests managen zu können.

Entwicklungsworkflows und Pipelines

Entwickeln ist eine Flow-Aktivität und wir Entwickler sollten daran interessiert sein, unsere Workflows effizient und die Turnaround-Zeiten gering zu halten, um von den Wartezeiten nicht abgelenkt zu werden.

Generell sollten wir sicherstellen, dass die gesamte Laufzeit unserer Tests nicht länger als ein paar Sekunden beträgt – zumindest für die Tests, die wir während unserer Entwicklung ausführen. Während des Entwickelns ist es obligatorisch, einen schnellen Turnaround für diesen Zyklus, der von Codeänderungen bis zum Verifizieren in einer produktionsnahen Umgebung reicht, zu haben.

Es würde zu lange dauern, wenn wir diese Verifikation nur in einer Umgebung ausführen, die jedes Mal unsere Applikation neu baut und deployt, egal, ob lokal oder auf einer CI/CD-Umgebung. Aus diesem Grund ist es sinnvoll, sich ein Setup zu erstellen, das bei Dateiänderungen die lokal laufende Applikation modifiziert und sofort updatet. Moderne Runtimes wie Quarkus oder Open Liberty ermöglichen das.

Genauer ausgedrückt läuft dieser Workflow dann wie folgt ab: Wir ändern die Produktionscode-Klassen, führen Unittests aus, die Applikation wird sofort „hot-deployed" und idempotente Systemtests werden lokal ausgeführt. Dieser komplette Zyklus sollte nicht viel mehr als ein oder zwei Sekunden dauern, da unsere Aufmerksamkeit ansonsten woandershin wandert. Wir könnten integrative Tests, zum Beispiel nach ihrer Ausführungszeit, weiter aufteilen. So könnten wir Tests, die länger brauchen oder komplexere Szenarien erfordern, in eine separate Testsuite, die nicht immer mit ausgeführt wird, auslagern.

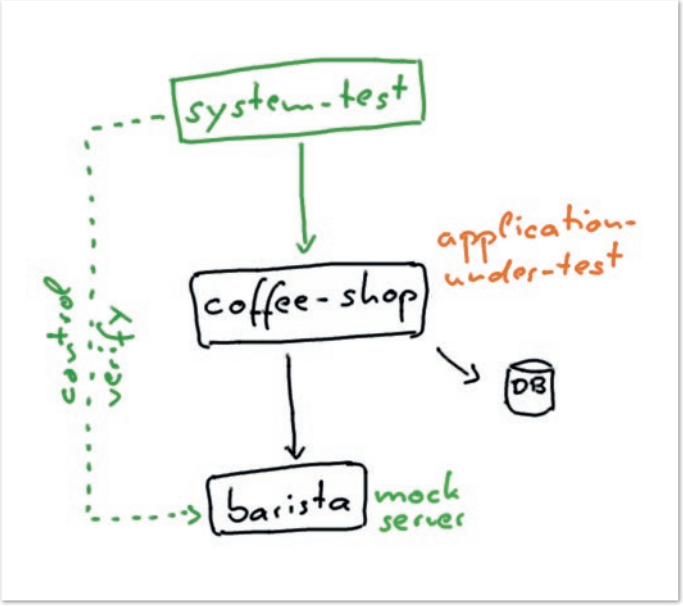


Abbildung 3: Systemtest der Coffee-Shop-Applikation (© Sebastian Daschner)

Vor allem Container-Technologien ermöglichen es uns, auch lokal komplexere Systeme, die aus mehreren Microservices, Datenbanken und Mock-Servern bestehen, aufzubauen. Diese Entwicklungsumgebung starten wir idealerweise nur einmal während unserer Entwicklungs-Session. Unsere Systemtests können dann direkt gegen die Umgebung ausgeführt werden und wir müssen nicht auf Startups warten.

Testcode-Qualität und wartbare Tests

Das größte Problem, das für nicht ausreichende Tests in Projekten sorgt, ist das Fehlen von wartbarem Testcode und Codequalität. In vielen Fällen werden Tests auf eine Art und Weise geschrieben oder per Copy-and-paste generiert, die es schwierig macht, die Tests anzupassen, sobald sich der Produktionscode ändert. Meist wird die Codequalität der Tests weniger stark beachtet als die für Produktionscode. Das funktioniert jedoch nur, solange wir nur wenige Testklassen haben. Je komplexer ein Projekt wird, desto wichtiger wird es, sich um Testcodequalität Gedanken zu machen, da Tests ansonsten immer weniger wartbar werden.

Das ist der wichtigste Punkt, um Tests wartbar zu machen: die gleichen Qualitätsprinzipien anzuwenden, die wir auch auf Produktion beachten. Vor allem *„separations of concerns"* und *„abstraction layers"*. Es ist möglich und sehr zu empfehlen, wiederverwendbare Komponenten innerhalb des Testcodes zu erstellen und gleichzeitig *„leaky abstractions"* zu vermeiden.

Schauen wir uns ein Beispiel an, das das Ganze verdeutlicht: Wir können damit anfangen, was ich manchmal *„comment-first programming"*

nenne, indem wir in Kommentaren, Pseudocode oder auf Papier schreiben, was unsere Testszenarien verifizieren sollen, auf rein konzeptueller, Business-orientierter Ebene. Zum Beispiel: „Bestelle einen Espresso in Größe groß". Oder „Verifiziere, dass die Bestellung im System ist, mit Typ Espresso und Größe groß". Das ist alles. Wie die Bestellung erstellt wurde, ist nicht Teil dieses Abstraktionslevels, sondern in einer unteren Ebene implementiert, in einer separaten Methode oder typischerweise einer Delegate. Die Delegates können dann ebenfalls Details, zum Beispiel darüber, ob der korrekte HTTP-Statuscode gesendet wurde, verifizieren. Die Delegates sollten die Implementierungsdetails wegabstrahieren.

Aus Sicht des Testers ist dieser Ansatz sehr sinnvoll, da wir mit Testszenarien anfangen, nicht mit Details, wie sie implementiert sind. Falls sich die Implementierung ändert, zum Beispiel die Kommunikation von HTTP zu etwas anderem, müssen wir nur an einem einzigen Ort anpassen. Weiterhin werden unsere Test-Cases sehr lesbar, da sie auf höherer Abstraktionsebene wiedergeben, was der Test testen soll. Falls wir an den Details interessiert sind, finden wir diese in den entsprechenden Delegates. Der in *Listing 2* abgebildete Systemtest verifiziert das Bestellen eines Kaffees.

Aus Erfahrung können auch nicht-technische Domänenexperten ohne tieferes Wissen in Java verstehen, was dieses Testszenario ausführt, wenn sie die Domäne hinter den *Types* und *Origins* verstehen und die Java-Syntax einfach ignorieren.

Aus diesem Grund behaupte ich, dass es für reale Projekte viel wichtiger ist, durch Abstraktionsebenen und Delegates ordentliche Testcodequalität einzuführen, anstatt sich auf Test-Frameworks zu fokussieren. Vor allem wenn unsere Projekte komplexer werden, zeigt sich dieser Unterschied dramatisch.

Test-Frameworks

Warum ich die meisten Test-Frameworks nicht für essenziell halte, ist, dass sie hauptsächlich syntaktischen Zucker und Bequemlichkeiten hinzufügen, aber per se nicht das Problem von wartbarem Code lösen. Anders ausgedrückt: Wenn die Testsuite nicht schon ohne bestimmte Test-Frameworks wartbar ist, wird sich die Qualität durch Hinzufügen weiterer Technologien nicht verbessern.

Ich behaupte, dass der größte positive Mehrwert für wartbaren und lesbaren Testcode dadurch zustande kommt, dass wir Testcode-APIs und Komponenten mit ordentlicher Abstraktion erstellen. Das ist vollkommen unabhängig von der Technologie und möglich in reinem Java, zum Beispiel in Tests, die per JUnit ausgeführt werden. Für die Verifikation von einzelnen Schritten hat sich AssertJ bewährt. Wir können eigene Assertion-Typen definieren, die unsere spezifische Businesslogik verifizieren, was weiterhin die Lesbarkeit unseres Codes erhöht. Mockito hat sich bewährt, wenn Klassen gemockt werden müssen, die außerhalb des Scopes unseres Tests liegen.

Weiterhin behaupte ich, dass diese Test-Technologien bereits ausreichend sind. Besonders seit JUnit 5 haben wir noch weitere interessante Features für dynamische oder parametrisierte Test. Trotzdem gibt es ein paar Test-Frameworks, die einen Blick wert sind, wie zum Beispiel Spock, Testcontainers oder Cucumber.

Generell gilt: Je komplexer ein Projekt wird, desto kleiner ist der Impact von Test-Frameworks auf die Produktivität, Lesbarkeit und Wartbarkeit und desto wichtiger ist, dass wir uns um die Testcodequalität, ordentliche Abstraktions-Layer und Delegates Gedanken machen. Wenn Entwickler weitere Technologien verwenden wollen, ist das in Ordnung, aber wir müssen uns der Trade-offs bewusst sein, zum Beispiel darüber, wie viel Zeit es erfordert, alternative JVM-Sprachen, Abhängigkeiten oder Versionen zu konfigurieren.

Fazit

Folgende Punkte möchte ich als Zusammenfassung noch mitgeben. Wir sollten:

```
class CreateOrderTest {

    private CoffeeOrderSystem coffeeOrderSystem;

    @BeforeEach
    void setUp() {
        coffeeOrderSystem = new CoffeeOrderSystem();
    }

    @Test
    void createVerifyOrder() {
        List<URI> originalOrders = coffeeOrderSystem.getOrders();

        Order order = new Order("Espresso", "Colombia");
        URI orderUri = coffeeOrderSystem.createOrder(order);

        Order loadedOrder = coffeeOrderSystem.getOrder(orderUri);
        assertThat(loadedOrder).isEqualToComparingOnlyGivenFields(order,
                "type", "origin");

        assertThat(coffeeOrderSystem.getOrders()).hasSize(originalOrders.size() + 1);
    }

    ...
}
```

Listing 2

- einfache Test-Technologien mit sauberem Code präferieren,
- die Benutzung von Testfällen mit erweiterten Test-Runnern, wie Spring-Tests, Arquillian oder CDI-Unit, limitieren,
- die Lebenszyklen von Tests und Testumgebungen trennen,
- für Microservice-Architekturen End-to-End-Systemtests den Code-Level-Tests vorziehen,
- lokale Entwicklungsumgebungen aufbauen, die die Systemtests unmittelbar ausführen können, gegen bereits deployte Anwendungen,
- wiederverwendbare Testkomponenten erstellen, die die Zuständigkeiten trennen, sowohl für System- als auch Code-Level-Testszenarien,
- Abstraktion und Delegation im Testcode verwenden,
- uns daran erinnern, dass Testcodequalität wichtiger als die verwendeten Frameworks ist,
- uns der Trade-offs der unterschiedlichen Testscopes bewusst sein, um eine effektive und balancierte Testsuite für unsere Projekte zu erstellen.

Referenzen

[1] Coffee-Shop-Beispielprojekt: *https://github.com/sdaschner/coffee-testing*

[2] Buch: Architecting Modern Java EE Applications *https://blog.sebastian-daschner.com/entries/book-modern-java-ee*

Sebastian Daschner

IBM

mail@sebastian-daschner.com

Sebastian Daschner ist Java Developer Advocate bei IBM, Consultant, Autor und Trainer. Er ist Autor des Buches „Architecting Modern Java EE Applications". Sebastian nimmt an Open-Source-Prozessen wie dem JCP oder der Eclipse Foundation teil, hilft mit, die Zukunft von Enterprise Java zu formen, und entwickelt an diversen Open-Source-Projekten mit. Für seinen Beitrag in der Java Community und dem Java-Ökosystem wurde er mit den Titeln Java Champion, Oracle Developer Champion und JavaOne Rockstar ausgezeichnet. Neben Java benutzt Sebastian intensiv Linux und Container-Technologien wie Docker und Kubernetes. Er evangelisiert Java- und Programmierthemen in seinem Blog unter https://blog.sebastian-daschner.com und auf Twitter unter @DaschnerS. Wenn er nicht gerade mit Java arbeitet, bereist er gerne die Welt oder genießt guten Kaffee.

Testen vs. Verifikation – Warum automatisiertes Testen eine Illusion ist

Thomas Papendieck, OPITZ Consulting Deutschland GmbH

Aus unserem täglichen Leben sind Computersteuerungen nicht mehr wegzudenken. Wir sind umgeben von unzähligen Geräten, die von Software gesteuert sind. Das Funktionieren dieser Geräte ist essenziell für unsere moderne Lebensweise. Die Folgen eines Versagens reichen von kleineren Unannehmlichkeiten, beispielsweise wenn die Kaffeemaschine falsch dosiert, bis hin zu Todesfällen, von denen das Versagen der Assistenzsysteme bei Tesla [1] oder der Flugzeugsteuerung in der Boing 737-Max [2] nur die Spitze des Eisberges mit medialer Beachtung darstellen. Je nach Gefährdungsklasse sind daher umfangreiche Tests notwendig, um das Verhalten eines Geräts und seiner Softwaresteuerung in allen, besonders aber in kritischen Situationen des Anwendungsbereichs, sicherzustellen. Obwohl es wahrscheinlich niemanden gibt, der dem widersprechen würde, ist es in der Software-Industrie so, dass gerade das Testen der Software häufig als Kostentreiber mit dem geringsten Einfluss auf das Endprodukt betrachtet wird. Die Automatisierung von Tests bietet dabei die Möglichkeit, ein erhebliches wirtschaftlichen Potenzial freizusetzen, ohne die Sicherheit des Produkts aufs Spiel zu setzen. In diesem Artikel möchte ich die Grenzen dieses Ansatzes aufzeigen.

Probleme manuellen Testens

Das Personal

Das Testen von Software – in erster Linie manuelles Testen – ist eine komplexe Aufgabe, die umfangreiche Kenntnisse und bestimmte Persönlichkeitseigenschaften erfordert *[3]*. Diese Tatsache wird leider gerade in Unternehmen, die weniger im Fokus der Öffentlichkeit stehen, ignoriert. Oft wird manuelles Testen der Software genutzt, um junge Mitarbeiter in das Projekt „einzuführen", obwohl sie die notwendigen Kenntnisse nicht haben, weder über das Testen selbst noch über die Fachlichkeit, die zu testen ist. Die Erwartung ist vielmehr, dass das Testen des Produkts die Einarbeitung in die Fachlichkeit darstellt. Die notwendigen Kenntnisse über das Testen werden dabei regelmäßig unterschätzt. Ob der neue Mitarbeiter sich von seiner Persönlichkeitsstruktur her überhaupt zum Tester eignet, wird in der Regel überhaupt nicht bedacht. Dies führt dazu, dass Tests nicht mit der nötigen Sorgfalt ausgeführt werden.

Die Ergebnisse der Testläufe können aufgrund der fehlenden Fachkenntnisse falsch interpretiert werden. Darüber hinaus findet dann die Dokumentation der Testläufe unvollständig statt. Die von nicht qualifiziertem Personal durchgeführten manuellen Tests sind also von fragwürdigem Wert. Es bleibt zu hoffen, dass sich diese Erkenntnis in den softwareproduzierenden Unternehmen weiter ausbreitet und die Verantwortlichen bereit sind, das Personal zu investieren, das ihr Produkt testen soll, um letztlich die Kosten der Tests zu senken, beispielsweise durch Automatisierung der Testläufe.

Die Wiederholbarkeit

Eine wichtige Testart bei der Prüfung von Software sind Regressionstests *[4]*. Deren wichtigste Eigenschaft ist die Wiederholbarkeit. Das bedeutet konkret, dass mehrere aufeinander folgende Testläufe zum selben Ergebnis führen müssen, sofern sich das Programm (im getesteten Bereich) nicht geändert hat. Leider sind wir Menschen nicht gut darin, komplexe Dinge in exakter Weise zu wiederholen. Zudem werden solche „Wiederholungstests" oft anhand schriftlicher Aufzeichnungen gemacht. Das sind meistens Listen mit Anweisungen dazu, welche Eingaben zu machen und welche Ausgaben als „korrekt" zu bewerten sind. Diese Aufzeichnungen enthalten häufig unpräzise Angaben, sodass sich für den Tester Interpretationsspielraum auftut, den er entsprechend seinem oft mangelhaften Wissen über die fachlichen Anforderungen ausschöpft.

Aus diesem Grund ist bei einem fehlgeschlagenen manuellen Regressionstest immer zu prüfen, ob sich tatsächlich Fehler in das Programm eingeschlichen haben oder ob der Fehler „nur" in der Testausführung liegt. Auf der anderen Seite kann ein manueller Testlauf als erfolgreich eingestuft werden, obwohl tatsächlich ein Fehler im Produkt vorliegt, weil die Ausführung des Testlaufs fehlerhaft war und so „um den Fehler herum" getestet wurde. Diese Unsicherheit schlägt sich nicht nur direkt auf den zu betreibenden Aufwand und damit auf die Kosten des Tests nieder, sondern auch auf die Produktsicherheit.

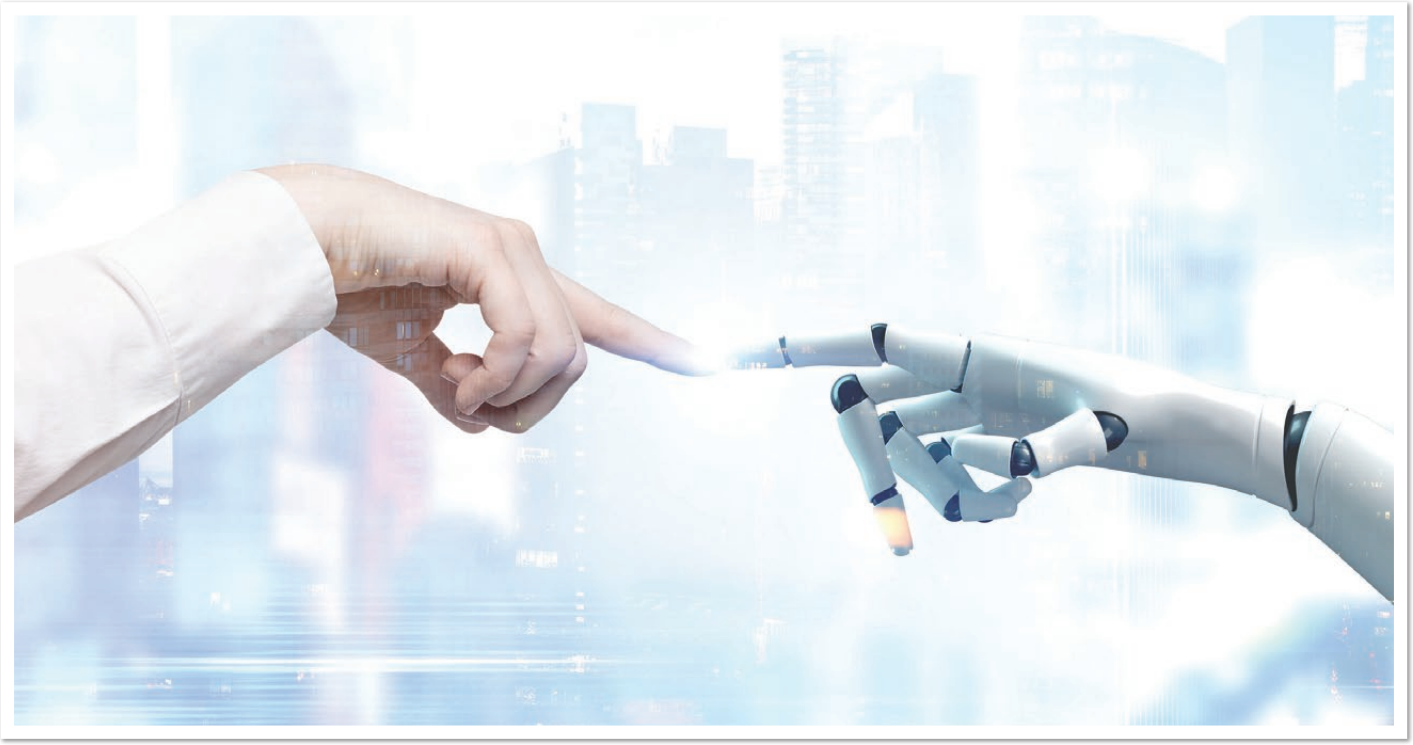
Der Zeitfaktor

Genau wie im vorigen Abschnitt ist auch dieser Nachteil manueller Tests direkt in der Limitierung menschlicher Fähigkeiten begründet.

Dabei sind zwei Einflussfaktoren zu unterscheiden, die den Zeitfaktor als kritischen Aspekt manuellen Testens in den Fokus rücken. Da ist zunächst die Tatsache, dass der Mensch sowohl kognitive als auch mechanische Tätigkeiten nur mit einer begrenzten individuellen Geschwindigkeit ausführen kann *[5]*. Aufgrund dessen kann ein Mensch nur eine begrenzte Anzahl von Testfällen in einer vorgegebenen Zeiteinheit ausführen. Diese Geschwindigkeit lässt sich im Prinzip dadurch steigern, dass der Tester die manuellen Tests häufiger durchführt. Diese durch „Training" zustande gekommene Steigerung hat aber Grenzen. Zudem birgt dieses „Training" auch Risiken, die sich auf die Testqualität auswirken. Ein Großteil des Geschwindigkeitsgewinns wird durch eine Verringerung der Aufmerksamkeit erreicht. Und ein weniger aufmerksamer Tester kann leichter einen Fehler bei der Testausführung machen oder ein problematisches Testergebnis falsch interpretieren.

Auf der anderen Seite ist das manuelle Testen von Software in besonderer Weise mental anstrengend, eben weil das Aufmerksamkeitslevel hier höher sein muss als bei anderen Tätigkeiten im Software-Entwicklungsprozess. Stellt man diese mentale Anstrengung der eines Programmierers gegenüber, zeigt sich, dass diese sich in ihrer Struktur unterscheiden.

So stehen dem Programmierer verschiedene Werkzeuge zur Verfügung, die dafür sorgen, das Flüchtigkeitsfehler schnell erkannt werden und so keinen Einfluss auf das Ergebnis nehmen können. Das beginnt bei der IDE und dem Compiler, die beispielsweise Tippfehler im Code aufdecken, bis zum Test-Driven-Development, das als Arbeitsmethode gegen versehentliche Änderungen bereits bestehender Funktionalität schützen kann. Das führt dazu, dass der Programmierer seine kognitiven Ressourcen in eine Richtung lenken kann, die sich vorwiegend mit der Erweiterung des Produktes und weniger mit der eigenen Fehlbarkeit beschäftigen. Auf diese Weise ist die Arbeit des Programmierers, obwohl ebenfalls kognitiv sehr anspruchsvoll, weniger erschöpfend. Ein Tester kann



bei der Ausführung manueller Tests keine adäquaten Hilfsmittel einsetzen, weil es sie schlicht nicht gibt.

In der Konsequenz muss die Zeit, in der ein Tester manuelle Tests der Anwendung ausführt, von mehr oder weniger regelmäßigen Pausen unterbrochen werden, damit seine Konzentrationsfähigkeit erhalten bleibt. Jedoch führt jede Unterbrechung einer vorwiegend kognitiven Tätigkeit dazu, dass die Person nach dieser Unterbrechung sich erneut in die Situation „hineinfinden" muss, was zu einer Verzögerung über die reine Pausenzeit hinausführt [6].

Eine weitere Restriktion, die den Zeitverlauf manueller Tests beeinflusst, ist der arbeitsrechtliche Rahmen, in den der Tester eingebunden ist. Die Arbeitszeit des Testers ist die begrenzende Zeiteinheit, in welcher der Tester die manuellen Tests durchführt. Nun kann man durchaus versuchen, diese Restriktion durch den Einsatz von Personal in anderen Zeitzonen auszugleichen, dies setzt jedoch eine gewisse Mindestgröße des Unternehmens voraus und ist daher kein universeller Ansatz. Außerdem hat der Einsatz mehrerer Personen wieder eigene Probleme, die ein schlichtes Addieren der Arbeitszeiten zu einer Milchmädchenrechnung degradieren.

Computer schlägt Mensch

Alle oben aufgeführten Probleme werden durch Testautomatisierung gelöst. Im Gegensatz zu uns Menschen sind Computer sehr gut darin, einmal festgelegte Abläufe beliebig oft exakt und mit hoher Geschwindigkeit zu wiederholen. Automatisierte Tests können unabhängig von Arbeitszeitgrenzen rund um die Uhr ausgeführt werden und lassen sich problemlos auf mehreren Computern gleichzeitig ausführen. Ein zusätzlicher Server ist meist kostengünstiger, als einen weiteren menschlichen Tester einzustellen. Wenn man Zeit sparen muss, ist bei automatisierten Tests der KIWI-Ansatz („Kill it with Iron") also durchaus eine Option, besonders, wenn man dabei auf virtuelle Rechner in der Cloud zurückgreifen kann.

Wo liegt das Problem?

Das Problem ist zunächst einmal ein sprachliches. Im üblichen Sprachgebrauch wird nämlich der Begriff des Testens falsch eingesetzt. Wenn man sich außerhalb des Kreises der „Wissenden" über das Testen unterhält, meint man nur einen Teilbereich dessen, was laut ISTQB (International Software Testing Qualifications Board) als „Testen" definiert ist. Deren Definition für das Testen lautet:

> ***„Der Prozess, der aus allen Aktivitäten des Lebenszyklus besteht (sowohl statisch als auch dynamisch), die sich mit der Planung, Vorbereitung und Bewertung eines Softwareprodukts und dazugehöriger Arbeitsergebnisse befassen. Ziel des Prozesses ist sicherzustellen, dass diese allen festgelegten Anforderungen genügen, dass sie ihren Zweck erfüllen, und etwaige Fehlerzustände zu finden [7]."***

Im allgemeinen Sprachgebrauch reduzieren wir das Testen dann aber auf den Teil, der als „dynamische Tests" bezeichnet wird, wobei wir nicht zwischen der Erstellung der Testfälle und deren Aus-

führung unterscheiden, sondern lediglich Letzteres im Blick haben. Dabei kennt das ISTQB einen gesonderten Teilprozess für die Testausführung, den sie aus der ISO 9000 übernommen haben, nämlich die Verifizierung:

> *„Bestätigung durch Bereitstellung eines objektiven Nachweises, dass festgelegte Anforderungen erfüllt worden sind [8].“*

Warum ist es nun wichtig, auf diese, auf den ersten Blick erbsenzählerisch erscheinende, Unterscheidung zu bestehen? Der Grund ist, dass die im Sprachgebrauch vernachlässigte Erstellung der Testfälle im Gegensatz zu deren Ausführung, also der Verifizierung, Kreativität erfordert. Ohne diese Unterscheidung müssen wir zwangsläufig eine falsche Vorstellung darüber entwickeln, was Testautomatisierung tatsächlich leisten kann.

Grenzen der Testautomatisierung

Die Testautomatisierung stellt Werkzeuge bereit, mit denen von Menschen in einer maschinenlesbaren Form definierte Testfälle von diesen Testwerkzeugen ausgeführt und die Reaktionen des getesteten Systems bewertet werden. Das Produkt wird also gegen die in den Testfällen definierten Kriterien verifiziert. Diese Definition eines Testfalls beinhaltet immer die Kriterien für die Bewertung der Reaktion des getesteten Systems. Diese Kriterien müssen derzeit zwingend von einem Menschen gemacht werden.

Einige Werkzeuge scheinen zwar in der Lage zu sein, diese Entscheidung selbst treffen zu können, beispielsweise solche, die das zu testende System mit zufälligen Eingaben füttern, aber bei genauer Betrachtung zeigt sich, dass auch hier ein Mensch bei der Erstellung des Tests den Bereich der Eingabewerte einschränken muss, da das Werkzeug nicht selbstständig entscheiden kann, welche Reaktion des Systems auf die zufälligen Eingabewerte gewünschtes Verhalten ist und welches nicht. Auf absehbare Zeit wird es die künstliche Intelligenz nicht schaffen, dem Menschen diese Entscheidung abzunehmen.

Fazit

Testautomatisierung ist ein großartiges Mittel, um nicht nur die Kosten und die Dauer der Testphase zu senken, sondern auch die Qualität der Tests zu erhöhen. Sie schafft dies, indem sie den Menschen mit seinen Einschränkungen als Störfaktor und Fehlerursache aus der Testausführung eliminiert. Sie übernimmt den Teil, der nach der Definition des ISTQB als Verifizierung bezeichnet wird.

Auf der anderen Seite werden weiterhin hochqualifizierte Personen mit dem Testen von Software beschäftigt sein, indem sie die Werkzeuge der Testautomatisierung nutzen, um neue Testfälle zu definieren. Die Werkzeuge der Testautomatisierung können die Tester zwar bei der Erstellung von Testfällen unterstützen, indem sie über Vorlagen und Schablonen formale Aspekte der Testfall-Generierung sicherstellen, aber in absehbarer Zeit werden sie den kreativen Akt nicht übernehmen können, in dem ein Mensch Testfälle aus den Anforderungsdokumenten extrahiert. Bis auf Weiteres wird automatisiertes Testen in dem Sinne, dass selbst die Testfälle selbstständig von den Werkzeugen erstellt werden und der Tester den Prozess nur noch überwacht, eine Illusion bleiben.

Quellen

[1] Gregor Hebermehl: Zwei Personen im gerammtem Honda Civic sterben *https://www.auto-motor-und-sport.de/elektroauto/tesla-autopilot-unfall-honda-civic/*

[2] Frank Bäumer: Die vermeidbare Katastrophe *https://www.tagesschau.de/wirtschaft/boeing-235.html*

[3] ALPHAJUMP: Was macht ein Softwaretester? *https://www.alphajump.de/karriereguide/beruf/softwaretester*

[4] Wikipedia: Regressionstest *https://de.wikipedia.org/wiki/Regressionstest*

[5] CogniFit: Verarbeitungsgeschwindigkeit. *https://www.cognifit.com/de/wissenschaft/kognitive-fahigkeiten/Verarbeitungsgeschwindigkeit*

[6] Christian Steinbrich: Hol dir die Kontrolle über deine Konzentration zurück! – Wirksame Ablenkungs-Killer für besseres Lernen *https://www.121watt.de/produktivitaet/besser-lernen-mehr-konzentration/*

[7] International Software Testing Qualifications Board: Glossar „Testen” *http://glossar.german-testing-board.info/v3.21/#t,*

[8] International Software Testing Qualifications Board: Glossar „Verifizierung”

Thomas Papendieck

OPITZ Consulting Deutschland GmbH

Thomas.Papendieck@opitz-consulting.com

Thomas Papendieck ist seit 20 Jahren in der Softwareentwicklung tätig, davon seit 15 Jahren bei OPITZ CONSULTING Deutschland GmbH. Er ist gelernter Elektrotechniker und studierte zunächst Radar-Technik und später Informatik.

Ist eine Testabdeckung von 100 Prozent erstrebenswert?

Dr. Roger Butenuth, codecentric AG

Warum testen wir unsere Programme? Bereits 1972 sagte Dijkstra, dass ein Test immer nur die Existenz von Fehlern zeigen kann, nicht aber deren Abwesenheit [1]. Die „saubere" Lösung wären Korrektheitsbeweise. Aber wer kann das schon leisten? Kann oder sollte man stattdessen alles testen? Zu 100 Prozent?

Jeder Computernutzer – nicht nur Programmierer – weiß es: Programme enthalten Fehler. Manche mehr, manche weniger. Fehlerfreie Programme wird vermutlich niemand je erlebt haben – von Trivialitäten wie „Hello World!" einmal abgesehen.

Haben wir uns damit abgefunden? Nicht bei allen Systemen: Geht es um Systeme wie ABS, ESP oder die Steuerung von Zügen oder Flugzeugen, sind die meisten doch etwas anspruchsvoller, da es

um Menschenleben geht. Als Java-Entwickler bewegt man sich jedoch meistens in anderen Domänen, in denen es nur um Geld geht, profane Dinge wie Web-Shops, Versicherungssoftware oder Online-Banking.

Warum akzeptieren wir bei dieser Business-Software so viele Fehler? Weil auch eine Kostenseite existiert: Entwicklungskosten. Dort zu sparen funktioniert allerdings nur begrenzt. Irgendwann wird einem der kurzsichtige Sparwahn auf die Füße fallen, entweder weil einem die Kunden wegen der schlechten Qualität weglaufen oder weil man sich durch einen kapitalen Fehler aus dem Geschäft katapultiert. Langfristig wird die Weiterentwicklung einer vermurksten Software außerdem teuer, da jede kleine Änderung das Risiko einer Fehlerlawine in sich birgt.

Aus dem Titel kann man raten, dass ich für 100 Prozent Testabdeckung plädieren werde, trotz der weisen Worte von Dijkstra. Warum? Für komplexe Software ist ein Korrektheitsbeweis mit der dazu notwendigen formalen Spezifikation – zumindest mit heutiger Technik – illusorisch. Ich sehe dafür zwei Gründe:

1. Der Fachbereich besteht nicht aus Ingenieuren. Wenn er spezifiziert, dann meistens mit Use-Cases, nicht mit formalen Bedingungen und Regeln.
2. Die Variabilität in Form unterschiedlicher diskreter Zustände ist bei Business-Software wesentlich größer als bei den meisten sicherheitskritischen Embedded Systems. Dort wird eher in einem Kontinuum gesteuert oder geregelt, was sich mathematisch leichter fassen lässt.

Wenn wir aus diesen Gründen die Korrektheit nicht beweisen können, müssen wir die Korrektheit entweder glauben oder durch Tests zeigen, dass zumindest gewisse Beispiele und die zugehörigen Wege durch den Programmcode zum Testzeitpunkt funktioniert haben.

Mit dem letzten Satz habe ich schon meine Strategie skizziert: Wenn der Fachbereich seine Spezifikation in Form von Use-Cases vornimmt, handelt es sich dabei um Beispiele. Im einfachsten Fall sind es simple Beispiele, die nur linear durchlaufen werden. Komplexe Beispiele können auch Verzweigungen und/oder Schleifen enthalten. Als Programmierer kann man entweder klassisch vorgehen und auf Basis der darin enthaltenen Informationen Code schreiben oder man nutzt Test Driven Development (TDD): Dabei erzeugt man zuerst aus den Beispielen Testfälle, anschließend wird so lange programmiert, bis alle Testfälle grün sind.

Geht man den TDD-Weg und programmiert wirklich minimalistisch, so hat der fachliche Code automatisch eine Testabdeckung von 100 Prozent. Problem gelöst? Nicht ganz, die Sache hat zwei Haken: Erstens werden die Use-Cases des Fachbereichs niemals vollständig sein. Wer in der Lage ist, so vollständig in Prosa ein gewünschtes Systemverhalten zu beschreiben, der könnte es meistens auch direkt programmieren. Als Programmierer werden wir immer mit Lücken kämpfen. Diese können sowohl dadurch entstehen, dass jemand nicht zu Ende gedacht hat, als auch dadurch, dass man Dinge nicht niederschreibt, die für denkende Menschen selbstverständlich sind. Computer sind jedoch dumm, ihnen muss man Sachverhalte in vielen Zeilen Code explizit erklären.

Zweitens gilt die Aussage nur für den fachlichen Code. In einer idealen Welt, in der fachlicher und technischer Code vollständig getrennt sind und wir auch ansonsten das perfekte Framework für unsere Anwendung nutzen, existiert der zweite Haken nicht. In der Praxis fällt jedoch oft noch technischer Code an, der durch diesen Mechanismus nicht vollständig getestet wird. Hier obliegt es dem Entwicklerteam, seine eigene Spezifikation zu schreiben – zum Beispiel in Form von Testfällen – und sicherzustellen, dass der Code korrekt ist.

Wem das Geschriebene bekannt vorkommt, der hat vermutlich richtig geraten, es ist nichts anderes als agile Entwicklung, bei der Schritt für Schritt (oder Sprint für Sprint) immer nur genau das programmiert wird, was der Product Owner vorgibt. Damit in Folgesprints nichts, was vorher erstellt wurde, kaputt gemacht wird, muss man die Funktionen aus den vorigen Sprints immer mittesten. Hier wird schnell klar, warum solche Tests automatisiert werden sollten, sonst hat man bei n Sprints und nur einem Testfall schon n*(n+1)/2 Tests durchzuführen.

Agile Entwicklung und automatisierte Tests sind heute gängige Praxis. Was ich in der Praxis bisher selten gesehen habe, ist eine automatische Messung der Testabdeckung. Und wenn sie gemessen wird, gibt es entweder keine Vorgaben für einen Mindestwert oder die Vorgabe ist kleiner als 100 Prozent.

Wie ist Testabdeckung überhaupt definiert? Ich benutze hier die Zweigüberdeckung, auch C1 genannt *[2]*. Für eine vollständige Zweigüberdeckung ist es zum Beispiel bei einem „if-then-else“ erforderlich, sowohl den „then“- als auch den „else“-Zweig zu durchlaufen, ich benötige dafür also schon zwei Testfälle. Stehen zehn „if-then-else“ hintereinander (nicht geschachtelt), kann ich (im günstigen Fall) immer noch mit zwei Testfällen eine komplette Abdeckung herstellen: Einer für alle „then“, einer für alle „else“.

Die zwei Testfälle für ein „if-then-else“ gelten nur dann, wenn eine einfache Bedingung abgefragt wird. Bei zusammengesetzten Bedingungen mit and/or hat man es genau genommen mit einem geschachtelten „if-then-else“ zu tun, entsprechend benötigt man schon drei oder vier Testfälle. Bei Java reichen wegen der Kurzschlussauswertung (short-circuit evaluation) drei, da bei einer And-Verknüpfung der zweite Teil nicht mehr ausgewertet werden muss, wenn der erste zu „false“ evaluiert.

Eine While-Schleife erfordert für die C1-Abdeckung immer mindestens zwei Fälle: Einen, bei dem die Schleife nicht durchlaufen wird, und einen zweiten mit einem oder mehr Durchläufen der Schleife.

Wenn man ein prozentuales Maß der Testabdeckung angeben möchte, kann man entweder Blöcke oder Zeilen zählen. Ein Block kann durchaus mehrere Zeilen lang sein, für ihn muss nur gelten, dass er ganz oder überhaupt nicht durchlaufen wird. Das heißt, er darf keine Verzweigungen oder Schleifen enthalten. Meine Erfahrung zeigt: Die C1-Abdeckung erfordert Testcode in der gleichen Größenordnung wie der zu testende Code.

Erheblich aufwendiger wird es, wenn man die C2-Abdeckung anstrebt, bei der alle Kombinationen von Pfaden abgedeckt werden. Hier steigt der Aufwand exponenziell mit der Anzahl von Schleifen

und Verzweigungen. Es ist klar, dass dafür eine 100-Prozent-Abdeckung nur für Systeme möglich ist, die im Wesentlichen einfache Dinge wie Regelalgorithmen ausführen und kaum Schleifen oder Verzweigungen enthalten.

Für den Rest des Artikels werde ich mich nur noch auf die C1-Abdeckung beziehen, mehr halte ich in der Praxis von Business-Anwendungen für nicht erreichbar. Die C1-Abdeckung ist – zumindest für Java – mit Open-Source-Tools in der IDE oder im Build-Prozess einfach zu ermitteln (EclEmma, Cobertura etc.) Man kann sich also nicht damit herausreden, dass Tools nicht existieren oder sie dem Chef zu teuer sind.

Die Frage ist, warum soll ich mir die Mühe mit den 100 Prozent machen? Erstens: Man spart sich Diskussionen – sollen es jetzt 60, 80 oder 95 Prozent sein? Zweitens: Wenn es weniger als 100 Prozent sind, mit welchem Recht existiert Code, der in keinem Test ausgeführt wird? Wer steht als Erster auf und traut sich zu sagen: „Ich kann Code schreiben, der fehlerfrei ist!" – ich bleibe sitzen, ich kann das nicht. Ich habe schon an trivialen Stellen Fehler gemacht, in *Listing 1* ist ein Beispiel (glücklicherweise nur aus einem Spielprojekt) dafür zu sehen.

Wer findet den Fehler? Kleiner Tipp: Es ist nicht die Tatsache, dass hier gegen ein im Klartext gespeichertes Passwort verglichen wird, das ist an der Stelle egal, da es nur um ein One-Time-Passwort geht. Es muss spät gewesen sein, als ich den Code geschrieben habe. Aufgefallen ist mir der Fehler erst, als ich die Testabdeckung auf 100 Prozent bringen wollte. Irgendwie gab die Passwortprüfmethode auch bei falschem Passwort grünes Licht. Kein Wunder, schließlich habe ich `user` und `password` jeweils mit sich selbst verglichen, eine der Seiten des `equals()` hätte jedoch ein `this`. erfordert. Irgendwie peinlich, passiert aber schnell.

Daher gilt für mich die Annahme: Was nicht getestet ist, funktioniert auch nicht. Man kann manuell oder automatisiert testen, aber es muss getestet werden. Jetzt höre ich schon das Wehgeschrei: Aber Getter, Setter, Konstruktoren – muss man die testen? Hier wünsche ich mir besseres Tooling: Wenn das Tool erkennt, dass die entsprechenden Methoden den Standardaufbau haben (nur ein Return, nur eine Zuweisung an ein Feld mit gleichem Namen), brauche ich dafür keinen Test. Dann habe ich etwas Besseres: Das Tool hat *bewiesen*, dass die Methode korrekt ist! Es gibt bei Java-Coverage-Tools Ideen in diese Richtung, es scheint aber bei keinem der Open-Source-Tools bis jetzt so richtig zu funktionieren – schade! Andererseits: Wenn es das Tool nicht hergibt, macht ein Test, der die Getter/Setter/Konstruktoren aufruft und die Ergebnisse mit einem *„assert"* prüft, auch kaum Arbeit.

Wenn ich also von der gleichen Menge Test- wie Produktivcode ausgehe und dabei die gleiche Produktivität der Entwickler in Zeilen pro Tag annehme (ich weiß, ein ziemlich unsinniges Maß), kann die Entwicklung um maximal den Faktor zwei aufwendiger werden. Wohlgemerkt: verglichen mit null automatisierten Tests! Der Faktor

```
context.setAuthenticator(new BasicAuthenticator("FPL Server") {
  @Override public boolean checkCredentials(String user, String password) {
      return user.equals(user) && password.equals(password);
  }
});
```

Listing 1

wird in der Praxis kleiner sein. Da ein Projekt nicht nur aus Entwicklung besteht, wird auch nicht das gesamte Projekt um den Faktor aufwendiger. Gegenrechnen lässt sich der gesparte Aufwand durch Fehlerbehebung und der höhere Wert der Software durch die gesteigerte Qualität.

Tests bieten eine gute Gelegenheit, sich seinen „code under test" näher anzuschauen: Habe ich mich wirklich an das DRY-Prinzip (Don't Repeat Yourself) gehalten? Wenn nicht; jetzt rächt es sich. Für jedes doppelt programmierte „if" muss ich jetzt auch doppelt Testfälle schreiben. Das ist eine prima Gelegenheit, aufzuräumen und doppelten Code zu entfernen, indem man ihn in eigene Methoden auslagert.

Schwierig zu testen sind Exceptions und Catch-Blöcke beim Zugriff auf externe Systeme. Man muss es irgendwie schaffen, die Fehler zu simulieren oder auszulösen. Gerade hier halte ich Tests aber für wichtig: Nichts ist schlimmer als ein System, das bei externen Fehlern mit internen Fehlern alles noch weiter eskaliert. Es gibt hier kein Patentrezept, manchmal reicht ein kaputter JDBC-Connection-String zum Auslösen einer Exception, manchmal benötigt man eine eigene Implementierung eines Interface oder man nutzt eines der vielen Mocking-Frameworks.

Aufwand spart auch hier das DRY-Prinzip: Solche Exceptions sollten nicht an jeder Ecke abgefangen werden, am besten geschieht dies an einer zentralen Stelle. Steht die Datenbank, bleiben einem in den meisten Systemen sowieso kaum Ausweichszenarien, als eine nette Fehlerseite anzuzeigen und hoffentlich irgendwie noch eine Nachricht in Richtung Betriebsteam abzufeuern. Leider sind die zugehörigen Exceptions in Java oft „checked Exceptions", was zu unschönem Code führt. Da hat man es gut, wenn ein Framework das schon versteckt.

Nachträglich lassen sich die 100 Prozent schwer erreichen. Es ist viel einfacher, wenn ein Refactoring des Codes erlaubt ist, oder noch besser: wenn direkt mit der 100-Prozent-Direktive gearbeitet wird. Die Folge ist dann eine TDD-Entwicklung oder zumindest ein zeitnahes Refactoring, das für bessere Testbarkeit sorgt.

Heißt 100 Prozent Testabdeckung, dass alles funktioniert? Leider nicht. Es heißt, dass der Code ausgeführt wurde. Wollen wir wissen, ob er auch das richtige Ergebnis geliefert hat, dürfen wir die Asserts nicht vergessen. Ich habe in meiner beruflichen Laufbahn schon eine Codebasis übernommen, die eine hervorragende Testabdeckung hatte. Doch leider ohne Asserts. Bei näherer Betrachtung stellte sich heraus, dass der Code komplett falsche Ergebnisse lieferte. Der zuständige Entwickler hatte Glück, er war nicht mehr greifbar. Ich hatte Pech, ich durfte den Code neu schreiben.

Aber auch mit 100 Prozent und Asserts: Wir wissen nur, dass die Beispiele funktionieren. Wenn der Product Owner richtige Use-Cases geliefert hat, ist damit schon viel gewonnen; es ist jedoch keine Garantie für Fehlerfreiheit.

In vielen Projekten werden nur Unittests automatisiert, Tests über die Grenzen des eigenen Systems (oder sogar Systemteils) hinweg werden dagegen nur manuell ausgeführt. Die Gefahren dadurch: Wegen des großen manuellen Aufwands fallen solche Tests schnell unter den Tisch. Auch ist die Messung der Testabdeckung bei ihnen schwieriger, schließlich muss sie bei jedem Lauf einzeln protokolliert werden und die Protokolle müssen zusammengeführt werden. Dabei sind Absprachen über Systemgrenzen – und damit meist auch über Teamgrenzen – hinweg ein viel größeres Risiko als innerhalb des Teams.

Ein (altes) mahnendes Beispiel ist der Erststart der Ariane 5, der aufgrund eines nicht entdeckten Fehlers in einem mehrere Hundert Millionen Euro teuren Feuerwerk endete. Was war das Problem? Das Navigationssystem stammte noch aus der Ariane 4. Die Rakete hatte sich jedoch geändert: Die Ariane 5 war schneller, sodass es zu einem Overflow kam. Hätte man die Tests der Ariane 4 mit den Flugbahndaten der Ariane 5 wiederholt, wäre der Fehler aufgefallen *[3]*.

Schlusswort

Lasst uns besseren Code schreiben! Wir werden in der Welt immer abhängiger von Software, Fehler werden irgendwann auf uns zurückfallen und unsere Branche immer weiter diskreditieren. Ich mag es nicht, dass wir in der Öffentlichkeit als Schrottproduzent wahrgenommen werden. Wer die Chance hat, sollte in seinem nächsten Projekt mal das 100-Prozent-Ziel ausprobieren. Ich habe es in einem Spielprojekt aus einer Laune heraus getan *[4]*. Es hat mir nicht nur gezeigt, an welch vermeintlich trivialen Stellen noch Fehler stecken. Es hat auch zu einem anderen Blick auf den Produktivcode geführt. Der testbare Code sieht einfach besser aus.

Quellen

[1] Edsger W. Dijkstra (1972) The humble programmer. ACM Digital Library, *https://dl.acm.org/doi/10.1145/355604.361591*

[2] Wikipedia (abgerufen am 14.2.2020) Kontrollflussorientierte Testverfahren, *https://de.wikipedia.org/wiki/Kontrollflussorientierte_Testverfahren*

[3] Prof. J. L. LIONS et. al. (1996) ARIANE 5 / Flight 501 Failure / Report by the Inquiry Board, *https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf*

[4] R. Butenuth (2018-20) FPL – Functional Programming Language, *https://github.com/rbutenuth/fpl*

Dr. Roger Butenuth

codecentric AG

roger.butenuth@codecentric.de

Dr. Roger Butenuth hat sich zu Uni-Zeiten mit Parallelrechnern beschäftigt. Er hat langjährige Erfahrung in der Projekt- und Produktentwicklung und beschäftigt sich in der letzten Zeit mit API-Management und Integration in großen IT-Umgebungen.

Moderne Integrationstests mit Testcontainers

Daniel Krämer und Maik Wolf, anderScore GmbH

Ist Testcontainers das lang ersehnte JUnit-Framework, um einfach und kostengünstig Integrationstests zu fahren? Wie verhält sich das Framework in der Praxis? Mit praktischen Beispielen gehen wir der Frage nach, ob mit Testcontainers die nächste Evolutionsstufe im Bereich Integrationstest erreicht ist.

Der Kollege ist irritiert und genervt, denn sein lokaler Integrationstest gegen die Testdatenbank ist überraschend fehlgeschlagen. Dabei hat er doch nur seinen Code umstrukturiert. Minuten vergehen, bis er schließlich realisiert: Der Kollege gegenüber hat zur gleichen Zeit seinen Test gefahren und damit seine Datenbasis überschrieben. Und schon geht er wieder los, der ständige Streit um die Frage: „Wer fährt wann seine Tests?"

Kennen Sie solche oder ähnliche Situationen auch? Wir Entwickler sind motiviert, unseren Code zu testen, wir möchten die Qualität auf einem hohen Niveau halten, doch unsere Infrastruktur oder Kollegen behindern uns bei dieser Mission. Aber muss das wirklich sein? In Zeiten von Containern auf Systemen wie Kubernetes, OpenShift, AWS, Azure etc., die uns eine schnelle, leichtgewichtige und kostengünstige Bereitstellung neuer Instanzen unserer Anwendungen versprechen, muss es doch eine Lösung geben.

Könnte vielleicht Testcontainers *[1]* ebendiese sein? Das Versprechen: „Testcontainers is a Java library that supports JUnit tests, providing lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container." Für das oben skizzierte Problem klingt das nach der gesuchten Lösung. Bevor wir jedoch in die Tiefen dieses Frameworks abtauchen, wollen wir zunächst herausfinden, in welchem Umfeld wir uns bewegen.

Integrationstests

Integrationstests finden sich in der Mitte der Testpyramide *(siehe Abbildung 1)* wieder und geht man nach Martin Fowler *[2]*, so kosten uns diese weniger als GUI-Tests, jedoch mehr als klassische Unittests. Dieser Umstand basiert darauf, dass unsere Testszenarien komplexer werden. Bezogen auf den Integrationstest bedeutet dies, dass wir Teile unserer Anwendung gegen „Fremdsysteme" testen wollen. In unserem Fall ist dies eine Datenbank, in anderen Kontexten könnte das aber auch Apache Kafka, ein Nginx- oder Elasticsearch-System sein.

Warum nun diese Art der Integrationstests? Warum nimmt man nicht den einfachen Weg und testet gegen eine In-Memory-Datenbank wie HSQLDB oder H2 *[3]*? Die einfache Antwort: Weil jedes System seine individuellen Eigenheiten hat. Nur, wenn wir unser Testszenario so nah wie möglich an der produktiven Umgebung halten, können wir sicherstellen, dass unsere Anwendung sich erwartungsgemäß verhält.

Herausforderungen beim Integrationstest

Sprechen wir über Integrationstests, so müssen wir auch über die Hindernisse sprechen, die diese mit sich bringen. Im oben skizzierten Fall gibt es für das Entwicklerteam nur eine Testdatenbank. Diese wird nicht nur von den Entwicklern verwendet, sondern auch von dem automatisierten Integrationstest auf unserem CI-System. Fahren Entwickler und/oder das CI-System parallel ihre Tests, so ist es unausweichlich, dass die Test-Cases sich stören.

Zugegeben, das skizzierte Szenario ließe sich durch lokale Datenbanken für die Entwickler umgehen. Jedoch hängt dies stark von der Entwicklungsumgebung ab. Auch die Konfiguration der jeweiligen Anwendung kann hier einen Zeitoverhead mit sich bringen, der nicht immer mit dem zur Verfügung stehenden Budget einhergeht. So durften wir genau dieses Szenario in einem unserer Projekte erleben.

> *„Wenn ich acht Stunden Zeit hätte, um einen Baum zu fällen, würde ich sechs Stunden die Axt schleifen."*
>
> Abraham Lincoln

Auch andere Hindernisse machen uns den Weg zu schnellen und damit kostengünstigen Integrationstests schwer. Wer garantiert uns, dass die Testdatenbank zu jeder Zeit erreichbar ist? Die Reaktionszeit in den SLAs (Service Level Agreement) für Testsysteme genießt nicht immer eine hohe Priorität (beziehungsweise Budget).

Besonders wenn ein Testsystem mehreren Entwicklern zur Verfügung steht; wer kann sicherstellen, dass die Konfiguration oder das Datenbankschema nicht von einem Kollegen für seinen Testlauf manipuliert wurde? Bestimmt hat dieser auch daran gedacht, neben seinem High-Prio-Bug und den anstehenden Meetings diese wieder auf den Normalzustand zurückzusetzen.

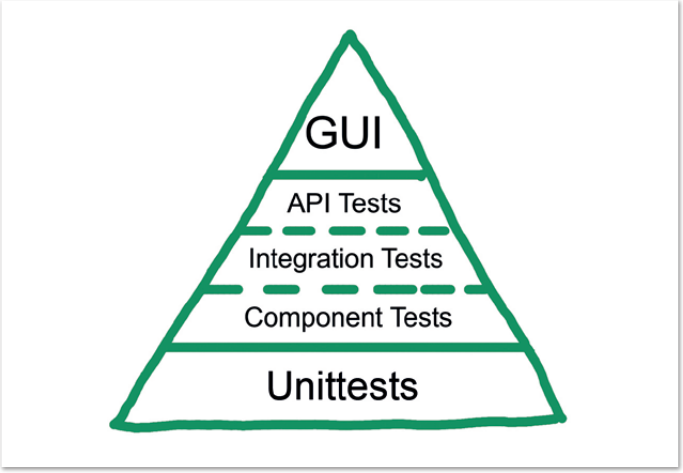


Abbildung 1: Testpyramide (© anderScore GmbH)

Durch solche und viele andere Szenarien verlieren wir die Kontrolle über unser Testsystem. Wir verbringen unnötig viel Zeit damit, nach Fehlern in unserem frisch produzierten Code zu suchen, die gar nicht da sind. Das alles nur, weil wir den Anspruch haben, unseren Code zu testen, um eine hohe Qualität zu gewährleisten.

> *„Es ist besser, ein Licht zu entzünden, als auf die Dunkelheit zu schimpfen."*
>
> Konfuzius

Was benötigen wir?

Was benötigen wir, damit wir effizienter und somit kostengünstiger unsere Integrationstests fahren können? Wir möchten…

- die Einheitlichkeit der Systeme gewährleisten, sodass uns auf der Produktionsebene keine Überraschungen erwarten.
- eine Testumgebung haben, in der unsere eventuellen Testfehler reproduzierbar sind und nicht von den Launen der Systemumgebung abhängen.
- die volle Kontrolle über unsere Testumgebung haben und nicht durch fremde Konfigurationen bestimmt werden.

Dies alles soll gewährleisten, dass unsere Testumgebung der Produktionsumgebung so nahe wie möglich kommt.

Testcontainers

Unsere grundsätzlichen Anforderungen an Integrationstests wären damit geklärt. Ein erneuter Blick auf das eingangs skizzierte Projekt macht jedoch deutlich, dass es einer zentralen Datenbank an Reproduzierbarkeit, Kontrolle und Lokalität mangelt, während eine In-Memory-Lösung mit Einheitlichkeit und Produktionsnähe in Konflikt steht – Zeit also für eine Alternative.

Ausgangslage

Schauen wir uns unseren "Patienten" zunächst noch einmal genauer an. Im konkreten Anwendungsfall haben wir es mit einer neu entwi-

```
@SpringJUnitConfig(TestConfig.class)
@ActiveProfiles("test")
@ExtendWith(DbContainerExtension.class)
@TestExecutionListeners({DependencyInjectionTestExecutionListener.class, FlywayTestExecutionListener.class})
public class SchedulerServiceTest {
    @Inject
    private SchedulerService schedulerService;

    @Test
    @FlywayTest(locationsForMigrate = {"db/scheduler_data"})
    public void findAll() {
        List<Scheduler> result = schedulerService.findAll();

        assertEquals(2, result.size());
    }
}
```

Listing 1: Einfacher Datenbanktest

ckelten Anwendung auf Basis von Spring Boot *[4]* zu tun, deren Datenbasis umgebungsspezifisch in PostgreSQL vorgehalten wird. Lokal und auf dem Build-Server ausgeführte Integrationstests der Persistenzschicht sowie Anwendungstests mit Selenium *[5]* teilen sich eine Datenbankinstanz. Teil des Projektes sind ferner die Kollegen von eben, die sich in schöner Regelmäßigkeit gegenseitig ihre Testdaten zerschießen. In der Test-Stage fehlgeschlagene Builds gehören gewissermaßen zum guten Ton, die Nachrichten des Build-Servers werden von den meisten E-Mail-Clients mittlerweile als Spam aussortiert.

Einführung

Stoßen wir als Entwickler auf ein Problem, so haben wir üblicherweise eine durchaus realistische Chance, dass ebendieses schon anderen Menschen derart Kopfschmerzen bereitet hat, dass sie bereits eine (mehr oder weniger) passende Lösung erarbeitet haben. In unserem Falle hört diese auf den Namen „Testcontainers", basiert auf docker-java *[6]* und steht als Open-Source-Bibliothek all jenen zur Verfügung, die ihre Java-Integrations- oder Anwendungstests mit Docker *[7]* und JUnit *[8]* zur Laufzeit ein Stück weit voneinander entkoppeln möchten.

Beispielcode

Wie sich ein solcher Test für unsere Anwendung (nach ein wenig eigener Vorarbeit) auf Basis von JUnit 5, Spring Test und Flyway *[9]* gestalten lässt, schauen wir uns im *Listing 1* an.

Auf den ersten Blick wird wenig Spektakuläres geboten. Der Test prüft die korrekte Funktion eines Service zur Verwaltung von Schedulern, der seine Inhalte offensichtlich aus einer mittels Flyway vorkonfigurierten Datenbank bezieht. Tatsächlich wird diese aber vor jedem Testlauf durch die (selbst implementierte) DbContainer-Extension in einem frischen Container hochgefahren und vorkonfiguriert. Bevor wir nun allerdings einen Blick unter die Haube werfen und die Magie lüften, wollen wir uns zunächst einen Eindruck darüber verschaffen, welche Funktionalität das Framework denn eigentlich so zu bieten hat.

Verwendung

Grundsätzlich erlaubt uns Testcontainers die Einbindung beliebiger öffentlicher oder interner Docker-Images, sodass sich für jeden Kontext ein geeignetes Setup finden sollte. Ausgangspunkt ist der GenericContainer, den wir wahlweise aus einem fertigen Image heraus starten oder on-the-fly mithilfe einer eigenen, am klassischen Dockerfile orientierten DSL definieren können. Wer bereits mit Docker vertraut ist, dürfte sich im Fluent-API *(siehe Listing 2)* schnell heimisch fühlen.

Im konkreten Beispiel starten wir in wenigen Zeilen ein kompaktes Alpine Linux, das auf dem freigegebenen Port 80 eine Antwort auf die Frage nach dem Leben, dem Universum und dem ganzen Rest *[10]* liefert. Sollte es bei der Verarbeitung wider Erwarten zu einem Fehler kommen, sitzen wir mit einem zusätzlichen LogConsumer direkt an der Quelle. Ausgaben auf der Kommandozeile können von diesem wahlweise eingesammelt oder gleich zur Laufzeit in einen Stream verpackt werden *(siehe Listing 3)*.

Bekanntlich kommt ein Container selten allein. Als erfahrene Nutzer von Docker wissen wir allerdings nur zu gut, dass ein Container zur Laufzeit von Werk aus ein striktes Kommunikationsverbot zu seinen

```
GenericContainer container =
    new GenericContainer("alpine:3.2")
            .withExposedPorts(80)
            .withEnv("MAGIC_NUMBER", "42")
            .withCommand("/bin/sh", "-c", "while true; do echo \"$MAGIC_NUMBER\" | nc -l -p 80; done");
```

Listing 2: Fluent-API

```
Slf4jLogConsumer logConsumer = new Slf4jLogConsumer(LOGGER);
container.followOutput(logConsumer);
```

Listing 3: LogConsumer

```
Network network = Network.newNetwork();

GenericContainer foo = new GenericContainer()
    .withNetwork(network)
    .withNetworkAliases("foo");

GenericContainer bar = new GenericContainer()
    .withNetwork(network);
```

Listing 4: Container Network

Nachbarn besitzt. Um dieses für unsere Tests aufzuheben, müssen wir daher zunächst ein wenig netzwerken *(siehe Listing 4)*.

Möchte unser Container `bar` zwecks Kollaboration nun mit seinem Artgenossen `foo` sprechen, steht ihm dieser lose gekoppelt unter der internen Domain „foo" zur Verfügung.

URL nun händisch zusammenzuschrauben, fragen wir den Container einfach höflich nach seiner aktuellen Adresse *(siehe Listing 6)*.

Dieses Vorgehen ist bereits deshalb anzuraten, weil der vom Host auf den Container gemappte Port des DBMS bei jedem Start randomisiert und nach Verfügbarkeit neu vergeben wird. Was zunächst wie eine vermeidbare Fehlerquelle anmutet, entpuppt sich im Umfeld der CI/CD-Pipeline als durchaus sinnvoll. Bei einer Vielzahl parallel betriebener Container müssten wir ansonsten nämlich selbst dafür sorgen, dass bei den angefragten Ports keine Konflikte entstehen.

Integration mit Spring – Persistenz

Nach Lektüre der vorausgegangenen Kapitel integrieren wir die neue Technologie nun voller Vorfreude in unser Spring-Boot-Entwicklungsprojekt. Schnell erinnern wir uns dabei an die Schmerzen

```
JdbcDatabaseContainer<?> container = new PostgreSQLContainer<>()
    .withDatabaseName("TestDB")
    .withUsername("TestUser")
    .withPassword("SecretPassword");
```

Listing 5: Konfiguration des Datenbankcontainers

Module

Prinzipiell bietet uns der GenericContainer bereits sämtliches Rüstzeug, das wir für containergestützte Integrationstests benötigen. In der Praxis entstehen aber schnell die ersten Fragen: Welche JDBC-URL benötige ich, um auf meine neu gedockerte Datenbank zugreifen zu können? Wie setze ich eigentlich Benutzername und Passwort? Welches Image sollte ich wählen? Um uns das Entwicklerleben an dieser Stelle noch ein wenig mehr zu erleichtern, stellt Testcontainers bereits vorkonfigurierte Module mit einem erweiterten API zur Verfügung. Zur gebotenen Prominenz gehören etwa:

- Datenbankmanagementsystem (DBMS) (unter anderem PostgreSQL, MySQL, Oracle, Cassandra, Neo4j)
- Elasticsearch
- Apache Kafka
- Nginx
- Selenium (inklusive VNC)

Wollen wir nun einen Container mit PostgreSQL konfigurieren, kommen wir mit einem „Einzeiler" aus *(siehe Listing 5)*. Anstatt die JDBC-

```
String jdbcUrl = container.getJdbcUrl();
```

Listing 6: Abfragen der Container JDBC-URL

parallel ausgeführter Integrationstests auf der Datenbank und entscheiden uns daher, dieses gemeinsame Fremdsystem in einen Container zu verfrachten. Um eine maximale Isolation zwischen den Testfällen zu erreichen, möchten wir den Container samt DBMS zudem nach jedem Testlauf frisch aufsetzen. JUnit 5 bietet uns für solche Aufgaben das Konzept der Callback an, das wir dankend annehmen und in den anfangs gezeigten Test einbinden. Um Flyway und den Container sinnvoll in den Spring Context integrieren und in unseren Tests abrufen zu können, verpacken wir die zugehörigen Objekte noch schnell in eigene Singleton Beans *(siehe Listing 7)*.

Anpassen müssen wir dort auch die obligatorische DataSoure, die sich nun stets nach der Konfiguration des hochgefahrenen Containers zu richten hat *(siehe Listing 8)*.

```
public class DbContainerExtension implements AfterEachCallback {

    @Override
    public void afterEach(ExtensionContext extensionContext) throws Exception {
        DatabaseContainerHolder containerHolder = SpringExtension.getApplicationContext(extensionContext)
                .getBean(DatabaseContainerHolder.class);

        Flyway flyWay = SpringExtension.getApplicationContext(extensionContext).getBean(Flyway.class);

        containerHolder.refresh();
        flyWay.migrate();
    }
}
```

Listing 7: DbContainerExtension

```
@Bean
public DataSource dataSource(DatabaseContainerHolder containerHolder) {
        JdbcDatabaseContainer<?> dbContainer = containerHolder.get();

        HikariConfig hikariConfig = new HikariConfig();
        hikariConfig.setJdbcUrl(dbContainer.getJdbcUrl());
        hikariConfig.setUsername(dbContainer.getUsername());
        hikariConfig.setPassword(dbContainer.getPassword());
        hikariConfig.setDriverClassName(env.getProperty("jdbc.driverClassName"));

        return new HikariDataSource(hikariConfig);
    }
```

Listing 8: HikariConfig

```
private JdbcDatabaseContainer<?> newContainer(){
    JdbcDatabaseContainer<?> container = new PostgreSQLContainer<>()
            .withDatabaseName(DB_SCHEMA)
            .withUsername(DB_USER)
            .withPassword(DB_PASSWD);

    if (hostDbPort > 0){
        container.setPortBindings(asList(hostDbPort + ":" + CONTAINER_DB_PORT));
    }

    container.start();

    return container;
}
```

Listing 9: JdbcDatabaseContainer

```
BrowserWebDriverContainer<?> container = new BrowserWebDriverContainer<>()
           .withCapabilities(new ChromeOptions())
           .withRecordingMode(RECORD_FAILING, new File("./target/"));
```

Listing 10: BrowserWebDriverContainer

Spätestens jetzt sind wir froh, den Container nicht selbst zusammengeschraubt, sondern das entsprechende Modul herangezogen zu haben. Wir starten unsere Datenbanktests und blicken gespannt auf den Monitor. Umgehend stellt sich Ernüchterung ein, denn bis auf eine Ausnahme stehen sämtliche Tests auf Rot: „Connection refused". Nach einer Weile erinnern wir uns schließlich an die randomisiert vergebenen Ports und müssen feststellen, dass Testcontainers und Spring leider doch nicht ohne Weiteres Hand in Hand gehen. Während die DataSource unseres Spring Context zu dessen Lebenszeit von einer festen JDBC-URL ausgeht, wechselt unser Container nach jedem Neustart seinen Port und ist somit nicht mehr aufzufinden. Den zwischen Testläufen gecachten Spring Context nach jeder Methode neu aufzubauen, wäre zwar eine naheliegende Option, aber schädlich für die Performance. Als kampferprobte Entwickler geben wir uns nicht so leicht geschlagen und behelfen uns schließlich mit einem Konfigurationstrick in unserem eigenen DatabaseContainerHolder *(siehe Listing 9)*.

Wird im Spring Context ein neuer Container hochgefahren, merken wir uns initial den zufällig zugeordneten Port auf dem Host. Kommt es im Anschluss zu Neustarts des Containers, setzen wir diesen wieder explizit fest. Auf diese Weise profitieren wir weiterhin von randomisierten Ports, synchronisieren sie aber mit dem Lebenszyklus des Spring Context. Alternativ bestünde auch die Möglichkeit, die Konfiguration der DataSource zur Laufzeit dynamisch anzupassen. Inwiefern es sich bei einem solchen Vorgehen um guten Stil handelt, ist allerdings umstritten.

Integration mit Spring – Anwendung

Nachdem die Einführung von Testcontainers zu einer Befriedung der parallel testenden Kollegen beigetragen hat, kommt uns schließlich der Gedanke, das Spiel noch ein wenig weiterzutreiben und auch unsere Anwendungstests auf Basis von Selenium stärker zu isolieren. In Ergänzung zur Datenbank benötigen wir hierfür auch einen Web Driver, den uns Testcontainers praktischerweise über ein vordefiniertes Modul zur Verfügung stellt. Unsere Aufgabe besteht nun wieder darin, diesen mithilfe eines JUnit-Callback in Szene zu setzen *(siehe Listing 10)*.

Auch hier genügen wenige Zeilen Code, um einen frischen Headless Chrome aufzusetzen und auf unsere Anwendung anzusetzen. Lassen wir nun einen unserer Anwendungstests laufen, begrüßt uns das Framework allerdings wieder mit einem „Connection refused". Wir hatten es bereits vermisst. Einige Augenblicke später werden uns gleich zwei konzeptionelle Probleme klar. Zum einen nutzt die Spring-Boot-Applikation in Tests ebenfalls randomisierte Ports und zum anderen dürfen Container gar nicht auf den Host und damit unsere Anwendung zugreifen. Selbstverständlich gibt es aber auch für diese Schmerzen ein geeignetes Heilmittel. In weiser Voraussicht haben die Entwickler von Testcontainern nämlich bereits eine

Möglichkeit vorgesehen, einzelne Ports des Hosts für Container zu öffnen. Ebenso ist Spring Boot durchaus auskunftswillig, was seine aktuelle Konfiguration betrifft, sodass wir beide APIs nur noch miteinander verbinden müssen *(siehe Listing 11)*.

Idealerweise möchten wir in unserem Test aber nicht direkt mit dem Container sprechen, sondern gleich mit dem enthaltenen WebDriver interagieren. Für solche Zwecke stellt uns JUnit das Konzept des ParameterResolver zur Verfügung *(siehe Listing 12)*.

```
String serverPort = SpringExtension.getApplicationContext(extensionContext).getEnvironment()
                .getProperty("local.server.port");

Testcontainers.exposeHostPorts(parseInt(serverPort));
```

Listing 11: HostPortExposing

```
@Override
public Object resolveParameter(ParameterContext parameterContext, ExtensionContext extensionContext)
            throws ParameterResolutionException {

        BrowserWebDriverContainer<?> container = extensionContext.getStore(GLOBAL)
                .get(BrowserWebDriverContainer.class.getSimpleName(), BrowserWebDriverContainer.class);

        return container.getWebDriver();
    }
```

Listing 12: WebDriver auslesen

```
@Override
public Object resolveParameter(ParameterContext parameterContext, ExtensionContext extensionContext)
            throws ParameterResolutionException {

        String serverPort = SpringExtension.getApplicationContext(extensionContext).getEnvironment().
getProperty("local.server.port");
        ServletContainerContext context = new ServletContainerContext("host.testcontainers.internal",
parseInt(serverPort));

        return context;
    }
```

Listing 13: Server-Port-Ermittlung

```
@SpringBootTest(webEnvironment = RANDOM_PORT)
@ActiveProfiles("test")
@ExtendWith(DbContainerExtension.class)
@ExtendWith(WebDriverContainerExtension.class)
@ExtendWith(ServletContainerContextParameterResolver.class)
@ExtendWith(WebDriverParameterResolver.class)
public class SchedulerTest {

  @Test
  public void createScheduler(ServletContainerContext context, RemoteWebDriver webDriver) {
      webDriver.get(context.getHttpUrl());

      // SchedulerOverviewPage
      assertEquals("Scheduler", webDriver.findElement(By.tagName("h1")).getText());
      webDriver.findElement(By.id("new")).click();

      [...]
  }
}
```

Listing 14: Selenium-Test

```
public class PostgreSQLContainer<SELF extends PostgreSQLContainer<SELF>> extends JdbcDatabaseContainer<SELF> {
[...]
}
JdbcDatabaseContainer<?> container = new PostgreSQLContainer<>();
```

Listing 15: PostgreSQLContainer

In unseren Tests müssen wir dem WebDriver mitteilen, wo unsere Spring-Boot-Applikation mit randomisiertem Port denn gerade zu finden ist. Auch hierfür nutzen wir ein JUnit-Callback *(siehe Listing 13)*. Unser bestehender Spring-Selenium-Test bedarf für Testcontainers dann nur ein paar zusätzlicher Extensions *(siehe Listing 14)*.

The Good, the Bad and the Ugly

Da nicht nur Medaillen zwei Seiten besitzen, sondern bekanntlich auch Konzepte und Technologien, wollen wir unsere Eindrücke von Testcontainers abschließend noch einmal Revue passieren lassen. Ausgehend von einem eingänglichen Fluent-API ermöglicht das Framework ein schnelles Aufsetzen von Containern zur isolierten Durchführung von Integrationstests. Ausgesprochen lobenswert ist die gleichermaßen ausführlich wie verständlich gestaltete Dokumentation *[1]*, die die Funktionsweise mit knackigen Codebeispielen intuitiv vermittelt. Hier kann sich manch anderes Projekt noch eine Scheibe abschneiden! Als durchdacht ist auch die Integration mit dem populären Java-Testframework JUnit zu bezeichnen, insbesondere mit der aktuellen Version 5. Durch ihren modularen Aufbau und die Unterstützung beliebiger Images sind dem Einsatz der Technologie kaum Grenzen gesetzt.

Prinzipbedingt erhalten wir durch Docker eine stärkere Isolation der Tests, die allerdings auch ihren Preis hat. Bereits durch das Hoch- und Runterfahren der Container verlangsamt sich die Ausführung mindestens um den Faktor zehn; berücksichtigt man zusätzlich auch das Aufsetzen der Umgebung, landen wir in unserem Setup gar bei Faktor 20. Gerade mit Hinblick auf eine große Testbasis und regelmäßige Builds ist das zu viel. In der Praxis sollten Entwickler also weise entscheiden, welche Tests sie „dockerisieren", ob sie regelmäßig genutzte Testdaten nicht besser gleich mit dem Image ausliefern und ob ein frisch aufgesetzter Container tatsächlich für jede Testmethode benötigt wird. Stichwort Performance: Aktuell unterstützt die offizielle Testcontainers-JUnit-Extension (und auch unser DatabaseContainerHolder) lediglich eine serielle Ausführung der Tests, was bei der Konfiguration des Build-Servers zu bedenken ist. Wie wir in den vergangenen Kapiteln gesehen haben, kommt eine (praxisgerechte) Integration mit Spring nicht out of the box, sondern bedarf initial etwas Handarbeit.

Beschäftigt man sich mit einer neuen Technologie, ist man zunächst vom Hochglanz der Verpackung angetan. Schaut man dann (auch bei namhaften Frameworks!) einmal hinter die Fassade, läuft einem zuweilen ein kalter Schauer über den Rücken. Ein besonderes Leckerchen hält die Definition des von uns geschätzten PostgreSQL-Container bereit *(siehe Listing 15)*.

Wollen wir die Klassen syntaktisch korrekt verwenden, müssen wir in der Konsequenz stets eine Generic Wildcard mit uns herumtragen. Das geht auch anders!

Der vollständige Source Code zu diesem Artikel kann in unserem Repository *[11]* bei GitHub eingesehen werden.

Quellen

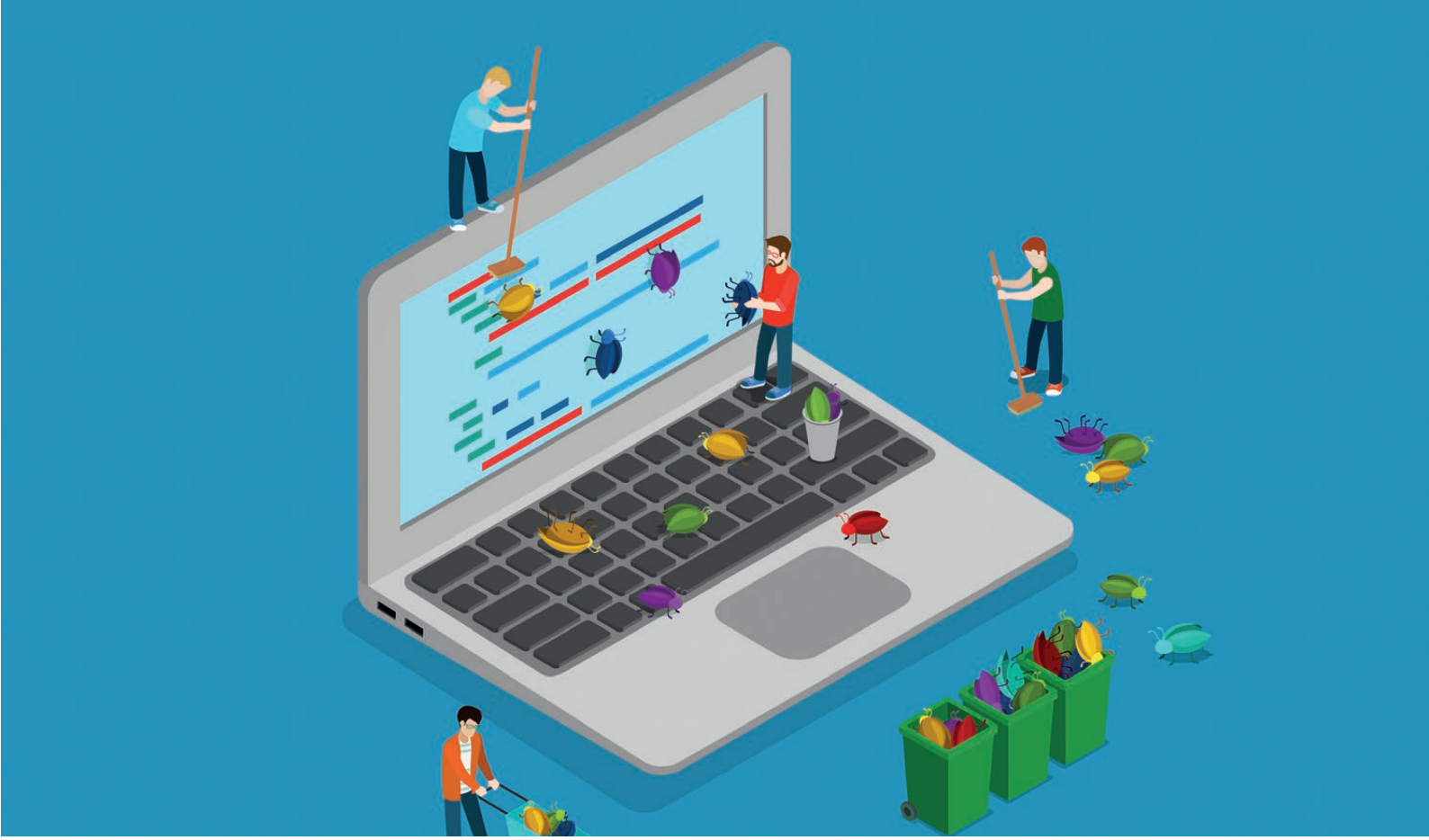
[1] *https://www.testcontainers.org*
[2] *https://martinfowler.com/bliki/TestPyramid.html*
[3] *https://www.h2database.com*
[4] *https://spring.io/projects/spring-boot*
[5] *https://selenium.dev*
[6] *https://github.com/docker-java/docker-java*
[7] *https://www.docker.com*
[8] *https://junit.org/junit5*
[9] *https://flywaydb.org*
[10] *https://de.wikipedia.org/wiki/42_(Antwort)*
[11] *https://github.com/anderscore-gmbh/Testcontainers-JavaAktuell*

Maik Wolf
anderScore GmbH
maik.wolf@anderscore.com

Maik Wolf ist bei der Kölner anderScore GmbH als Fullstack-Entwickler für Java-Enterprise-Projekte und Agile Coach im Kundeneinsatz tätig. Durch seine Expertise in den Branchen Logistik, Managed-Hosting, Groß- und Einzelhandel sowie Dialogmarketing verfügt er über vielseitige und intensive Einsichten in verschiedene Softwarelandschaften und in ganz unterschiedliche Geschäftsanforderungen. Da sein zweites Herz für agile Arbeitsmethoden schlägt, integriert und optimiert er diese in seinen Projekten.

Daniel Krämer
anderScore GmbH
daniel.kraemer@anderscore.com

Daniel Krämer ist bereits seit mehreren Jahren als Software Engineer für die anderScore GmbH tätig und begeistert sich für durchdachte Software-Architektur und strukturiertes Design. In seinen Kundenprojekten setzt er sich überwiegend mit individueller Java- und Web-Entwicklung sowie Fragestellungen rund um Migration und Integration (zum Beispiel Microservices) auseinander. Bei seinen Kollegen ist er dafür bekannt, auch für komplexe Szenarien effektive und effiziente Strategien zur Testautomatisierung zu finden. In Ergänzung zur Arbeit mit dem Code macht es Daniel auch Spaß, seine Kenntnisse und Erfahrungen mit anderen Entwicklern zu teilen. Zu den Inhalten regelmäßig abgehaltener, öffentlicher Trainings zählen Themen wie Microservices, Spring, Apache Wicket und jQuery.



Automatisierte Qualitätssicherung von Ansible Playbooks

Sandra Parsick

Dieser Text wurde im Englischen im Cloud Report 01/2020 veröffentlicht

Wenn Serverinstanzen in der Cloud provisioniert werden, werden sie selten manuell aufgesetzt, sondern mithilfe von Provisionierungsskripten automatisiert. Diese beschreiben, wie der Server aufgesetzt werden soll. Dabei handelt es sich um normalen Code, so wie wir es von der normalen Softwareentwicklung kennen, nur dass dieser Code auf die Domäne Infrastruktur spezialisiert ist. In der Softwareentwicklung haben sich statische Code-Analyse („linten") und automatisierte Tests als Mittel für gute Wartbarkeit des Produktionscodes etabliert. Warum diese gute Praxis nicht auch für Infrastructure-as-Code (IaC) anwenden? Dieser Artikel beschreibt anhand des Provisionierungswerkzeugs Ansible, wie eine automatisierte Qualitätssicherung für Provisionierungsskripte aussehen kann.

Ansible beschreibt sich selbst als Werkzeug für das Konfigurationsmanagement, für die Verteilung von Software und für die Ausführung von Ad-hoc-Kommandos *[1]*. Ansible ist in Python geschrieben. Um es zu benutzen, muss man nicht zwangsläufig Python-Kenntnisse haben, da die Ansible-Skripte in YAML geschrieben sind. YAML ist eine Abstraktion von JSON mit dem Unterschied, dass sie besser lesbar ist *[2]*. Diese Ansible-Skripte werden „Playbooks" genannt.

Damit Ansible eingesetzt werden kann, muss auf den Zielmaschinen Python installiert und ein Zugriff auf diese Maschinen über SSH möglich sein. Ansible ist nur auf der Maschine zu installieren, von der die Playbooks ausgeführt werden („Host-Maschine"); sie muss ein Linux-System sein. Für Windows auf den Zielmaschinen bietet Ansible eine rudimentäre Unterstützung an. Wenn die in YAML geschriebenen Playbooks ausgeführt werden, übersetzt Ansible diese in Python-Skripte und führt sie auf den Zielmaschinen aus. Anschließend werden sie wieder von den Zielmaschinen entfernt. Ein typisches Ansible-Playbook sieht wie in *Listing 1* gezeigt aus. Für einen tieferen Einstieg in Ansible empfiehlt sich mein Artikel „Ansible für Entwickler" *[3]*.

Funktionsweise einer QA-Pipeline für Ansible-Playbooks

In der Softwareentwicklung werden statische Code-Analyse und automatisierte Tests kontinuierlich in einer Pipeline auf einem CI-Server durchgeführt. Dieses Konzept soll für Ansible-Playbooks wiederverwendet werden.

Als Erstes legen wir die Provisionierungsskripte in ein Versionskontrollsystem ab. Änderungen an den Skripten, die im Versionskontrollsystem eingecheckt werden, triggern eine Pipeline an. Eine Pipeline für Provisionierungsskripte durchläuft mehrere Schritte *(siehe Abbildung 1)*: Zuerst überprüft ein Lint den Code daraufhin, ob dieser syntaktisch richtig ist und Best Practices folgt. Wenn der Lint nichts zu beanstanden hat, wird eine Testumgebung vorbereitet. Gegen diese Testumgebungen laufen die Provisionierungsskripte. Wenn sie ohne Fehler durchgelaufen sind, werden Tests ausgeführt, um zu überprüfen, ob alles so konfiguriert wurde wie erwartet. Am Ende wird die Testumgebung wieder abgebaut.

Im Folgenden werden die einzelnen Schritte genauer vorgestellt und gezeigt, wie sie in einer Pipeline eingebunden werden. Das wird mithilfe eines Git-Repository demonstriert, das ein Ansible-Playbook enthält, Skripte, um die Testumgebung aufzubauen, die Tests und eine Beschreibung der Pipeline. Das Ansible-Playbook wird ein OpenJDK 8 und eine Tomcat-Instanz installieren *(siehe Listing 2)*. Das komplette Projekt ist auf GitHub zu finden *[4]*. Als Startpunkt dient das Git-Repository, das erstmal nur das Ansible-Playbook erhält.

```
- hosts: ansible-test-instance
  vars:
  tomcat_version: 9.0.27
  tomcat_base_name: apache-tomcat-{{ tomcat_version }}
  #catalina_opts: "-Dkey=value"
  tasks:
  - name: install java
    apt:
  name: openjdk-8-jdk
  state: present
    become: yes
    become_method: sudo
```

Listing 1

```
.
├── LICENSE
├── README.md
└── setup-tomcat.yml
```

Listing 2

Statische Code-Analyse mit ansible-lint

Zuerst überprüft ein Lint, ob die Ansible-Playbooks syntaktisch korrekt sind und ob sie nach Best-Practices-Regeln geschrieben sind. Für Ansible-Playbook gibt es den Lint „*ansible-lint*" *[5]*. Es wird auf der CLI ausgeführt *(siehe Listing 3)*.

In diesem Beispiel findet ansible-lint mehrere Regelverstöße. Sollen einzelne Regeln nicht gecheckt werden, gibt es zwei Möglichkeiten, diese zu exkludieren. Einmal können die Regeln global für das ganze Projekt ausgeschaltet werden oder für einzelne Fälle (sogenannte „false-positive" Fälle). Für die globale Einstellung legen wir eine .ansible-lint-Datei im Root-Verzeichnis des Projektes ab *(siehe Listing 4)*.

Abbildung 1: Alle Schritte einer Pipeline (Quelle: Sandra Parsick)

```
$ ansible-lint setup-tomcat.yml
[502] All tasks should be named
setup-tomcat.yml:30
Task/Handler: file name=/opt __file__=setup-tomcat.yml __line__=31 mode=511
owner=tomcat group=tomcat

[502] All tasks should be named
setup-tomcat.yml:57
Task/Handler: find patterns=*.sh paths=/opt/{{ tomcat_base_name }}/bin
```

Listing 3

```
.
├── .ansible-lint
├── LICENSE
├── README.md
└── setup-tomcat.yml
```

Listing 4

```
skip_list:
- skip_this_tag
- and_this_one_too
- skip_this_id
- '401
```

Listing 5

In dieser Konfigurationsdatei pflegen wir eine *exclude*-Liste *(siehe Listing 5)*. Darin können wir noch weitere Verhaltensweisen konfigurieren, wie zum Beispiel, in welchem Pfad sie abgelegt sind. Mehr Informationen dazu auf der Projektseite *[4]*.

Möchten wir false-positive Fälle vom Check herausnehmen, dann hinterlassen wir im Ansible-Playbook einen entsprechenden Kommentar dazu *(siehe Listing 6)*.

Fehlen uns noch weitere Regeln, dann können wir diese mithilfe von Python-Skripten selbst definieren *(siehe auch die Dokumentation [5])*.

Testumgebung aufsetzen und abbauen mit Terraform

Nachdem der Lint erfolgreich durchgelaufen ist, soll die Testumgebung für die Ansible-Playbooks aufgebaut werden. Das wird mithilfe von Terraform *[6]* und der Hetzner Cloud *[7]* gemacht. Terraform hilft uns, Cloud-Infrastruktur mit Code zu provisionieren.

Bevor wir mit dem Terraform-Skript loslegen können, müssen wir im Hetzner-Cloud-Konto einen Public-SSH-Key ablegen und ein API-Token generieren. Der Public-SSH-Key wird später in die anzulegende Server-Instanz abgelegt, damit Ansible sich mit dieser Instanz verbinden kann.

```
- file: # noqa 502
    name: /opt
    mode: 0777
    owner: tomcat
    group: tomcat
  become: yes
  become_method: sudo
```

Listing 6

Mithilfe von Terraform beschreiben wir, welchen Server-Typ wir haben wollen, welches Betriebssystem, in welchem Standort die Server-Instanz gehostet und was als Grund-Setup provisioniert werden soll. Dazu legen wir im Root-Verzeichnis des Projekts eine *testinfrastrucuture.tf*-Datei an *(siehe Listing 8)*.

Als Grund-Setup geben wir an, welcher Public-SSH-Key auf dem Server abgelegt werden soll (*ssh_keys*) und dass Python installiert wird (*provisioner remote-exec*) *(siehe Listing 9)*. Der Public-SSH-Key und Python werden benötigt, damit später Ansible seine Skripte auf diesem Server ausführen kann.

Da die Test-Server-Instanz in der Hetzner-Cloud betrieben werden soll, muss Terraform das benötigte Provider-Plug-in nachinstallieren. Dazu rufen wir „*terraform init*" im Ordner auf, in dem *testinfrastructure.tf* liegt. Dann ist alles vorbereitet, um die Server-Instanz zu provisionieren.

Wir müssen dem apply-Befehl die Variable hcloud_token mit dem API-Token, den wir vorher in der Hetzner-Cloud-Konsole generiert haben, mitgeben *(siehe Listing 10)*. Sobald der Server zur Verfügung steht, können wir die Ansible Playbooks und die Tests gegen diesen Server ausführen. Unabhängig vom Erfolg der Playbooks oder der Tests wird die Serverinstanz mithilfe von Terraform wieder abgebaut. Auch beim destroy-Befehl muss das API-Token mitgegeben werden *(siehe Listing 11)*.

Ansible-Playbooks ausführen

Nachdem wir die Server-Instanz mit Terraform erstellt haben, führen wir Ansible-Playbooks gegen diese Server-Instanz aus. In einer klassischen Infrastruktur hätte die Server-Instanz eine feste IP-Adresse und wir hätten diese IP-Adresse in Ansibles sogenanntes „statisches Inventory" eingetragen *(siehe Listing 12)*, damit Ansible weiß, mit welchem Server es sich verbinden soll.

Da in der Cloud die Server bei jedem Bereitstellen eine neue IP-Adresse zugeteilt bekommen, können wir in diesem Fall nicht das statische Inventory benutzen. Da wir gegen die Hetzner-Cloud gehen, verwenden wir das Ansible-Inventory-Plug-in „*hcloud*". Dazu muss ein *inventory*-Ordner mit der Datei *test.hcloud.yml* erstellt werden *(siehe Listing 13)*. Beim Dateinamen ist das Suffix „*hcloud.yml*" wichtig *(siehe Listing 14)*.

Im Anschluss müssen wir den richtigen Servernamen, den wir vorher im Terraform-Skript definiert haben, im Playbook unter „*hosts*" eintragen *(siehe Listing 15)*.

Beim Ausführen der Playbooks müssen wir noch darauf achten, dass wir den richtigen Private-SSH-Key mitgeben und dass das API-

```
from ansiblelint import AnsibleLintRule
class DeprecatedVariableRule(AnsibleLintRule):
  id = 'ANSIBLE0001'
  shortdesc = 'Deprecated variable declarations'
  description = 'Check for lines that have old style ${var}'
  + 'declarations'
  tags = { 'deprecated' }
  def match(self, file, line):
     return '${' in line
```

Listing 7

Token in der System-Environment-Variablen *HCLOUD_TOKEN* definiert ist *(siehe Listing 16)*.

Das API-Token können wir auch in der Inventory-Datei *test.hcloud.yml* definieren. Da diese Datei aber in einem Version-Control-System abgelegt wird, ist davon abzuraten, da keine Credentials in einem VCS gespeichert werden sollten.

```
.
└──
├──.ansible-lint
├──LICENSE
├──README.md
├──setup-tomcat.yml
├──terraform.tfvars
└──testinfrastructure.tf
```

Listing 8

```
# testinfrastructure.tf
variable "hcloud_token" {}
variable "ssh_key" {}
# Configure the Hetzner Cloud Provider
provider "hcloud" {
  token = var.hcloud_token
}
# Create a server
resource "hcloud_server" "ansible-tests" {
  name = "ansible-tests"
  image = "ubuntu-18.04"
  server_type = "cx11"
  location = "nbg1"
  ssh_keys = [
    "ansible-test-infrastructure"
  ]
  provisioner "remote-exec" {
    inline = [
    "while fuser /var/lib/apt/lists/lock >/dev/null 2>&1; do sleep 1; done",
    "apt-get -qq update -y",
    "apt-get -qq install python -y",
  ]
  connection {
    type = "ssh"
    user = "root"
    private_key = file(var.ssh_key)
    host = hcloud_server.ansible-tests.ipv4_address
  }
 }
}
```

Listing 9

```
terraform apply -var="hcloud_token=…"
```

Listing 10

```
terraform destroy -var="hcloud_token=…"
```

Listing 11

Funktionale Tests mit Testinfra

Nachdem die Testumgebung aufgesetzt wurde und die Playbooks erfolgreich durchliefen, sollen noch Tests ausgeführt werden, die überprüfen, ob das OpenJDK-Package installiert wurde und ob die Datei */opt/tomcat/bin/catalina.sh* auf dem Server existiert.

Es existieren einige Testframeworks für Provisionierungswerkzeuge, zum Beispiel ServerSpec *[8]*, Goss *[9]* und Testinfra *[10]*. Der Hauptunterschied zwischen den Testframeworks ist, in welcher Syntax die Tests geschrieben werden. Bei ServerSpec werden die Tests in Ruby-Syntax geschrieben, bei Goss in YAML-Syntax und bei Testinfra in Python-Syntax. Die Arbeitsweise der Testframeworks ist gleich. Sie verbinden sich auf einen Server, der vorher mit Provisionierungswerkzeugen provisioniert wurde, und prüfen, ob die Serverprovisionierung den Testbeschreibungen entspricht.

Hier werden die Tests mit Testinfra geschrieben. Dazu legen wir im Root-Verzeichnis des Projekts den Ordner „*tests*" an. Dort werden

```
#statische Inventory
[ansible-tests]
78.47.150.245
```

Listing 12

```
# aktuelle Projektstruktur
.
├── .ansible-lint
├── ansible.cfg
├── inventory
│   └── test.hcloud.yml
├── LICENSE
├── README.md
├── setup-tomcat.yml
├── terraform.tfvars
└── testinfrastructure.tf
```

Listing 13

```
# test.hcloud.yml
plugin: hcloud
```

Listing 14

```
Ansible-Playbook Snippet
- hosts: ansible-test-instance
```

Listing 15

```
$ export HCLOUD_TOKEN=...
$ ansible-playbook --private-key=/home/sparsick/.ssh/id_hetzner_ansible_test -i
inventory/test.hcloud.yml setup-tomcat.yml
```

Listing 16

```
.
├── .ansible-lint
├── ansible.cfg
├── inventory
│   └── test.hcloud.yml
├── LICENSE
├── README.md
├── setup-tomcat.yml
├── terraform.tfvars
├── testinfrastructure.tf
└── tests
    └── test_tomcat.py
```

Listing 17

```
# test_tomcat.py
def test_openjdk_is_installed(host):
  openjdk = host.package("openjdk-8-jdk")
  assert openjdk.is_installed

def test_tomcat_catalina_script_exist(host):
  assert host.file("/opt/tomcat/bin/catalina.sh").exists
```

Listing 18

die Tests abgelegt *(siehe Listing 17)*. In *test_tomcat.py* schreiben wir die Tests *(siehe Listing 18)*.

Testinfra bringt fertige Module mit, die die Testbeschreibung vereinfachen. In diesen Tests benutzen wir zum Beispiel *host.package*, um den Package-Manager abfragen zu können, oder *host.file*, um die Existenz einer bestimmten Datei zu testen.

Testinfra unterstützt verschiedene Möglichkeiten, um sich mit dem Server zu verbinden. Eine Möglichkeit ist, die Verbindungskonfiguration von Ansible wiederzuverwenden. Da Ansible hier ein dynamisches Inventory benutzt und Testinfra nicht alle Informationen aus diesem dynamischen Inventory lesen kann, müssen wir einige Konfigurationen explizit in der Ansible-Konfigurationsdatei *ansible.cfg* eintragen *(siehe Listing 19)*. Diese Datei wird im Root-Verzeichnis des Projektes abgelegt. Dann können wir die Tests ausführen *(siehe Listing 20)*.

Zusammenführung in einer Pipeline

Nachdem jeder Schritt in der Pipeline einzeln betrachtet wurde, sollen die Schritte in eine Pipeline im CI-Server Jenkins zusammen-

```
# ansible.cfg
[defaults]
remote_user=root
private_key_file = /home/sparsick/.ssh/id_hetzner_ansible_test
```

Listing 19

```
$ py.test --connection=ansible --ansible-inventory=inventory/test.hcloud.yml --force
-ansible -v tests/*.py
============================================================
===================== test session starts ==================
============================================================
platform linux2 -- Python 2.7.15+, pytest-3.6.3, py-1.5.4, pluggy-0.13.0 --
/usr/bin/python
cachedir: .pytest_cache
rootdir: /home/sparsick/dev/workspace/ansible-testing-article, inifile:
plugins: testinfra-3.2.0
collected 2 items
tests/test_tomcat.py::test_openjdk_is_installed[ansible://ansible-test-instance]
PASSED
[ 50%]
tests/test_tomcat.py::test_tomcat_catalina_script_exist[ansible://ansible-test-
instance] PASSED
[100%]
============================================================
================== 2 passed in 11.52 seconds ===============
============================================================
```

Listing 20

```
# Jenkinsfile
pipeline {
  agent any
  environment {
    HCLOUD_TOKEN = credentials('hcloud-token')
  }
  stages {
    stage('Lint Code') {
      steps {
        sh 'ansible-lint setup-tomcat.yml'
      }
    }
    stage('Prepare test environment') {
      steps {
        sh 'terraform init'
        sh 'terraform apply -auto-approve -var="hcloud_token=$   {HCLOUD_TOKEN}"'
      }
    }
    stage('Run Playbooks') {
      steps {
        sh 'ansible-playbook -i inventory/test.hcloud.yml setup-tomcat.yml'
      }
    }
    stage('Run Tests') {
      steps {
        sh 'py.test --connection=ansible -ansible -inventory=inventory/test.hcloud.yml --force-ansible -v tests/*.py'
      }
    }
}
post {
  always {
    sh 'terraform destroy -auto-approve -var="hcloud_token=${HCLOUD_TOKEN}"'
  }
 }
}
```

Listing 21

geführt werden. Die Pipeline beschreiben wir in einem Jenkins-File. Diese Jenkins-File beschreibt vier Stages und eine Post-Action. Je eine Stage für den Code-Check, das Aufbauen der Testumgebung und das Ausführen der Playbooks und die letzte Stage für die Ausführung. In der Post-Action wird die Testumgebung abgebaut, egal, ob in den Stages Fehler auftraten *(siehe Listing 21)*. Damit diese Pipeline auch funktioniert, müssen wir in Jenkins das API-Token der Hetzner-Cloud hinterlegen. Damit das Token nicht im Klartext gespeichert wird, legen wir es im Bereich „*Credentials*" als *Secret Text* ab und vergeben eine ID, die dann per *credential*-Methode im Jenkins-File wieder abgerufen werden kann (hier: *hcloud-token*).

Vereinfachung für Ansible Role

Für Ansible Role, einer Strukturierungsmöglichkeit in Ansible, um Playbooks über mehrere Projekte wiederzuverwenden, gibt es eine Vereinfachung für diese vorgestellte Pipeline. Bei Ansible Role können wir mithilfe von Molecule *[11]* die komplette Pipeline und die vorgestellten Werkzeuge in einem Rutsch konfigurieren. Wir benötigen dann auch nur einen Befehl (*molecule test*), um die komplette Pipeline auszuführen. Eine sehr gute Einführung in Molecule gibt der Blog Post „Test-driven infrastructure development with Ansible & Molecule" *[12]* von Jonas Hecht.

Weiterführende Informationen

[1] *https://docs.ansible.com/*
[2] *https://en.wikipedia.org/wiki/YAML*
[3] *https://www.sandra-parsick.de/publication/ansible-fuer-dev/*
[4] *https://github.com/sparsick/ansible-testing-article/tree/cloudreport19*
[5] *https://github.com/ansible/ansible-lint*
[6] *https://www.terraform.io/*
[7] *https://www.hetzner.com/cloud*
[8] *https://serverspec.org/*
[9] *https://github.com/aelsabbahy/goss*
[10] *https://github.com/philpep/testinfra*
[11] *https://github.com/ansible/molecule*
[12] *https://blog.codecentric.de/en/2018/12/test-driven-infrastructure-ansible-molecule/*

Sandra Parsick
mail@sandra-parsick.de

Sandra Parsick ist als freiberufliche Softwareentwicklerin und Consultant im Java-Umfeld tätig. Seit 2008 beschäftigt sie sich mit agiler Softwareentwicklung in verschiedenen Rollen. Ihre Schwerpunkte liegen im Bereich der Java-Enterprise-Anwendungen, agilen Methoden, Software Craftsmanship und in der Automatisierung von Softwareentwicklungsprozessen. Darüber schreibt sie gerne Artikel und spricht gerne auf Konferenzen. In ihrer Freizeit engagiert sich Sandra Parsick in der Softwerkskammer Ruhrgebiet, einer Regionalgruppe der Software Craftmanship Community im deutschsprachigen Raum. Seit 2019 ist sie Mitglied im Oracle Groundbreaker Ambassador Programm.

Grüne Inseln im Schlamm – Mit Side-by-Side Refactoring allzeit lieferbereit, Teil 1

Georg Berky, Valtech Mobility

Dieser Artikel zeigt anhand von Emily Baches „Gilded Rose"-Kata [1] eine fortgeschrittene Art des Refactoring von Legacy Code, bei der der Code jederzeit lieferbar bleibt, auch wenn das Refactoring unterbrochen werden muss. Dadurch wird der Code auch schrittweise wieder wartbar gemacht. Dies ist Teil 1 des vollständigen Artikels.

Legacy Code ist eine große Herausforderung für Entwickler. Das ursprüngliche Team, das ihn entwickelt hat, ist nicht mehr da. Der Code hat nur schlechte oder keine Dokumentation und schon gar keine Tests. Ändert man an der einen Stelle etwas im Code, tritt in einer komplett unerwarteten Klasse ein seltsamer Fehler auf. Man spricht von Brownfield oder Legacy Code – als würde man tief im Schlamm eines Feldes waten, wo jeder Schritt viel zu viel Kraft kostet. Es braucht Zeit, Nerven und Ressourcen, solchen Code wieder in einen wartbaren Zustand zu bringen.

Vor Kurzem bin ich auf Twitter über folgendes Zitat von Michael Feathers gestolpert: „Key lesson is, you can always (read: most of the time) do greenfield in a legacy codebase. [...]"

Dieses Zitat habe ich zum Anlass genommen, nochmal den Dschungel an Literatur und Videos zum Thema Legacy Code zu durchsuchen, und ein paar Techniken mitgebracht, die die IDE nicht automatisch ausführen kann, aber dafür an schwierigen Stellen oft umso wirkungsvoller sind.

Sich um Software kümmern

Software lebt zusammen mit den Menschen, die sie pflegen. Der Code beeinflusst die Menschen. Die Menschen beeinflussen den Code. Obwohl Code komplett substanzlos ist, gehen wir froh nach Hause, wenn wir ihm etwas Gutes getan haben, und haben schlechtere Laune, wenn wir ihn auch nach mehreren Anläufen nicht verstehen konnten oder vergeblich versucht haben, eine schwierige Stelle zu verschönern. Code, der nicht mehr gepflegt wird, wird langsam unbenutzbar. Bibliotheken veralten, man findet Security-Schwachstellen, die Version der Programmiersprache steigt. Code braucht also konstante Pflege und Aufmerksamkeit, nur um benutzbar zu bleiben – eine Aufgabe, die deswegen zur regulären Arbeit der Entwickler gehören sollte. Ein klassischer Werkvertrag, der nur Entwicklung und Abnahme des Codes beinhaltet, greift also oft zu kurz und sorgt langfristig für Probleme beim Kunden. Das sollte bei Auftragsverhandlungen besprochen und berücksichtigt werden.

Idealerweise sollte der Benutzer des Codes auch nicht direkt bemerken, dass wir Änderungen vornehmen. Das System sollte weiterhin wie gehabt funktionieren. Im Folgenden möchte ich einen Ansatz für ein von mir oft benutztes Refactoring zeigen, bei dem der Code jederzeit release- und lieferbar bleibt, auch wenn das Refactoring noch nicht abgeschlossen ist. So pflanzen wir kleine grüne Inseln im Brownfield, die im Laufe der Zeit größer werden und so den Code schrittweise verbessern.

Als Simulation eines Legacy-Systems möchte ich das „Gilded Rose Kata" *[1]* benutzen.

Code, den wir nicht ändern können

Manchmal haben wir Code vor uns liegen, den wir nicht ändern können. Das könnte die Bibliothek eines Drittanbieters sein, die wir nur binär, aber nicht im Quelltext vorliegen haben. Vielleicht ist der Anbieter auch längst vom Markt verschwunden oder passt die Bibliothek aus anderen Gründen nicht mehr an.

Auch in der „Gilded Rose" findet sich ein Beispiel dafür: „However, do not alter the `Item` class or property as those belong to the goblin in the corner who will insta-rage and one-shot you as he doesn't believe in shared code ownership." Meine Übersetzung davon: „Die Klasse `Item` oder ihre Properties darfst du trotzdem nicht verändern. Sie gehört dem Goblin in der Ecke. Er glaubt nicht an geteilte Verantwortung für den Code und haut dich sonst sofort wie ein Berserker auf einen Schlag um."

Die Klasse `Item` hat ein Design-Problem: Alle Felder sind `public`, jede Klasse kann darin herumschreiben und die Logik, die mit Daten aus `Item` arbeitet, liegt natürlich nicht in `Item`. Der Code Smell dazu heißt „Feature Envy" *[2]* beziehungsweise „Anemic Domain Model" *[3]* – je nachdem, von welcher Seite aus man darauf blickt.

Wenn wir also nicht wollen, dass uns der Goblin in der Ecke umhaut, weil wir es gewagt haben, seinen Code anzufassen, dann können wir `Item` nicht ändern. Wenn wir etwas nicht reparieren können, können wir es vielleicht codetechnisch vergolden.

Im Legacy Code kann man dafür die Technik „Wrap Class" *[4]* aus „Working Effectively with Legacy Code" (WELC) verwenden.

Das Sicherheitsnetz beim Refactoring

Vor jedem Refactoring baue ich jedoch ein Sicherheitsnetz durch Characterization-Tests auf. Diese lasse ich nach jedem kleinen Refactoring-Schritt laufen, besonders, wenn ich unsicher bin, ob ich gerade etwas kaputt gemacht habe. Je unsicherer ich bin, desto kleiner wähle ich meine Schritte. Im Idealfall komme ich durch ein einziges „Undo" wieder zum letzten grünen Stand des Codes zurück. Nach einigen Zwischenschritten stage ich meine Änderungen mit `git add` oder committe lokal mit `git commit`.

Beim Schreiben der Tests hilft mir die Funktion „run with coverage" meiner IDE, die mir zeigt, ob ich beim Schreiben der Tests Stellen im Code übersehen habe. Hundert Prozent Coverage garantieren nicht, dass ich nichts vergessen habe, aber ein offensichtliches Loch ist trotzdem der Beweis, dass noch ein Test für diesen Teil des Codes fehlt.

Der gebotenen Kürze wegen kann ich hier nicht im Detail auf Characterization-Tests eingehen. Im Prinzip funktionieren sie so, dass man, statt direkt Erwartungen an den Code zu formulieren, die wirklich gelieferten Ergebnisse eines Funktionsaufrufs ansieht, verifiziert und zum Schluss in Tests festhält, die das Verhalten dokumentieren.

In echten Systemen muss man oft auch noch Dependency-Breaker einsetzen, um beispielsweise den Zugriff auf Datenbank oder Dateien durch Stubs zu ersetzen.

Details finden sich in Michael Feathers' Blog *[4]* und in WELC *[5]*.

Die Bruchstellen vergolden

Zuerst wende ich mich also der `Item`-Klasse zu. Ich kann sie selbst nicht ändern, aber ich kann ändern, wer sie benutzt. Die Klasse enthält nur Daten, also kann ich sie auch zu einem nicht ganz perfekten Field einer schöner designten Model-Klasse machen, die zum Wrapper um `Item` wird. Nicht ganz perfekt ist ok. Code ist organisch. Organisch ist nicht perfekt.

Also fange ich mit dem Wrapper an *(siehe Listing 1)*. Das ist noch nicht viel, aber wenn wir fertig sind, soll `GildedItem` die einzige Klasse sein, die jemals eine Instanz von `Item` manipuliert.

Wir wissen jetzt, was wir mit `Item` machen wollen, aber noch nicht, wie wir diese Änderung am Code so sicher machen, dass wir jederzeit lieferfähig sind, also so, dass wir jederzeit bei grünen Tests committen, pushen, releasen und liefern können. Die Zeit, in der die Tests dabei rot bleiben, soll so kurz wie möglich sein. Das bedeutet, dass wir kleine, sichere Schritte machen müssen. Damit das möglich wird, müssen wir ein paar Schritte in die Gegenrichtung machen: Der Weg beim „Preparatory Refactoring" *(siehe Abbildung 1)* führt zuerst zurück, um dann an einem Punkt auf die befestigte Straße zu treffen, die uns letztendlich schneller ans Ziel bringt als der *direkte* Weg durch den Wald. Bis zu diesem Punkt macht man manchmal Schritte rückwärts, bevor man wirklich anfangen kann, sauber zu machen.

Mein erster Schritt rückwärts: Ich benutze `Item` und `GildedItem` nebeneinander *(siehe Listing 2)*. `GildedRose` hat jetzt ein zweites Field, ein Array mit Instanzen von `GildedItem`. Es liegt auch noch neben dem alten Array mit `Item`. Dazu habe ich einen zweiten Konstruktor eingeführt, der ein Array von `GildedItem` als Parameter akzeptiert. Den bestehenden Konstruktor habe ich auch modifiziert. Um alles noch schlimmer zu machen, erstellen beide Konstruktoren zusätzliche Instanzen von `Item` oder `GildedItem`, die auch noch aufeinander *aliased* sind. Wenn ich ein `Item` ändere, ändere ich auch das dazugehörige `GildedItem`, das es wrappt. Unter normalen Umständen würden wir solche unsichtbaren Änderungen an gekapselten Objekten um jeden Preis vermeiden wollen. In diesem Fall ist es genau das, was wir wollen: `GildedRose` kann dadurch wie gehabt seine Instanzen von `Item` manipulieren, ohne dass ich sofort alle ändernden Stellen im Code umbauen muss. Gleichzeitig sieht `GildedItem` durch das *Aliasing* diese Änderungen. In der Gegenrichtung manipuliert auch `GildedItem` dieselben Instanzen und `GildedRose` sieht dessen Änderungen. Beide Arrays sind `private`. Die Unordnung hat also wenig Chancen, andere Klassen zu beeinflussen. Nach einer Änderung kann ich lokal committen, sobald die Tests wieder grün sind. So kann ich jederzeit den letzten grünen Stand wiederherstellen. Dank des Schrittes zurück kann ich während des Umbaus jetzt jederzeit liefern, solange meine Tests grün sind.

```
public class GildedItem {

    private final Item item;

    public GildedItem(Item item) {
        this.item = item;
    }

    Item getItem() {
        return item;
    }
}
```

Listing 1: Wrapper für Item

```
class GildedRose {

    private final GildedItem[] gildedItems;
    private final Item[] items;

    public GildedRose(Item[] items) {
        this.items = items;
        this.gildedItems = Arrays.stream(items)
                .map(GildedItem::new)
                .toArray(GildedItem[]::new);
    }

    public GildedRose(GildedItem[] items) {
        this.items = Arrays.stream(items)
                .map(GildedItem::getItem)
                .toArray(Item[]::new);
        this.gildedItems = items;
    }
}
```

Listing 2: Wrapper und Item Side-by-Side

Die Tests umziehen

Wo wir gerade bei Tests sind: Diese haben zu Beginn den Konstruktor benutzt, der ein Array von `Item` annimmt. Das habe ich so gelassen, bis ich mit meiner Änderung am Produktivcode fertig war. Ich versuche zu vermeiden, Produktiv- und Testcode gleichzeitig zu ändern, weil sie sich gegenseitig testen. Ich würde sonst das Sicherheitsnetz verlieren, das ich mir vorher durch die Characterization-Tests erar-

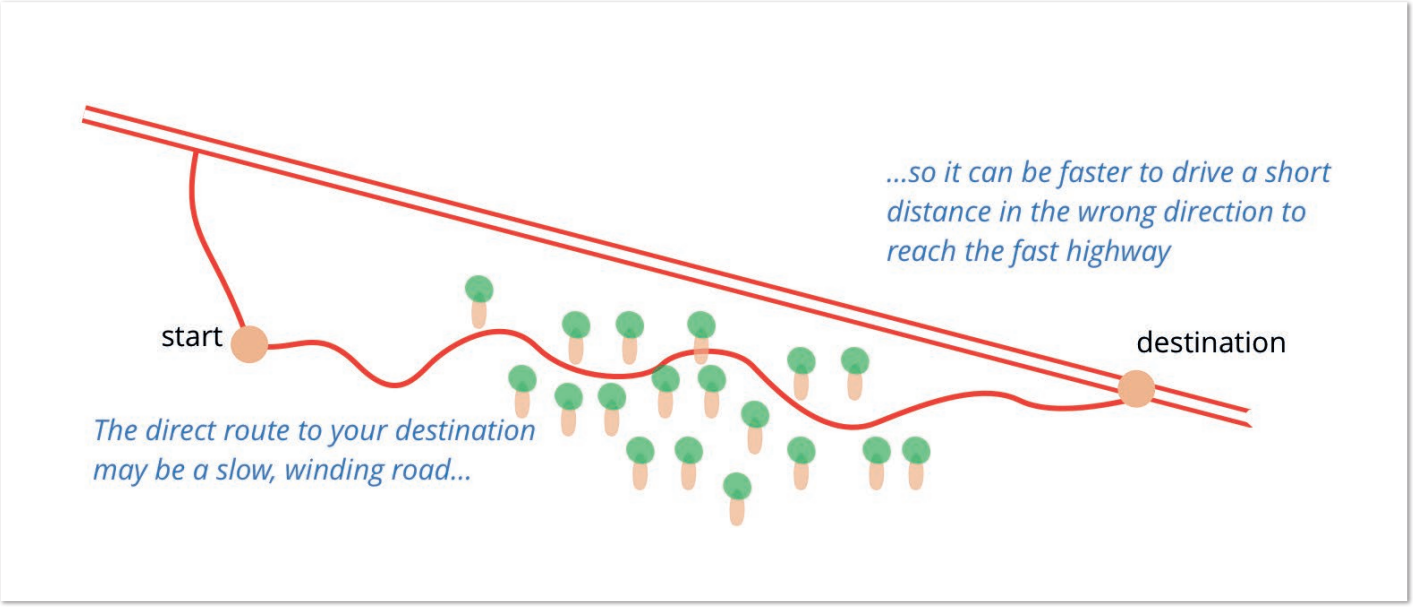


Abbildung 1: Preparatory Refactoring (© Jessica Kerr [6])

beitet habe. Nach dem Hinzufügen des neuen Konstruktors lasse ich meine Tests laufen und sehe, dass noch alles funktioniert. Ich habe durch das neue Array nichts kaputtgemacht.

Im nächsten Schritt kann ich jetzt die Tests ändern, damit sie den neuen Konstruktor benutzen *(siehe Listing 3* und *Listing 4)*. Jetzt verifiziert mein Produktionscode, dass sich die Semantik meiner Tests nicht geändert hat.

Protest?

„Aber das macht doch alles nur noch schlimmer!" – Ja, es ist noch nicht schöner geworden, aber wenn ich auf diese Weise arbeite, kann ich `Item` schrittweise aus `GildedRose` entfernen, ohne jemals zu viel auf einmal machen zu müssen und dabei nicht lieferfähig zu sein. Das ist der Weg, der zurückzuführen scheint, der uns aber zum Highway bringt. Die Alternative wäre, lange nicht lieferfähig zu sein, weil das Refactoring nicht fertig ist.

Auf dem Highway zum Ziel

Jetzt, wo wir `Item` und `GildedItem` an Ort und Stelle haben, können wir Funktionalität von `GildedRose` nach `GildedItem` verlagern. Dabei benutze ich einen leicht modifizierten Ansatz, der immer wieder bei vielen Refactorings funktioniert hat, wenn ich mit dem Smell Anemic Domain Model konfrontiert bin:

Normalerweise würde mein Ansatz so aussehen:

1. Extrahiere eine Methode, die die anämische Instanz als Parameter annimmt
2. Manipuliere das anämische Modell in der Methode
3. Benutze das *Move Method* Refactoring, um die Methode in die Klasse des anämischen Modells zu verschieben

Das anämische Modell wird durch die neue Methode weniger anämisch und bekommt eigene Funktionalität, die vorher außerhalb des Modells verstreut im Code lag.

```
private GildedRose createApp(Item item) {
  Item[] items = new Item[] { item };
  return new GildedRose(items);
}
```

Listing 3: Tests mit altem Konstruktor

```
private GildedRose createApp(Item item) {
GildedItem[] items = new GildedItem[] {
  new GildedItem(item)};
return new GildedRose(items);
}
```

Listing 4: Tests mit neuem Konstruktor

In unserem Beispiel dürfen wir `Item` aber nicht verändern. Darum kann ich nicht direkt Methoden dorthin verschieben und mich auch nicht auf die vollautomatische Version von *Move Method* verlassen. Stattdessen wende ich die händische Version des Refactoring an:

1. Erstelle eine Methode in der **neuen** Model-Klasse.
2. Kopiere den manipulierenden Code, der auf `Item` arbeitet, in die neue Methode (noch nicht kompiliert).
3. Ändere den Code so, dass er stattdessen das gekapselte Item manipuliert (kompiliert).
4. Ersetze den Aufruf des alten Codes durch den neuen.

In der Gilded Rose:

1. Code, den wir verschieben wollen *(siehe Listing 5)*
2. und 3. Methode in `GildedItem` anlegen *(siehe Listing 6)*
4. Aufrufer ändern *(siehe Listing 7)*

Jetzt lassen wir zur Sicherheit nochmal die Tests laufen und sehen, dass immer noch alles funktioniert. Mit `quality` machen wir analog weiter *(Listing 8)*.

```
if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
    items[i].sellIn = items[i].sellIn - 1;
}
```

Listing 5: Zu verschiebender Code im Aufrufer

```
public class GildedItem {

    private final Item item;

    public GildedItem(Item item) {
        this.item = item;
    }

    ...

    public void decreaseSellIn() {
        //nach 2.: items[i].sellIn = items[i].sellIn - 1;
        item.sellIn = item.sellIn - 1;

        //später: refactor to item.sellIn--;
    }
```

Listing 6: Verschobene Methode in GildedItem

```
if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
    gildedItems[i].decreaseSellIn();
}
```

Listing 7: Aufrufer danach

```
//vorher:
items[i].quality = items[i].quality + 1;
//nachher:
gildedItems[i].increaseQuality();


//vorher
items[i].quality = items[i].quality - 1;
//nachher
gildedItems[i].decreaseQuality();
```

Listing 8: Refactoring von Quality

Immer noch auf dem Weg

Für unsere `Item`-Klasse gibt es natürlich noch mehr zu tun als das, was ich in diesem Artikel zeigen konnte. `GildedItem` könnte ein paar Prädikatsmethoden gut vertragen, mit denen wir die Conditionals ersetzen. Die Methoden `increaseQuality()` und `decreaseQuality()` könnten ihre Invarianten besser forcieren und etwa verhindern, dass `quality` jemals negativ oder größer als 50 wird. Wenn wir mit all dem fertig sind, können wir das Array `Items[]` aus der Klasse `GildedRose` entfernen und `GildedItem` ist der einzige Ort, wo `Item` noch verwendet wird und bleiben kann. Unser Code ist trotz des Umbaus jederzeit lieferbar, weil wir uns für paralleles Refactoring entschieden haben.

Mit der vorgestellten Arbeitsweise sind Refactorings kein Blocker für den Rest der Arbeit mehr. Man kann sie auch während der Arbeit anhalten, etwas Wichtigeres einschieben und später fortsetzen. Zusammen mit den Tests, die wir als Sicherheitsnetz hinzugefügt haben, gewinnen wir an Sicherheit und Flexibilität und reduzieren das Risiko für unser Projekt.

In der nächsten Ausgabe der Java aktuell finden Sie Teil 2 dieses Artikels.

Quellen

[1] Emily Bache (2011): Gilded Rose Kata.
https://github.com/emilybache/GildedRose-Refactoring-Kata

[2] C2 Wiki: Feature Envy Smell
http://wiki.c2.com/?FeatureEnvySmell

[3] Martin Fowler (2003): Anemic Domain Model
https://martinfowler.com/bliki/AnemicDomainModel.html

[4] Michael C. Feathers (2016): Characterization Tests
https://michaelfeathers.silvrback.com/characterization-testing

[5] Michael C. Feathers (2004): Working Effectively with Legact Code. Prentice Hall, Upper Saddle River, NJ, United States

[6] Jessica Kerr (2015): Preparatory Refactoring
https://martinfowler.com/articles/preparatory-refactoring-example.html

Georg Berky
Valtech Mobility
georg.berky@valtch-mobility.com

Georg entwickelt hauptberuflich Connected-Car-Dienste. Handwerk und Leidenschaft ist für ihn die Programmierung, meistens in JVM-Sprachen wie Java, Groovy oder Clojure. Zum Handwerk gehören für ihn auch Themen wie die Pflege von Legacy Code, Automatisierung von Builds und Deployments oder Agilität im Team. Seit einigen Jahren ist er Co-Organisator der Software-Craftsmanship-Communities im Ruhrgebiet und in Düsseldorf. Wenn er mal nicht programmiert, spielt er Trompete oder praktiziert Aikido. Er twittert als @georgberky.

Das QA-Navigation-Board – Wie, wir müssen das noch testen?

Kay Grebenstein, Carl Zeiss Digital Innovation

In diesem Artikel wird das QA-Navigation-Board vorgestellt. Mit dem QA-Navigation-Board haben agile Entwicklungsteams ein visuelles Hilfsmittel, mit dem sie frühzeitig die planerischen Aspekte der Qualitätssicherung beurteilen können. Darüber hinaus kann das QA-Navigation-Board innerhalb der Projektlaufzeit auch als Referenz des aktuellen Vorgehens und als Ansatz für potenzielle Verbesserungen genutzt werden.

Wenn ein Projekt beginnt, erscheinen bei der Planung die Aspekte der Qualitätssicherung meist sehr spät auf der Agenda. Dies macht sich in späteren Projektphasen negativ bemerkbar. So treten währenddessen neue Hürden für das Team auf, beispielsweise die Integration vergessener Testarten, neuer Werkzeuge oder Testumgebungen.

In den klassischen Projekten gibt es bereits ein geeignetes Vorgehen, um alle Aspekte der Qualitätssicherung im Auge zu behalten: ein detailliertes Testkonzept, das die Testziele dokumentiert sowie entsprechende Maßnahmen und eine Zeitplanung festlegt *(siehe Abbildung 1)*. So ein Testkonzept umfasst meist 50, 100 oder mehr Seiten.

Abbildung 1: Aspekte der Teststrategie/Inhalte eines Testkonzepts (© ZEISS Digital Innovation)

Diese Detailtiefe eignet sich aber nicht für agile Projekte und Entwicklungsteams. Trotzdem sollte sich das Team vor dem Start eines agilen Projektes über die meisten Aspekte, die im Testkonzept genannt werden, Gedanken machen. Darum haben wir ein Hilfsmittel entwickelt, das es den Teams ermöglicht, alle Maßnahmen für eine optimale Testbarkeit in Softwareprojekten mit einzubeziehen. Das Hilfsmittel berücksichtigt sowohl die Frage „Was ist zu testen?" als auch „Wie und wo wollen wir testen?".

Der QA-Oktant

Um die erste Frage „Was ist zu testen?" im Hinblick auf Softwareprodukte zu beantworten, ist die Ausprägung der Qualitätskriterien der umzusetzenden Anforderungen ausschlaggebend. Die unterschiedlichen Qualitätskriterien sind in der ISO 25010 „Qualitätskriterien und Bewertung von System und Softwareprodukten (SQuaRE)" enthalten *(siehe Abbildung 2)*.

Je nachdem, wie stark sich die umgesetzten Anforderungen auf die Qualitätskriterien auswirken, ergibt sich die Notwendigkeit, diese mit einer entsprechenden Testart zu prüfen. So verlangen Apps mit hohem Datendurchsatz nach Effizienztests, Webshops sollten auf Kompatibilität in verschiedenen Browsern geprüft werden.

Um den Teams den Einstieg in das Thema und damit die Bestimmung der notwendigen Testarten zu erleichtern, nutzen wir den QA-Oktanten. Folgend der ISO 25010 beinhaltet der QA-Oktant die Qualitätskriterien für Softwaresysteme wie Funktionalität, Benutzbarkeit, Effizienz und so weiter. Sie geben einen Hinweis auf die notwendigen Testarten, die sich aus der gesetzten Gewichtung der unterschiedlichen funktionalen und nichtfunktionalen Kriterien ergeben *(siehe Abbildung 3)*.

Durch die einfache Visualisierung und Gewichtung der unterschiedlichen Qualitätskriterien kann der QA-Oktant für die Bestimmung der notwendigen Testarten innerhalb eines Workshops genutzt werden. Dazu stellt das Team dem Product Owner oder dem Fachbereich die Frage: „Wie wichtig sind die einzelnen Qualitätskriterien?" Ziel der Fragerunde ist es, eine Abstufung in der Gewichtung der verschiedenen Kriterien erkennbar zu machen.

Dabei werden die meisten Befragten keine große Abstufung zwischen den Qualitätskriterien machen oder besser gesagt wird die Antwort lauten: „Alles ist wichtig!". Das Team beziehungsweise der Moderator des Meetings haben nun die Aufgabe, die Fragestellung dahin zu konkretisieren, dass eine Differenzierung erfolgen kann. Dazu können verschiedene Fragetechniken zum Einsatz kommen. So kann eine Differenzierung nach Abgrenzung über das Einsatzgebiet passieren: Erfolgt der Einsatz einer fachlichen Anwendung auf HTML-Basis in einem Firmennetz und gibt es laut IT-Compliance nur einen Browser und nur eine Betriebssystemversion, dann können der Aspekt Kompatibilität und die damit verbundenen Tests niedriger eingestuft werden. Gibt es dagegen eine hohe Anzahl an Kombinationen von Plattformen, müssen umfangreiche Tests eingeplant werden. Für weitere Differenzierung kann zum Beispiel auch eine Negativfragetechnik eingesetzt werden: „Was passiert, wenn zum Beispiel die Benutzbarkeit herabgesetzt wird?". Für eine Anwendung zur monatlichen Rechnungsstellung wird angenommen, dass die negative Auswirkung der Benutzbarkeit die Geschwindigkeit der Erstellung einer Rechnung von zwei auf vier Stunden erhöht. Da die Anwendung nur einmal im Monat benutzt wird, wäre diese „Verzögerung" vertretbar und die Benutzbarkeit kann im QA-Oktanten niedriger eingestuft werden.

Diese Fragetechnik kann mithilfe von Priorisierung durch Risikoabschätzung erweitert werden: „Was passiert beziehungsweise welche Auswirkungen ergeben sich, wenn beispielsweise das Kriterium Sicherheit verringert wird?". Die Antworten ergeben sich hier aus folgenden Aspekten:

- Welche monetäre Auswirkung hätte ein Ausfall der Anwendung, wenn der Fokus auf das Kriterium verringert wird?

Abbildung 2: Qualitätskriterien nach ISO 25010 (© ZEISS Digital Innovation)

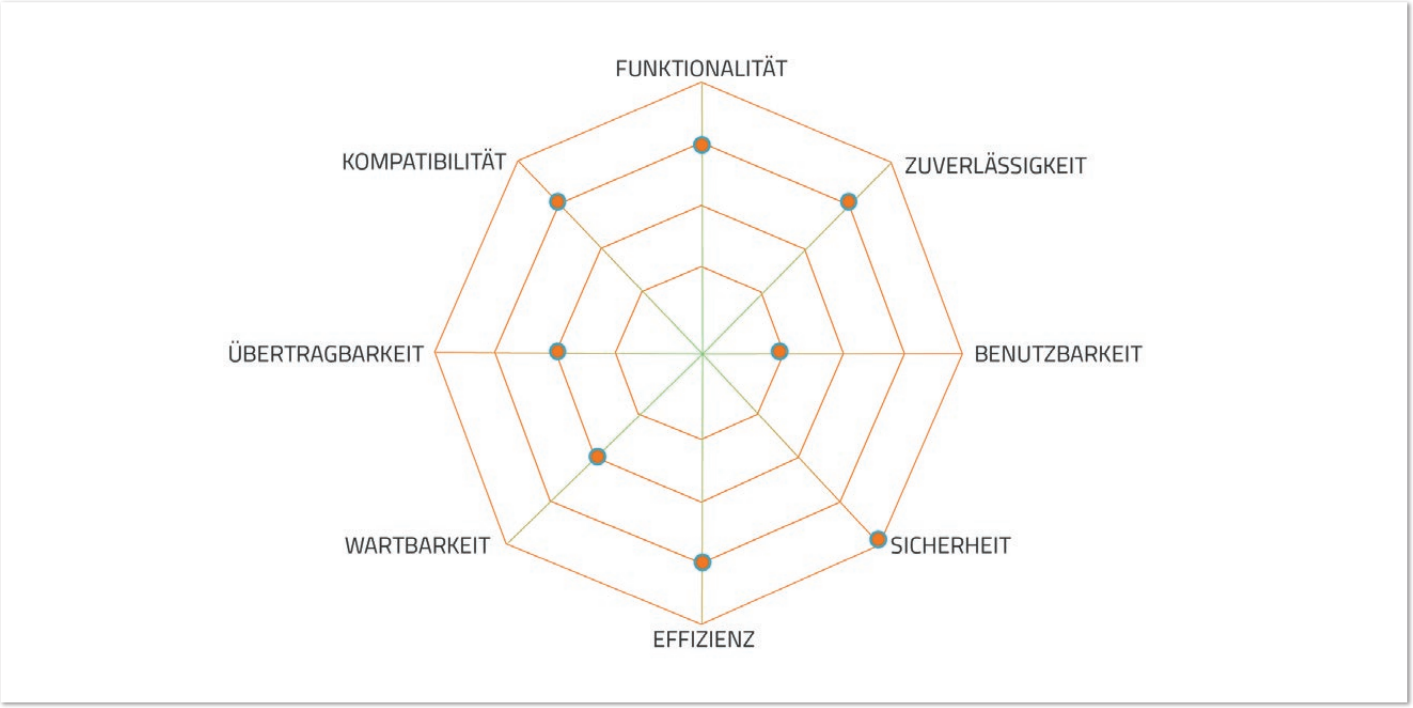


Abbildung 3: Der QA-Oktant mit gewichteten Qualitätskriterien (© ZEISS Digital Innovation)

- Wie viele Anwender wären bei dem Ausfall der Anwendung betroffen, wenn der Fokus auf das Kriterium verringert wird?
- Wären bei dem Ausfall der Anwendung Leib und Leben gefährdet, wenn der Fokus auf das Kriterium verringert wird?
- Würde die Firma beim Ausfall der Anwendung Reputation verlieren, wenn der Fokus auf das Kriterium verringert wird?

Liegen Ergebnisse und Erkenntnisse bei einem oder mehreren Qualitätskriterien vor, kann man diese zusätzlich mit den offenen Qualitätskriterien vergleichen und wie beim Komplexitätsvergleich im Planning oder der Estimation agieren. Mit der richtigen Fragestellung ergibt sich eine Übersicht über die Qualitätskriterien. Durch die einfache Visualisierung und Gewichtung der unterschiedlichen Qualitätskriterien kann der QA-Oktant für die Planung der Testarten genutzt werden. Dabei ist nicht immer nur das Ergebnis der wichtigste Aspekt des QA-Oktanten, sondern auch „der Weg ist das Ziel". Durch die Gewichtung im Team und gemeinsam mit dem Product Owner und/oder dem Fachbereich können unterschiedliche Meinungen besser wahrgenommen werden und ein besseres Verständnis aller Beteiligten entsteht. Am Ende lassen sich aus dem Ergebnis die notwendigen Testarten ableiten.

Die QA-Karte

Um die zweite Frage „Wie und wo wollen wir testen?" zu beantworten, müsste das Team den gesamten Entwicklungsprozess nach Test- und QA-Aspekten durchforsten und diese dokumentieren. Je nach Projekt kann der Entwicklungsprozess unterschiedlich ausgeprägt sein und damit die Fragestellung schnell recht komplex werden lassen *(siehe Abbildung 4)*.

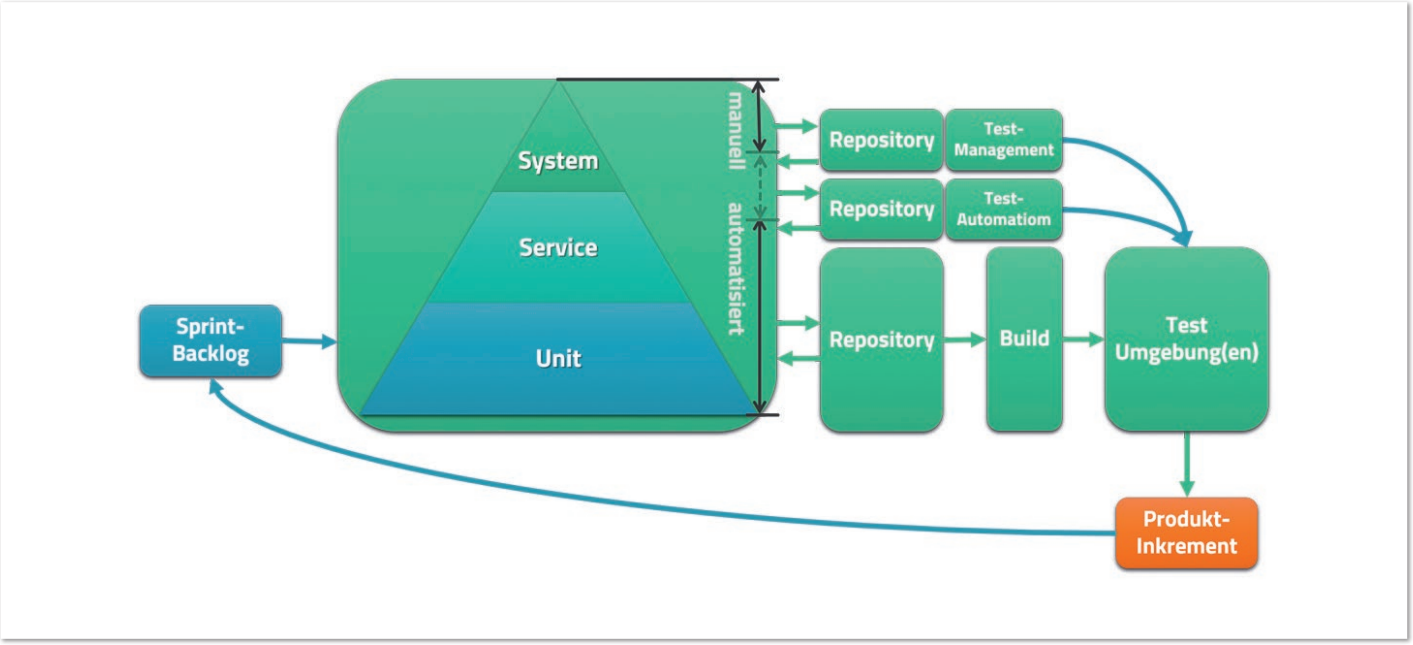


Abbildung 4: Entwicklungs- und QA-Prozess (© ZEISS Digital Innovation)

Schritt	Beschreibung
Vorbereitung	Es sollten alle Beteiligten des agilen Projektes eingeladen werden: ▪ Entwicklerteam (Entwickler, Tester) ▪ Scrum Master ▪ Product Owner ▪ weitere Stake- und Shareholder Das QA-Navigation-Board wird an einer (Pinn-)Wand befestigt. Zusätzlich wird der QA-Oktant als Arbeitsblatt für jeden Teilnehmer ausgedruckt.
Vorstellung	Vorstellung des QA-Navigation-Board und der Ziele des Workshops durch den Moderator (agiler QA-Coach) sowie Vorstellung der Beteiligten.
Oktant I	Kurze Vorstellung des QA-Oktanten und der Qualitätskriterien. Ziel muss sein, dass alle Beteiligten das Arbeitsblatt ausfüllen können und alle die Qualitätskriterien verstanden haben beziehungsweise nicht anschließend aneinander vorbeireden. Des Weiteren stimmen sich die Teilnehmer über die Dimensionen des QA-Oktanten ab: Wie sollen die Abstände des Diagramms bewertet werden (1, 2, 3 oder S, M, L, XL und so weiter). Danach werden die Arbeitsblätter ausgeteilt, innerhalb von fünf bis zehn Minuten von jedem Beteiligten allein ausgefüllt und mit seinem Namen versehen.
Oktant II	Nach Ablauf der Zeit sammelt der Moderator die Arbeitsblätter wieder ein und platziert sie an einer (Pinn-)Wand. Die Aufgabe des Moderators ist es nun, die einzelnen Qualitätskriterien durchzugehen. Dazu identifiziert er pro Kriterium den gemeinsamen Nenner (Durchschnitt) und bespricht die größten Abweichungen mit den dazugehörigen Personen. Wurde im Team ein Konsens über den Wert des Kriteriums erzielt, wird dieser Wert vom Moderator dokumentiert.
Testarten	Auf Basis der Wertung der Qualitätskriterien leiten nun die Beteiligten die notwendigen Testarten ab. Je wichtiger ein Qualitätskriterien bewertet wurde, desto wahrscheinlicher muss es durch ein passendes Testvorgehen geprüft werden. Die identifizierten Testarten werden vom Team in der Testpyramide des QA-Navigation-Board platziert.
Karte	Sind alle Testarten identifiziert und platziert, können die notwendigen Testressourcen und anderen Testartefakte auf dem QA-Navigation-Board platziert werden. Hierbei hilft die Checkliste.
Abschluss	Hat das Team das Navigation Board weitestgehend ausgefühlt, wird er an einem passenden Platz im Teamraum aufgehängt. Der Moderator schließt den Workshop ab und verweist darauf, dass das QA-Navigation-Board vom Team weiterentwickelt und auch in den Retrospektiven genutzt werden kann.

Tabelle 1: Ablauf für den Workshop

Um auch hier den Teams einen mühelosen Einstieg in das Thema zu geben, haben wir die QA-Karte entwickelt. Die QA-Karte bietet dem Team eine praktische Möglichkeit, die Maßnahmen für eine optimale Testbarkeit der Projekte zu planen und zu dokumentieren. Ziel ist es, bereits zu einem frühen Zeitpunkt alle QA-relevanten Fragen für die Teams und Entwicklungsprojekte durch einen spielerischen Ansatz zu ermitteln.

Nach der Definition der Testschwerpunkte durch die Nutzung des QA-Oktanten und der Bestimmung der notwendigen Testarten können alle Aspekte der Teststrategie wie Testarten, Ressourcen und Werkzeuge visualisiert, diskutiert und priorisiert werden. Als Good Practice zeigte sich, bei der Steuerung der Befüllung der QA-Karte zwei Hilfsmittel zu nutzen. Zum einen eine Checkliste, die die passenden Fragen beinhaltet, um die verschiedenen Teile der QA-Karte zu befüllen:

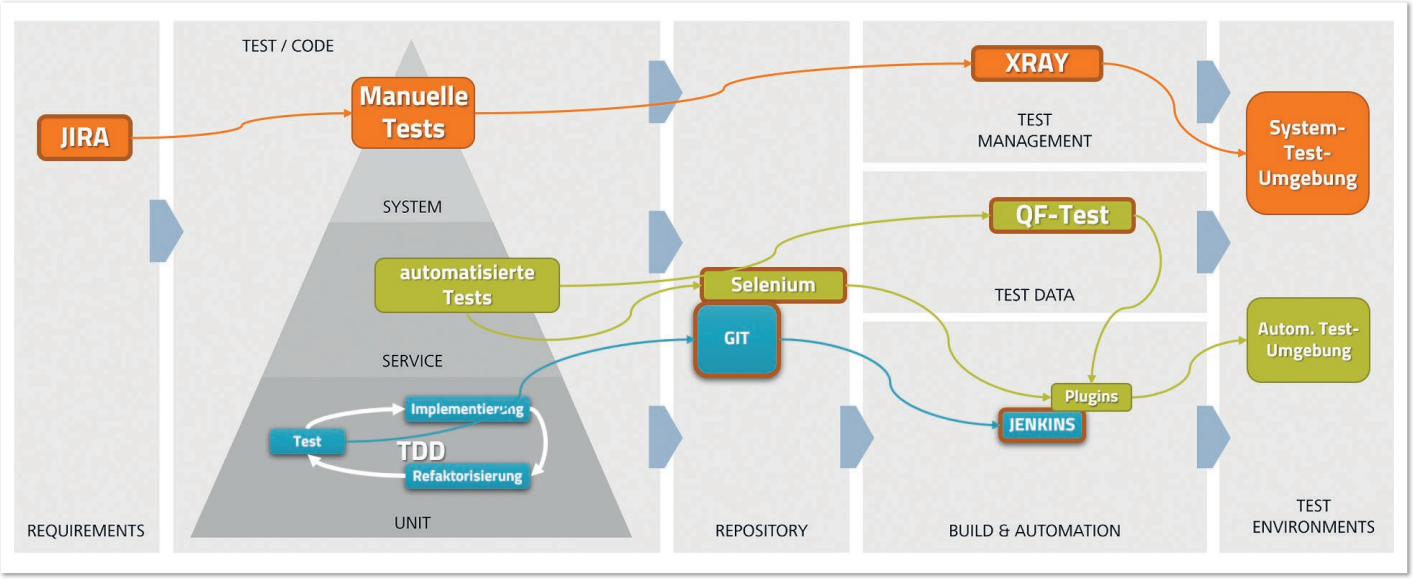


Abbildung 5: Ausgefüllte QA-Karte (© ZEISS Digital Innovation)

- Feld: Requirements
  - Wo liegen die Anforderungen?
  - Unterstützen die Anforderungen die Testfallerstellung?
  - Lassen sich Anforderungen und Tests verknüpfen?
- Feld: Test/Code
  - Wo werden die Tests platziert?
  - Haben wir die nötigen Skills?
- Feld: Repository
  - Wo werden die Testartefakte gespeichert?
  - Gibt es unterschiedliche Artefakte?
- Feld: Test Management
  - Wie planen wir unsere Tests?
  - Wir dokumentieren wir unsere Tests?
  - Wie informieren wir? Und wen?
- Feld: Automation
  - Wie viel Testautomatisierung?
  - Benötigen wir weitere Werkzeuge?
  - Benötigen wir Testdaten?
- Feld: Build
  - Wie oft wollen wir bauen und testen?
  - Wie wollen wir die QA integrieren?
  - Wollen wir Wartbarkeit prüfen?
- Feld: Test Environments
  - Haben wir für jeden Test die passende Umgebung?
  - Kommen wir uns in die Quere?

Das zweite Good Practice ist ein guter Moderator, der den Workshop in die richtige Richtung leitet. Dabei sollte der Workshop nicht länger als eineinhalb Stunden dauern und den in *Tabelle 1* beschriebenen Ablauf haben.

Zusammengesetzt ergeben der Oktant und die Karte das QA-Navigation-Board, der sich als Wandbild platzieren lässt. Mit dem ausgefüllten QA-Schlachtplan *(siehe Abbildung 5)* haben die Entwicklungsteams ein visuelles Hilfsmittel, mit dem sie frühzeitig die planerischen Aspekte der Qualitätssicherung beurteilen können. Dabei kann das QA-Navigation-Board innerhalb der Projektlaufzeit auch als Referenz des aktuellen Vorgehens und als Ansatz für potenzielle Verbesserung genutzt werden.

Gerne senden wir Ihnen unser QA-Navigation-Board auch in Originalgröße (700 x 1000 mm) zu. Schreiben Sie uns dazu einfach eine E-Mail mit Ihrer Adresse an *kay.grebenstein@zeiss.com*.

Kay Grebenstein
Carl Zeiss Digital Innovation
kay.grebenstein@zeiss.com

Kay Grebenstein arbeitet als Tester und agiler QA-Coach für die Carl Zeiss Digital Innovation AG in Dresden. Er hat in den letzten Jahren in Projekten unterschiedlicher fachlicher Domänen (Telekommunikation, Industrie, Versandhandel, Energie) Qualität gesichert und Software getestet.

NoSQL-Injections und wie sie verhindert werden können

Matthias Altmann, Micromata GmbH

Mit der zunehmenden Beliebtheit von NoSQL-Datenbanken wächst auch die Gefahr, dass sie zum Ziel von Cyberattacken werden. Dieser Artikel beschreibt zunächst die Gefahrenlage, um dann praktische Tipps zur Abwehr von NoSQL-Injections zu geben.

Die Speicherung von Daten erfolgte bisher in der Regel über die Structured Query Language, kurz SQL. Heute hat sich darüber hinaus mit den sogenannten NoSQL-Datenbanken („Not only SQL") ein alternativer Ansatz etabliert, der zunehmend an Bedeutung gewinnt. Die Website db-engines.com listet 25 Prozent aller derzeit genutzten DB-Systeme als NoSQL-Datenbanken, Tendenz steigend.

Im Falle der klassischen SQL haben wir es traditionell mit eher festen Strukturen zu tun, die angesichts der zunehmenden Digitalisierung nicht ohne Weiteres mitskalieren. Zum einen, weil das Datenvolumen weltweit rapide steigt, zum anderen, weil diese Daten als logische Folge einer immer intensiveren Nutzung digitaler Technologien vielfältiger und diverser werden: Kundendaten oder Social-Media-Daten, Daten aus komplexen Lieferketten oder Webtraffic-Daten, Sensordaten aus Smartphones, Kameras und Automobilen. Jeden Tag bewegen wir einen schier unendlichen Datenstrom – mit dem auch unsere Datenbanken umgehen müssen.

Ein probates Mittel, der Lage Herr zu werden, sind NoSQL-Datenbanken: Sie kommen deshalb gut mit der dynamischen Datenentwicklung zurecht, weil sie wesentlich besser mit schnell veränderbaren Daten umgehen können.

NoSQL – Paradigmenwechsel für die Softwareentwicklung

Dazu brechen sie mit den Paradigmen von relationalen Datenbanken und mit SQL. So erlaubt NoSQL beispielsweise deutlich flexiblere, offenere Datenstrukturen und -abfragen, ist hochskalierbar und erfordert nicht zwingend, dass Daten sofort konsistent zur Verfügung stehen müssen. Das hat nicht nur zur Folge, dass Änderungen in NoSQL-Datenbanken viel schneller und einfacher durchgeführt werden können – sondern auch, dass mehr Code aus der Datenbank in die Programmlogik wandert.

Vieles, was früher der Datenbankadministrator beziehungsweise die Datenbank selbst durchgeführt hat, wandert nun als Aufgabe zum Softwareentwickler.

NoSQL als Herausforderung für die Datensicherheit

Allerdings sind Programmiersprachen und Frameworks oft nicht ausreichend gehärtet, um das Entstehen von Sicherheitslücken gänzlich zu verhindern. Deshalb sollten Softwareentwickler unbedingt wissen, was sie je nach NoSQL-System, um die es im weiteren Verlauf gehen soll, beachten müssen.

Die Agentur für Cybersicherheit gibt jedes Jahr einen Report zur aktuellen Sicherheitslage *[1]* im Netz heraus. Der letzte Stand von Januar 2020 beschreibt noch das Jahr 2018 und zeigt, dass der Angriff auf SQL-Datenbanken mit Abstand der häufigste ist. Da NoSQL-Datenbanken im Wachstum sind, kann sich dieser anhaltende Trend schnell auch auf diesen Datenbanktyp ausweiten. Datenbanken vor unbefugtem Zugriff zu schützen sollte also oberstes Gebot einer professionellen IT-Security sein.

NoSQL-Injections wirksam abwehren

Die Gefahren auf dem Gebiet von NoSQL sind bislang noch kaum untersucht, ausgereifte Tools in dem Bereich fehlen auch hier. Anders als bei SQL gibt es hier auch keinen Standard, an dem sich die einzelnen Abfragetypen orientieren. Dadurch können auf unterschiedliche Weisen Lücken aufgemacht werden, deren Tragweite (Stand heute) noch schwer abzuschätzen ist. Was wir wissen, ist: Das Einschleusen von Späh- oder Schadcode gelingt vor allem dort, wo Eingaben nicht korrekt überprüft werden oder wo Daten durch sogenannte String-Konkatenation mit Code vermischt werden.

Deshalb ist es wichtig, Daten und Code sortenrein voneinander zu trennen. Denn wenn wir keine Ordnung zwischen beiden herstellen, sind auch bei NoSQL prinzipiell ähnliche Attacken möglich wie auf relationale Datenbanken. Diese sogenannten NoSQL-Injections sind verheerend, weil sie die Daten in der Datenbank nicht nur auslesen, sondern auch verändern können. Schlimmer noch: Im Unterschied zu relationalen Datenbanken befindet sich der Angreifer hier oft nicht nur auf der Datenbank selbst, sondern hat auch Zugriff auf den Anwendungscode. Das bedeutet, dass er im Ernstfall die komplette Anwendung umbauen kann.

Sanitizer und Validations für mehr Datenhygiene

Es folgen zwei Beispiele mit MongoDB, um zu veranschaulichen, wie solche Angriffe aussehen können – inklusive Coding-Tipps, wie man das verhindern kann.

Beispielangriff auf Document Store von MongoDB

Der Document Store von MongoDB ist derzeit das beliebteste NoSQL-System *[2]*. Nehmen wir beispielhaft an, Request-Daten wandern hier direkt in eine Abfrage – im in *Listing 1* gezeigten Beispiel mit PHP der Einfachheit halber als GET. Dann sieht eine Standardabfrage zum Beispiel so aus: `http://mydomain.com?user=matthias`.

Problematisch sind nun die Operatoren: Mittels `http://mydomain.com?user[$ne]=matthias` können nun alle anderen Nutzer abgefragt werden. Dadurch wird daraus die in *Listing 2* gezeigte Anfrage. `$ne` ist der Operator für „nicht gleich". Alles, was nicht gleich „matthias" ist, wird für `user` ebenso herangezogen. Ähnliches gilt für `http://mydomain.com?user[$gt]=""`. Hier sind alle Strings größer als der leere String.

Sieht der Code zum Abfragen eines Nutzerzugangs wie in *Listing 3* aus, so können mit `http://mydomain.com?user[$ne]=xyz&pass[$ne]=p` alle Zugänge abgefragt werden, die nicht dem Zugang mit Nutzer „xyz" und nicht dem Passwort „p" entsprechen.

Verwendet der Softwareentwickler in diesem Szenario nun die Menge der Passwort-Rückmeldungen für das Login, hat sich der Angreifer erfolgreich eingeloggt – und kann dies jetzt durch Austauschen des Nutzernamens beliebig vielen weiteren Nutzern ermöglichen.

Beispielangriff mit Java auf MongoDB

Ähnliches gilt auch in der Java-Welt. Die aktuelle Library 3.4 von MongoDB verwendet ein sogenanntes BasicDBObject für den Zugriff *[3]*. Wird stattdessen aber die Eingabe als JSON geparst, so sind dergleichen Angriffe möglich.

Als Beispiel wird hier der Nutzername in der Anmeldemaske in der Variablen `user` hineingereicht. Statt also *Listing 4* auszuführen, geschieht das in *Listing 5* gezeigte Szenario. Dies führt im Ergebnis dazu, dass die Eingabe `matthias', name:{$ne:'matthias'}, password:'mypass` in der Folge alle anderen Nutzer ausgibt, wenn `matthias/mypass` die Credentials sind.

```
$query = array(“user” = > $_GET[‘user’]);
$result = $collection.find($query);
```

Listing 1

```
$query = array(”user” => array(“$ne”: matthias));
```

Listing 2

```
$query = array(“user” = > $_GET[‘user’], “pass” => $_GET[‘pass’]);
$result = $collection.find($query);
```

Listing 3

```
BasicDBObject dbQuery = new BasicDBObject("user", user);
DBCursor result = characters.find(dbQuery);
```

Listing 4

```
String stringQuery = "{ 'user' : '" + user + "'}";
DBObject query = (DBObject) JSON.parse(stringQuery);
DBCursor result = characters.find(query);
```

Listing 5

Validations und Sanitizer gegen NoSQL-Injections

Je nach Framework und Programmiersprache sind die Methoden gegen die Angriffe unterschiedlich, Folgendes sollten wir aber prinzipiell beherzigen.

Validation

Die erste Regel lautet: Eingaben validieren! Wir wissen ja zum Glück einiges über den Nutzer – zum Beispiel, dass er ein String sein muss, oder auch, dass eventuell nur bestimmte Zeichen oder Längen verwendet werden dürfen. Also prüfen wir erst mal genau das.

In PHP können wir dafür `is_string` verwenden – oder, sofern wir bestimmte Klassen wie E-Mails, URLs oder IPs identifizieren wollen, auch `filter_var`. Bei Java bieten sich Apache Commons hierfür an.

Sanitizing

Die zweite Regel lautet: Sanitizing! Im ersten Schritt haben wir geprüft, ob die Bedingungen, die wir bei der Eingabe erwarten, tatsächlich stimmen. In diesem zweiten Schritt geht es darum, alles rauszuwerfen, was nicht reinpasst.

- In PHP kann diese Eingaben-Desinfektion ebenfalls mit `filter_var` erledigt werden. Hierzu bietet die Sprache verschiedene Filterregeln an, die man mitgeben kann, zum Beispiel `FILTER_SANITIZE_STRING`.
- In JavaScript kann für diesen Zweck die Library `mongo-sanitize` [4] eingesetzt werden.
- In Java empfiehlt sich dafür meistens Apache Commons – hier zum Beispiel `normalize`- und `strip`-Methoden anwenden.

In anderen Programmiersprachen und Frameworks können diese beiden Maßnahmen anders aussehen oder bereits durch das Framework oder die Sprache erledigt werden. Im Zweifelsfall sollten Softwareentwickler dies nachschlagen.

Tools zur Behebung von NoSQL-Lücken

Um NoSQL-Injections zu verhindern, kommt auch dem Pentesting eine entscheidende Rolle zu.

Ausgereifte Tools zum Aufspüren von NoSQL-Lücken müssen allerdings noch entwickelt werden. Versuche wie NoSQLMap *[5]* haben zurzeit noch großes Potenzial nach oben, insbesondere im Bereich der Webschnittstelle. SQLMap kann bisher gar kein NoSQL integrieren *[6]*. Bis es so weit ist, bleibt auch hier nur das manuelle Testen.

Zusammenfassung

In Zeiten wachsender Popularität von NoSQL-Datenbanken und der noch immer großen Beliebtheit von Datenbankangriffen ist es wichtig, unsere IT-Systeme so weit wie möglich abzudichten. Hierfür ist bis auf Weiteres den Empfehlungen der jeweiligen Datenbank Folge zu leisten, sofern solche vorhanden sind.

Falls nicht, empfiehlt es sich, Eingaben zu validieren und zu säubern. Wer von uns will schließlich für den nächsten großen Daten-Leak verantwortlich sein, bei dem große Datenmengen illegal abgeschöpft werden konnten?

Quellen

[1] *https://www.enisa.europa.eu/publications/enisa-threat-landscape-report-2018*
[2] Stand 5.2.2020 *https://db-engines.com/en/ranking*
[3] *https://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBObject.html*
[4] *https://www.npmjs.com/package/mongo-sanitize*
[5] *https://github.com/codingo/NoSQLMap*
[6] *https://github.com/sqlmapproject/sqlmap/issues/1674*
[7] *https://www.meetup.com/de-DE/IT-Security-Kassel/*
[8] *https://www.heise-devsec.de/2019/lecture.php?id=8488&source*
[9] *https://www.micromata.de/referenzen/publikationen/*
[10] *https://secf00tprint.github.io/blog/about.html*

Matthias Altmann
Micromata GmbH
m.altmann@micromata.de

Matthias Altmann ist Softwareentwickler und IT-Security-Experte bei der Micromata GmbH, wo er gemeinsam mit seinen Kollegen den Bereich IT-Sicherheit betreut und fortentwickelt. Er ist außerdem Mitbegründer und Organisator des IT-Security-Meetups Kassel [7] und teilt sein Know-how darüber hinaus auf Fachkonferenzen [8], in Fachbeiträgen [9] und gelegentlich auch auf seinem Blog [10].

Spooky Magic! – Fehler verhindern, bevor sie entstehen?

Georg Haupt, oose innovative Informatik eG

Wie testen Sie? Die letzte Umfrage von softwaretest-umfrage.de im Jahr 2016 ergab, dass die meisten deutschen Unternehmen ihre Software mit nachgelagerten Methoden testen. Das bedeutet: Erst baut ein Entwickler lustige Fehler in den Code ein, dann haben die Tester etwas zu tun und Spaß beim Finden. Aber ist dieses Spiel, erst Fehler im Code einbauen, um sie anschließend per Qualitätssicherung zu finden, nicht unnütz teuer? Ja, ist es, trotzdem machen es doch alle so! Aber betrachten wir es mal aus einem anderen Winkel. Dieses Vorgehen ist in etwa so, als würden Sie jeden Tag zum Arzt gehen, um sicherzustellen, dass Sie gesund sind. Also würden Sie sich niemals die Hände waschen und sich auch nicht warm anziehen, wenn es draußen kalt ist. Warum auch, Sie werden ja jeden Tag untersucht. Oder leben Sie in einer agilen Familie? Das heißt, ein Familienmitglied ist praktischerweise Arzt? Dieser würde dann nach jeder ausgeübten Tätigkeit einen kompletten Körper-Regressionstest durchführen. Und wenn dann eine Abweichung von der Normalfunktion entdeckt würde, würden Sie direkt an Ort und Stelle behandelt werden. Klingt nicht wirklich realistisch! So ähnlich ist aber häufig die Testkultur. Ist es nicht sinnvoller, die Fehler schon im Entstehen zu verhindern? Ist denn „da draußen" nicht etwas, das uns hilft, Fehler zu verhindern? Quasi die dicken Pullis gegen die Bugs? Doch, denn es gibt auch Wasser und Seife sowie Schal und Mütze für den Entwicklungsprozess. Dieses alte Hausmittel nennt sich „konstruktive Qualitätssicherung" – das ist eben keine Zauberei.

Der wirtschaftliche Schaden durch Softwarefehler lagt 2014 in Deutschland bei etwa 84,4 Milliarden Euro. Für das Beheben und den Umgang mit den Fehlern werden zusätzliche 14,4 Milliarden Euro geschätzt *[1]*. In Summe also ein wirtschaftlicher Gesamtschaden von 98,8 Milliarden Euro. Die aktuell zu beziffernde Summe 2019/2020 liegt vermutlich nochmal deutlich höher. Einige Experten sprechen in diesem Zusammenhang von hohen dreistelligen Milliardensummen.

Wenn wir aber diese Kosten einmal näher betrachten, stellen wir fest, dass es eben nicht zwingend Bugs oder falscher Code sind. In vielen Fällen sind es die Entwicklungsprozesse und die vorherrschende Fehlerkultur, die diese schlechte Qualität zu verantworten haben. Fehler werden immer dort gehäuft gemacht, wo die Umstände sie begünstigen.

Der rumänisch-amerikanische Wirtschaftsingenieur Joseph M. Juran schreibt in seinem Buch „Qualität von Anfang an“: „Die Fehlerursachenerfassung sollte nicht mit dem Ziel aufgestellt oder betrieben werden, Schuldige zu finden, sondern auf eine Prozessverbesserung abzielen. Das Verhältnis zwischen Systemfehlern (beziehungsweise Prozessfehlern, also vom Management zu vertreten) zu Mitarbeiterfehlern liegt bei 85 zu 15.“ Bedeutet also: Die Fehler werden zwar von „normalen Mitarbeitern“ gemacht, aber die Umstände, also die Prozesse, werden vom Management verantwortet.

Daher ist es sinnvoll, Prozesse zu betrachten und Fehlermöglichkeiten weniger Raum zu bieten. Allerdings ist es wenig hilfreich, wenn diese Prozesse vom Management gesetzt werden. Lösungsfindung bei prozessualen Problemen sollten an dieser Stelle von den beteiligten Mitarbeitern durchgeführt werden. Dann werden die beschlossenen Änderungen und Maßnahmen auch zu den Bedürfnissen und Problemen der Beteiligten passen. Hinzu kommt die Faustregel, dass etwa 70 Prozent der Fehlerkosten in der Softwareentwicklung auf mangelhafte Anforderungen zurückzuführen sind. Auch hier sind die oben beschriebenen nachgelagerten Tests das falsche Mittel.

Wir können also zwei Problemfelder identifizieren:

1. Die qualitativen Folgen durch Prozessfehler sind zu hoch
2. Die Qualität der Anforderungen ist zu niedrig

Genau bei diesen beiden Problemen setzt die konstruktive Qualitätssicherung (QS) an. In der konstruktiven QS geht es primär darum, Fehlerprävention zu betreiben, Prozesse und Umfelder anzupassen, damit Fehler nicht mehr oder weniger passieren. Aus Fehlern lernen und sich zu verbessern, ist einer der Leitsätze.

Prozessuale Probleme identifizieren

Betrachten wir einmal das sicherste Verkehrsmittel der Welt: die private Linienluftfahrt. Trotz der enorm wachsenden Komplexität der Systeme, Zunahme des Verkehrs und des immer höheren Alters der Flotten, sinkt die Zahl der Zwischenfälle (Fehler) kontinuierlich. Einer der Gründe für diesen jahrelangen Trend ist der kontinuierliche Fehler-Lernprozess und die Verbesserung der Qualität. Dies ist unter anderem auf unabhängige und sorgfältige Untersuchungen von relevanten Vorfällen zurückzuführen.

In diesen Untersuchungen werden die Umstände eines Zwischenfalls minutiös betrachtet. Nicht aber, um etwaige Schuldige zu identifizieren. Der Fokus liegt vielmehr auf der Kausalität zwischen Ursache und Wirkung, der Herleitung von Fehlerwirkungen auf Basis der Fehlhandlung von Personen oder Systemen. Dieses Vorgehen in Verbindung mit dem Ziel, Aufklärung und Prozessverbesserung zu erzielen, ist das Erfolgsrezept in der Luftfahrt. Warum sollten Sie nicht in Ihrem Entwicklungsprozess die gleichen Mechanismen etablieren?

Nochmal, es geht nicht darum, einen Fehler in Ihrem Produkt zu finden – das wäre Bugfixing –, sondern die Ursachen für diese Fehler zu finden. Die meisten Fehler in Quellcodes sind auf ungünstige Umgebungsparameter zurückzuführen, die während des Zeitpunkts der Fehlhandlung herrschten. Normalerweise möch-

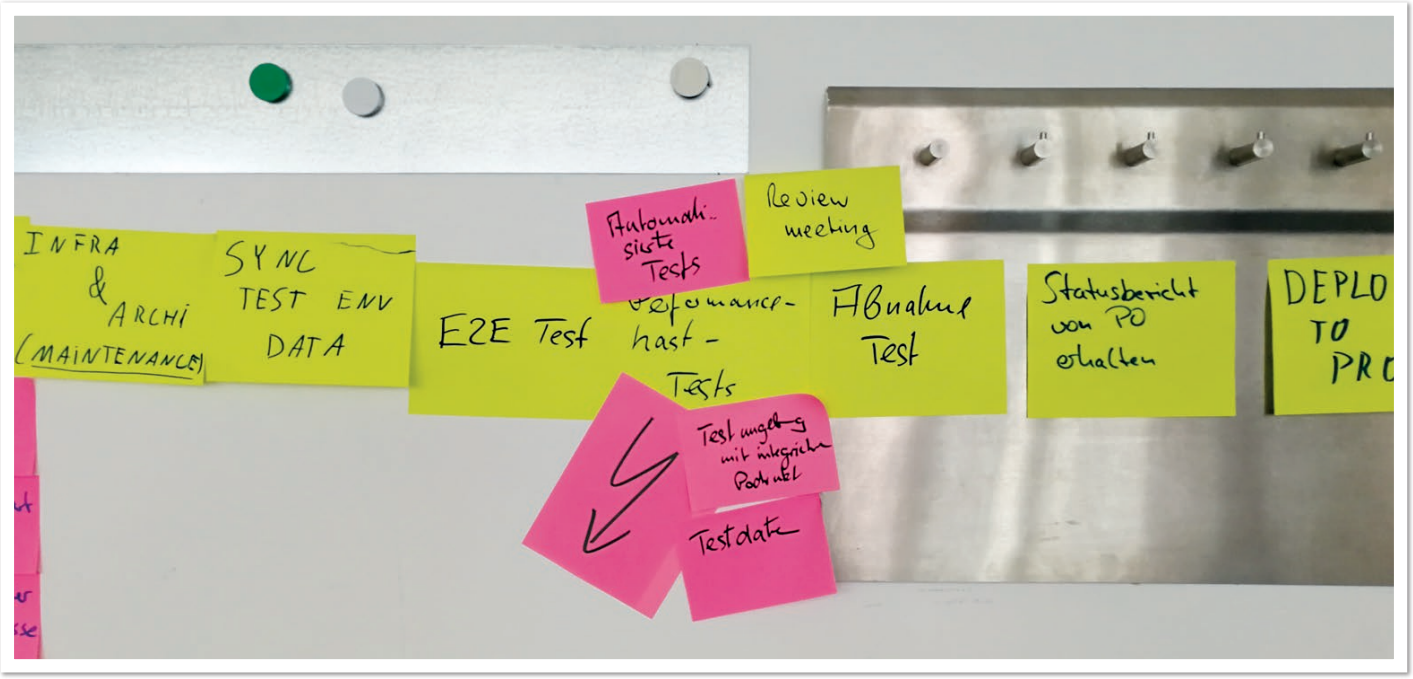


Abbildung 1: Quality Storming (© Haupt, QS-Barcamp)

ten Menschen keine Fehler machen. Sie sind in der Regel bemüht, die bestmögliche Arbeit abzuliefern. Trotzdem sind immer wieder Fehler im Code vorhanden. Wie kann das sein?

Also schauen wir uns an, welche Begleitumstände zu dem Zeitpunkt vorlagen, als der Fehler entstand. Dort suchen wir nach den möglichen Ursachen, um zukünftig diese Sorte Fehler zu vermeiden. Für eine solche Suche hilft es, einen Fehlerworkshop durchzuführen, um dann Maßnahmen daraus abzuleiten, die die Umgebungsparameter zukünftig positiv beeinflussen. Diese recht einfache Vorgehensweise kann einen Großteil der auftretenden Fehler schon im Vorfeld vermeiden.

Was beleuchten wir in einem solchen Workshop? Es geht darum, im Prozess die Stellen zu identifizieren, an denen Fehler gehäuft auftauchen. Jeder erfahrene Tester weiß, dass Fehler Rudeltiere sind. Sie treten selten allein auf. Die Methode des Quality Storming, eine Abwandlung des bekannten Event Storming, hat sich hier bewährt. An einem solchen Workshop sind alle am Entwicklungsprozess Beteiligten zusammen. Wichtig ist, dass die Beteiligten aus einer Hierarchieebene kommen, um den „Ober-sticht-unter-Effekt" zu vermeiden. Besonders die unterschiedlichen Sichtweisen vom Supportmitarbeiter, Anforderer, Entwickler bis zum Tester sind wichtig und hilfreich. Betrachtet wird der ganze Entwicklungsprozess – angefangen von der Idee einer Softwareänderung bis zur Auslieferung oder Nutzung durch den Kunden. Es werden von den Teilnehmern alle dazwischenliegenden Events auf Klebezetteln festgehalten und an die Wände geklebt *(siehe Abbildung 1)*. Da jeder Beteiligte die Events aus seiner eigenen Sicht in den Zeitstrahl einsortieren soll, decken sich hierbei schon während des Schreibens und Klebens Missverständnisse und Ungereimtheiten im Prozess auf.

Anschließend werden die markantesten Fehler des letzten Entwicklungszyklus als Beispielfehler untersucht. Achtung: Es geht nicht darum, dem Fehler ein Gesicht zu geben! Schuldigensuche ist tödlich für die Effizienz und Akzeptanz eines Fehlerworkshops. Zuerst betrachten wir den Entdeckungszeitpunkt – wann wurde der Fehler gemeldet oder entdeckt? Danach kümmern wir uns um den Zeitpunkt des Fehlers, also wann der Fehler „eingebaut" wurde.

Im Folgenden untersuchen die Prozessbeteiligten die Umstände zum Fehlhandlungszeitpunkt. Wir stellen uns immer die Frage: „Welche Gründe lagen vor, dass der Entwickler den Fehler im Code hinterlassen hat?" Es kann helfen, die Fehler zu klassifizieren. Die Fehlerklassen haben häufig ähnliche Begleitumstände. Häufige Gründe für Fehler sind:

- **Komplexität:** der Code ist so komplex, dass die Auswirkung durch die Änderung nicht absehbar war
- **Kompliziertheit:** die Menge der Änderung war so hoch, dass dabei Details unter den Tisch gefallen sind
- **Stress/Druck:** der Entwickler war gezwungen, unter Zeitdruck die Änderung einzupflegen
- **Kommunikation:** es wurde nicht richtig, unvollständig oder gar nicht kommuniziert
- **Ablenkung:** Störungen reißen den Entwickler aus der Konzentration
- **Wissensschwächen:** für die gestellte Aufgabe war das Wissen oder die Erfahrung nicht ausreichend
- **Missverständnis:** vermeintlich klare oder implizite Inhalte wurden unterschiedlich interpretiert
- **Gegeneinander statt miteinander:** das Team ist keine Einheit oder fühlt sich nicht für die Qualität verantwortlich
- **Motivation:** die Bedingungen untergraben die Motivation der Entwickler
- **Persönliche Gründe:** der Entwickler war infolge von persönlichen Problemen „nicht bei der Sache"

In vielen Fällen sind es auch Mischungen der Gründe, die dazu führen, dass im Code Fehler landen. Dies gilt es herauszufinden, ohne dabei gegenseitige Schuldzuweisungen oder Fingerpointing zu betreiben. Hier hilft es, einen erfahrenen Moderator oder „Quality-Evangelisten" zurate zu ziehen.

Zusätzlich wird untersucht, wie der Zeitraum zwischen Machen und Finden verkürzt werden kann. Die Frage lautet: „Was können wir tun, dass wir solche Fehler künftig früher entdecken?" Denn, je mehr Zeit vergeht, bis einen Fehler dem Verursacher gemeldet wird, desto weniger fühlen sich die Verursacher noch für den Fehler verantwortlich. Folge ist, dass die Verantwortlichkeit für Qualität auf den nachfolgenden Prozessschritt delegiert wird. Dadurch sinkt die Qualität noch mehr. Häufig ist eine weitere Folge, dass sich Schuldzuweisungen zwischen Entdecker und Verursacher etablieren. Zusätzlich kostet es unnötig viel Entwicklungsbudget, Fehler später anstatt früher zu entdecken.

Die Faustregel, dass sich die Fehlerkosten von Prozessschritt zu Prozessschritt verzehnfachen, ist nach wie vor richtig.

Qualitätsverbesserung durch gute Anforderungen

Die bereits erwähnten 70 Prozent Fehlerkosten, die auf die mangelhafte Anforderungen zurückzuführen sind, können nur wenig bis gar nicht durch dynamisches Testen verringert werden. Eine funktionierende Review-Kultur reduziert diese Art der Fehlerkosten um etwa 75 Prozent. Wasch mich, aber mach mich nicht nass?

Viele Reviews werden eher halbherzig durchgeführt. Meist steht nicht die Verbesserung des Arbeitsergebnisses im Vordergrund, sondern Abteilungsinteressen, Fingerpointing oder Lustlosigkeit. Eine gelebte Review-Kultur ist eins der effektivsten Mittel, um Fehler frühzeitig zu finden oder zu vermeiden. Hierbei meine ich nicht nur das Prüfen von Dokumenten. Peer-Review-Verfahren in der Entwicklung vermeiden Codefehler und tragen zur Wissensverteilung bei.

Bei hochbrisanten Codeteilen ist ein Mob-Coding noch besser. Aber warum nicht auch mal Peer-Testing? Sowohl bei Dokumenten als auch bei Code ist diese besondere Art des Testens sehr effektiv. Der Entwickler oder Ersteller eines Dokuments/Codes erhält Einsicht in die Methodik eines Testers. Parallel wird mit dem Vier-Augen-Prinzip die Qualität des Testobjekts massiv verbessert. Besonders durch den unterschiedlichen Blickwinkel der beteiligten Personen sind die Qualitätssteigerung und der Wissensaustausch sehr wertvoll. Zudem fördert es den Zusammenhalt und gegenseitiges Verständnis, was sich langfristig in der verbesserten Kommunikation widerspiegelt. Dies führt zusätzlich zu einer Fehlerverringerung.

Alles in allem sind Reviews, im Großen wie im Kleinen, ein sehr wertvolles Mittel, um Fehler zu verhindern, bevor sie im Code entstehen.

Übergeordneter Qualitätsansatz im agilen Umfeld

Im agilen Kontext werden häufig schon gute konstruktive QS-Mittel wie Retrospektiven, Peering und Reviews angewendet. Allerdings sind die agilen Teams häufig sehr mit ihren eigenen teaminternen Aufgaben beschäftigt. Dadurch geht oft der Blick für das große Ganze verloren.

Auch bei Teams, die in der Transition von klassisch zu agil stecken, herrscht nicht selten eine Unsicherheit. Der Testmanager ist in der agilen Welt nicht mehr vorgesehen. Gleichwohl sind dessen Aufgaben noch immer da. Diese müssen von den Teams übernommen werden. Nur welches Team macht was?

Hier ist es hilfreich, wenn eine Rolle etabliert wird, die die Wichtigkeit der Qualitätssicherung in die Teams trägt. Diese Rolle ist der Quality-Evangelist. Er hat auch die Aufgabe, die Teams anzuhalten, eine übergeordnete Koordination zu etablieren. Zum Beispiel in Form einer Gilde. In dieser Gilde können dann prozessuale Aspekte gemeinsam betrachtet werden. Auch hier ist ein Quality Storming über die Teamgrenzen hinweg hilfreich. Der Fokus liegt auf gemeinsamen qualitativen Standards und dem gegenseitigen Lernen aus Fehlern. Gemäß einem Ansatz des Clean Codes „Don't repeat yourself" wollen wir auch teamübergreifend Fehler nicht wiederholen.

Quellen

[1] *http://www.cecconi-partner.de/news-archiv/2014-03-30-softwarefehler-und-die-volkswirtschaft/*. (2019)

Georg Haupt
Oose innovative Informatik eG
gh@oose.de

Georg Haupts Motto lautet: „Aus der Praxis für die Praxis!" Er blickt als ISTQB-zertifizierter Tester und Quality-Evangelist auf 20 Jahre praktische Erfahrung für Soft- und Hardwaretests zurück. Er ist Mitglied in diversen Gremien zur Erstellung und zum Review von Standards wie Normen und Zertifizierungslehrplänen. Ebenso erfahren ist er in der Testautomatisierung, -koordination sowie -analyse. Auch wenn ihm die klassische Entwicklung nicht fremd ist, am wohlsten fühlt sich Georg Haupt in der agilen Entwicklung. Hier legt er besonderen Wert auf eine ausgewogene Mischung zwischen explorativen und automatisierten Tests. Das Entwickeln und Etablieren von Testmethoden und Prozessen ist für ihn langjährige Praxis. Privat ist er ein experimentell verspielter Technikfreak, hausautomatisierungsverrückt, über 20 Jahre DJ, Webradiomoderator und Cineast.

Mach mehr aus nervenden Meetings – vier wirkungsvolle Tipps

Armin Schubert, Emendare GmbH & Co. KG

Meetings sind oft energieraubende Zeitfresser. Mit einfachen Tipps und Initiative der Teilnehmer werden die Meetings zu Treffen, die Wirkung und Mehrwert haben. Armin Schubert hat eine Sammlung von grundlegenden Tipps zusammengestellt. Dabei berücksichtigt er auch das aktuell aufgrund der Corona-Krise immer mehr an Bedeutung gewinnende Home-Office.

Motivation

Die schlechte Nachricht vorweg: An der Qualität eines Meetings wird sich ohne aufmerksames und aktives Zutun der Teilnehmer nichts ändern. Wer über die Meetingkultur klagt, bekommt hier eine Anleitung mit vielen kleinen Ansatzpunkten. Jeder Ansatzpunkt kann angewendet werden und stellt einen Schritt auf dem Weg zu besseren Meetings dar. Wer nicht bereit ist, etwas an den vorhandenen Gewohnheiten zu ändern, darf diesen Artikel überspringen und aufhören, sich zu beklagen.

Einige mögen jetzt denken: „Ja… aber dafür haben wir doch unseren Coach/Teamleiter/Moderator! Ich mach da gar nichts!" Ein Moderator wird Impulse setzen und das Meeting „führen", dazu nimmt er seine Sonderrolle außerhalb des Teilnehmerkreises ein. Der Anspruch an unser aller Arbeit ist doch, dass jeder seinen Beitrag leisten wird. Also packen wir es an! Ich werde in diesem Artikel vier kurze und einfache Methoden vorstellen, die durch ihre Einfachheit überall anwendbar sind. Genießen Sie den Erfolg beim Ausprobieren!

Einen immer größer werdenden Teil unserer Arbeitszeit wirken wir im Homeoffice, remote oder in Online-Konferenzen. Besonders in der jetzigen gesundheitlichen Notlage und der Kontaktsperre zur Eindämmung des Coronavirus arbeiten die, denen es möglich ist, von zuhause aus. Damit auch diese Art von Meetings wirkungsvoller wird, werden wir diese Besonderheit ebenfalls berücksichtigen.

1. „Jede Kommunikation hat ein Ziel" sorgt für Fokus und macht deutlich, welchen Mehrwert wir schaffen.
2. „Positiven emotionalen Rahmen schaffen" ist die Grundlage für eine langfristige und gemeinsame Erfolgsgeschichte.
3. „Mit Erfolg(en) arbeiten" ist die zentrale Komponente, mit der wir für uns persönlich und im Team Motivation und Energie aufbauen.
4. „Ziele aufladen" hilft, langfristig motiviert und begeistert zu bleiben.

„Jede Kommunikation hat ein Ziel!"

Jeder Teilnehmer des Meetings kann zum Erfolg desselben beitragen. Wenn viele Teilnehmer die folgenden Tipps befolgen, werden die Treffen einfacher, produktiver und besser.

Abbildung 1: Klare Ziele im Meeting (© Cyber Manufaktur GmbH)

Eine der ersten Veränderungen, die Ihre Meetings positiv beeinflussen wird, ist die klare und offene Zielsetzung. Folgende Schritte führen hier zum Erfolg:

1. Das Ziel vor dem Termin festlegen oder abfragen
2. Das Ziel im Termin noch einmal abfragen und gegebenenfalls bestätigen oder anpassen
3. Dieses Ziel durch eine sichtbare Agenda fortschreiben
4. Alle Teilnehmer in die Pflicht nehmen, aktiv auf das Ziel hinzuarbeiten

Wenn das Ziel des Termins klar und abgegrenzt ist, können alle den Fokus halten *(siehe Abbildung 1)*. Kurz noch mal innehalten, um den gerade im Meeting erreichten Erfolg sichtbar und spürbar zu machen, und dann die Zusammenkunft zu beenden. Nicht selten übernimmt diese Moderationsaufgaben wie das Visualisieren der Ziele und das Einhalten eines Zeitrahmens ein ScrumMaster oder Moderator. Ich bin sicher, dass auch normale Teilnehmer einen wesentlichen Beitrag zum Gelingen der Meetings leisten werden.

Ein klares Ziel stiftet den Kontext, in dem sich das Zusammentreffen bewegen soll. Ein kreativer Austausch setzt eine andere Art der Teilnahme voraus als ein Entscheidungs- oder Eskalationsmeeting. Es hilft, diese impliziten Dinge explizit zu machen. Denn jetzt können sich alle Teilnehmer auf das Ziel ausrichten. Folgende Fragen können zum Klären des Ziels benutzt werden:

- Was müssen wir heute entscheiden, um erfolgreich zu sein?
- Woran bemerken wir, dass dieses Meeting ein Erfolg ist?
- Wenn wir nur ein Thema abschließen könnten, welches müsste das sein?
- Welche Punkte müssen heute entschieden werden, um das Treffen beenden zu können?
  - ...

Remote-Termine werden oft als anstrengender und unpersönlich wahrgenommen. Hier bietet es sich an, kurze und sehr fokussierte Termine zu machen. Die Anzahl der Ziele und Themen lieber klein halten und dafür mehrere sehr kurze Termine von circa 20 Minuten ausmachen. Es empfiehlt sich, Termine mit einer Länge von mehr als 45 Minuten durch lange Pausen zu unterbrechen. Wenn diese Pausen zum Teil für lockeren Austausch im Team genutzt werden, bleibt die persönliche Bindung erhalten.

In diesem Zuge möchte ich noch einen Hinweis geben: Ich werde oft gefragt, ob man all diese Tipps auch auf Meetings anwenden kann, die zum Beispiel der Chef, der Projektleiter oder der Vorstand einberufen haben. Gerade diese Menschen sollten über Ihren Beitrag zu kürzeren, erfolgreicheren und effektiveren Entscheidungsrunden dankbar sein. Machen Sie auch gegenüber Ihren Vorgesetzten die Ziele sichtbar und sie werden euch verstehen. Schließlich sind die Vorgesetzten für den Rahmen und die psychologische Sicherheit verantwortlich.

„Positiven emotionalen Rahmen schaffen“

Wie ich schon angemerkt habe, darf im konstruktiven Austausch die Hierarchie und Position der Teilnehmer keine Rolle spielen. Allerdings spielt die innere Haltung, also die Positur, sehr wohl eine Rolle.

„Positur statt Position!“

Meetingteilnehmer haben gute und schlechte Tage. Durch eine positive, neugierige und aktive Ausrichtung schaffen wir eine Stimmung, die die anderen Teilnehmer mitreißt, selbst die positivere Haltung einzunehmen *(siehe Abbildung 2)*. Wir alle nehmen die Emotionen unserer Gegenüber unterbewusst und zu jeder Zeit wahr. Um hier für einen positiven Rahmen zu sorgen, ist es unser aller Aufgabe, die Ziele des Projekts so lebendig und emotional wie möglich in unserem Bewusstsein zu halten. Warum? Weil uns das motiviert! Somit hilft es uns, unsere Ziele zu erreichen.

> ***„Probleme kann man niemals mit derselben Denkweise lösen, durch die sie entstanden sind.“***
>
> Albert Einstein

Wenn unsere Sprache von Problemdenken und Risikomanagement geprägt ist, schaffen wir es oft nicht, in einen geistigen Zustand zu finden, der beim Lösen der Probleme hilft. Ja, es ist sinnvoll, Probleme zu analysieren und daraus zu lernen. Es ist sinnvoll, mit Risiken bewusst umzugehen und sich vorzubereiten. Noch viel sinnvoller ist es, die Chancen und Möglichkeiten des Projekts ins Bewusstsein zu rufen, die bisherigen Erfolge zu feiern und dadurch eine emotionale Bindung zum Projektfortschritt herzustellen. Wir schaffen eine gute

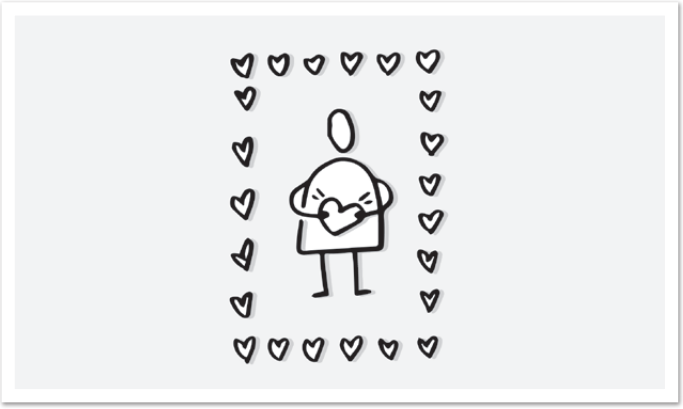


Abbildung 2: Positiver emotionaler Rahmen (© Cyber Manufaktur GmbH)

Kommunikation, indem wir den Problem- und Chancendenkern zuhören und beiden Gruppen klarmachen, dass insbesondere die Ausgewogenheit der Perspektiven dem Projekt weiterhilft. Der Problemdenker hat eine positive Absicht mit seiner Kritik, genau wie der Träumer ein Ziel verfolgt, das das Projekt bereichern wird. Wir dürfen immer einen ausgewogenen Punkt zwischen naiver Träumerei und schwarzsehender Kritik einnehmen.

In einzelnen Gesprächen kann jeder einzelne Teilnehmer mit einfachen Fragen für eine Stimmungsänderung im Meeting sorgen:

- Worüber hat sich unsere Testkundin Frau Müller am meisten gefreut?
- Wenn wir diesen Teil des Projektes erfolgreich abschließen, welchen Mehrwert haben unsere Kunden dann?
- Nehmen wir mal an, wir finden eine Lösung für das aktuelle Konstruktionsproblem, die zehn Prozent besser ist als die aktuelle Variante. Was wäre dann an heute undenkbaren Features möglich?
- Welchen positiven Nebeneffekt können wir mit diesem Vorgehen noch erreichen?
- Welche positive Absicht verfolgt Herr Schneider mit seiner Kritik? Wie können wir darauf sinnstiftend reagieren?

In Remote-Sessions ist es eine zusätzliche Herausforderung, Emotionen zur Redezeit zu spüren oder sie genau in dem Maße zu senden, wie es beabsichtigt war. Es wird den Teilnehmern helfen, wenn die positive Absicht bewusst und deutlich formuliert wird. Darüber hinaus expliziter über die Gefühle zu sprechen, sorgt dafür, dass der Interpretationsspielraum sehr klein wird, wenn die Online-Sitzung beendet ist und man wieder allein zuhause sitzt. Dafür aufgewendete Zeit sorgt für Identifikation mit Team und Produkt und ist daher gut investiert. Beispielfragen:

- „Du hast angemerkt, dass… Wobei hilft uns diese Perspektive in der jetzigen Situation?"
- „Mich hat erreicht, dass du dich … fühlst. Welchen Wunsch hast du für die weitere Zusammenarbeit?"

Selbst einfache und offene Formate wie virtuelle Mittagessen und die Online-Kaffeeküche sorgen für lockeren Austausch und halten die Bindung zum Team aufrecht. Das hilft, auch beim Remote-Arbeiten mit Emotionen und stressigen Situationen erfolgreich zu sein.
Wir sollten gleich viel Zeit in das Risiko- und Problemdenken investieren, wie wir in die Chancen- und Möglichkeitsperspektive stecken. Damit sind wir nicht nur motivierter, sondern auch kreativer, umsichtiger und wirksamer!

„Mit Erfolg(en) arbeiten"

Wenn wir nun also einen positiven, zielgerichteten Rahmen für unsere Kommunikation geschaffen haben, können wir zum wichtigsten Punkt kommen. Ab jetzt können wir nicht nur am Erfolg, sondern mit dem Erfolg arbeiten.

Während wir in den Meetings sitzen und Entscheidungen treffen oder unseren kreativen Ideen folgen, sammeln wir kleine Erfolge für das Produkt oder Projekt. Zu meiner großen Überraschung wird mit diesen Erfolgen nicht weiter in der Interaktion gearbeitet. Der Psychologe Albert Bandura hat in den 1970er Jahren das Konzept der Selbstwirksamkeitserwartung („Vertrauen in die eigene Tüchtigkeit") beschrieben *[1]*. Dahinter verbirgt sich, dass wir Menschen aus unseren Erfahrungen lernen, dass wir in der Lage sind, unser Umfeld selbst zu unserem Vorteil zu gestalten.

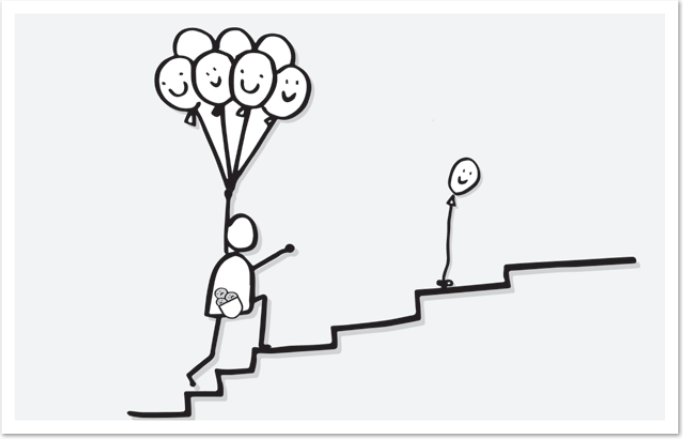


Abbildung 3: Selbstwirksamkeit Aufwärtsspirale (© Cyber Manufaktur GmbH)

Abbildung 4: Selbstwirksamkeit Abwärtsspirale (© Cyber Manufaktur GmbH)

Wir steigen eine Treppe hoch und sammeln immer wieder Erfolge, die uns Auftrieb schenken. Unser Weg ist nicht frei von Problemen – wir haben auch ein paar Rückschläge und negative Erfahrungen gesammelt. Jedoch sorgen die vielen guten Dinge in unserem Erfahrungsrucksack dafür, dass sich die Treppen immer flacher anfühlen. Die Herausforderungen der Zukunft fallen uns leichter *(siehe Abbildung 3)*.

Das hat zur Folge, dass wir durch die Steigerung an Selbstsicherheit, Kreativität und Mut mit höherer Wahrscheinlichkeit erfolgreich sein werden *[2]*. Das nächste Erfolgserlebnis steigert somit wieder unsere Selbstwirksamkeitserwartung und damit erneut unsere Erfolgschancen. Diese positive, selbstverstärkende Spirale können wir alle für uns selbst und unsere Meeting-Teilnehmer nutzen.

Während wir am nächsten Erfolg arbeiten, sollten wir uns sehr transparent machen, welche Erfolge wir bisher erarbeitet haben. Salopp gesagt, sollten wir uns immer wieder für das Erreichte auf die Schulter klopfen, denn es stärkt uns den Rücken für die Herausforderungen, die vor uns liegen. Ab jetzt arbeiten wir nicht mehr nur **am** Erfolg, jetzt arbeiten wir **mit** den Erfolgen der Vergangenheit.

Es könnte sein, dass früher die Abwärtsspirale vorhanden war: Wenn wir unsere Treppe erklimmen und uns immer wieder mit negativen Dingen, überzogenen Erwartungen, negativen Einschätzungen, Angst

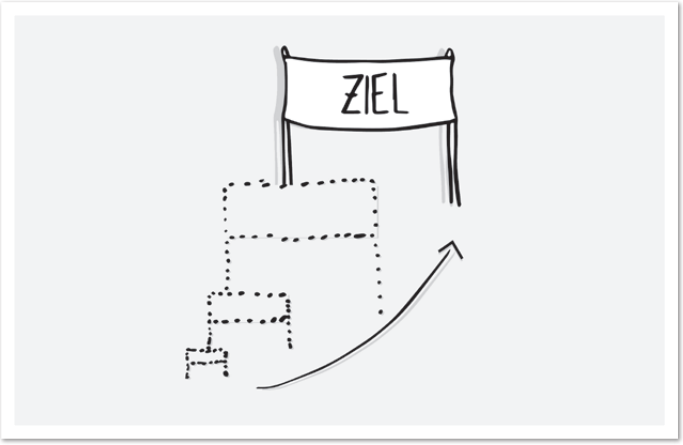


Abbildung 5: Ziele aufladen (© Cyber Manufaktur GmbH)

und Stress beladen, werden die Stufen unserer Treppe immer höher. Wir machen uns selbst den Weg unnötig schwer *(siehe Abbildung 4)*.

Wenn wir uns deutlich machen, dass viele kleine Erfolge einen deutlichen Einfluss auf unsere Erfolgschancen der Zukunft haben, wird schnell klar, dass wir jeden noch so kleinen Erfolg kurz und bewusst feiern sollten. Es geht nicht um die Qualität des einzelnen Erfolgs, sondern um die Quantität der kleinen Erfolge. Eine kleine Liste von Dingen, die ich bei agilen Transformationen, die ich begleite, immer wieder einsetze:

Eine Timeline, die permanent im Teamraum sichtbar ist.

- Sie dient der Visualisierung der Vergangenheit inklusive der möglicherweise erlittenen Rückschläge. Denn diese Rückschläge haben wir überwunden. Das heißt, sie sind uns als stressig und anstrengend in Erinnerung, haben aber durch die Überwindung eine positive Konnotation erhalten. Wir sind heute nicht nur stolz auf die Erfolge der Vergangenheit, sondern insbesondere die überwundenen Hindernisse machen uns glücklich.
- Sie dient der Visualisierung der Ziele in der Zukunft. Damit macht sie deutlich, woran und zu welchem Zweck wir gerade an der aktuellen Problemstellung arbeiten. Im nächsten Abschnitt werde ich noch weiter auf attraktive Ziele eingehen.

Dankbarkeitsrunden

- Am Ende eines Meetings nehmen wir uns kurz Zeit, um uns gegenseitig Dank auszusprechen. Schon diese kleinen Erfolge auszutauschen hilft, den Stresslevel drastisch zu reduzieren und die Selbstwirksamkeit wahrzunehmen. Nicht selten kommen hier Kleinigkeiten zutage, die im Stress des Arbeitsalltags verloren gegangen sind. Wenn sich der Kollege noch an die freundlichen und aufmunternden Worte im letzten Meeting erinnert, waren sie wirksam und wertvoll.

Journaling

- Immer häufiger treffe ich auf Menschen, die im Arbeitsumfeld keine Erfolge mehr für sich wahrnehmen oder in Anspruch nehmen. Genau wie oben beschrieben, fängt dann die selbstverstärkende Spirale an zu drehen, nur dieses Mal leider zum Nachteil des Einzelnen. Um diesen Menschen ein einfaches und schnelles Mittel an die Hand zu geben, gibt es das Journal [3]. Dort werden sie Dinge sammeln, die mit einem positiven Gefühl verbunden waren. Dieses Positiv-Tagebuch führt jeden Abend aus der Abwärtsspirale heraus. In den allermeisten Fällen dreht sich das Vorzeichen der Spirale sehr schnell und der Arbeitsalltag macht deutlich mehr Spaß.
- Eine Kombination aus der öffentlichen Timeline und dem Journal könnte ein Team-Logbuch sein. Alle wichtigen Dinge dort kurz sammeln und in längeren Abständen mal darin stöbern. So kann die Person „Team" aus der Vergangenheit Energie für die spannende Zukunft sammeln.

Kurz zusammengefasst könnte man dieses Konzept „Tue Gutes und rede darüber!" nennen. Insbesondere im inneren Dialog wird dieser Leitsatz schnell Früchte tragen. Natürlich gibt es noch viele weitere Methoden, die diesen positiven Dialog fördern und die gerne in Meetings zur Wirkung gebracht werden können.

Jede dieser Methoden kann auch im Homeoffice angewendet werden. Ein Terminwechsel dauert im Bürogebäude drei bis fünf Minuten, weil ich den Raum wechseln muss. In der digitalen Besprechungswelt dauert dieser Wechsel nur zwei bis drei Klicks. Unser Gehirn profitiert davon, wenn wir zwischen den Meetings…

- eine Pause einbauen und durchatmen,
- unsere Aufgaben und Erfolge des Tages visualisieren (Post-ist an der Wand),
- für uns selbst klarmachen, was wir gerade erreicht haben,
- für uns selbst herausfinden, wieso wir uns auf den nächsten Schritt freuen und was genau dieser beinhalten wird.

„Ziele „aufladen"!"

Immer wieder kommen wir an Projektzielen, persönlichen Zielen, Sprintzielen, Teamzielen, Jahreszielen, Neujahrszielen etc. vorbei *(siehe Abbildung 5)*. Anfang dieses Jahres sprach ich mit mehreren Auszubildenden über agile Methoden und die Wichtigkeit von Zielen im agilen Umfeld. Wir kamen schnell überein, wie ein attraktives Ziel beschaffen sein muss und warum es uns hilft. Nach wenigen Minuten meinte ein Teilnehmer: „Wenn ich das, was wir gerade besprechen, auf mich als Mensch anwende, schmeckt mir die nächste Zigarette überhaupt nicht, weil sie mich von meinem langfristigen Ziel abhält!" Wir haben im Einzelgespräch noch intensiver über seine persönlichen Ziele gesprochen und diese noch ein wenig „emotional aufgeladen". Am Anfang der darauffolgenden Woche habe ich ihn angeschrieben und gefragt, wie sein Wochenende war, und nicht ohne Stolz kam ein „rauchfrei" zurück. Was war passiert?

Abbildung 6: Erfolge feiern (© Cyber Manufaktur GmbH)

Der junge Mann genießt die Arbeit beim aktuellen Arbeitgeber, weil er dort Freiheit empfindet. In seiner Tagesgestaltung, in der Definition seiner Aufgaben und sogar in seinem Teamumfeld ist viel Gestaltungsrahmen. Wir haben herausgearbeitet, dass für ihn als Privatmensch der Ausdruck von Freiheit ganz eng an Autos und schnelles Fahren geknüpft ist. Der große Traum ist eine Fahrt im Renntaxi auf dem Nürburgring oder ein Wochenende im Fahrerlager. Als ich ihn bat zu beschreiben, auf welche Dinge im Fahrerlager er sich freut, konnte ich sehen, dass er die Abgase riechen, die heulenden Motoren hören und den Fahrtwind spüren konnte. Genau das ist seine Form von Freiheit!

Leider sind solche Renntaxifahrten und Fahrerlagereintrittskarten nicht sehr günstig. Das heißt, mit dem aktuellen Einkommen als Azubi, nur langfristig abbildbar. Der junge Mann hat während unseres Gesprächs bemerkt und später gefestigt, dass jede Zigarette, die er raucht, nicht nur seiner Gesundheit schadet, sondern seinem Freiheitsbedürfnis entgegenwirkt. Mit jedem Tag, an dem keine Schachtel Zigaretten ver(b)raucht wird, bringen wir uns dem Ziel um sieben Euro näher. Nach etwas mehr als zwei Monaten wäre das Event zu mehr Freiheitsgefühl bezahlbar.

Warum gewinnt die Zigarette dennoch so oft? Weil wir unsere Ziele nicht emotional aufladen. Wir sind uns selten im Klaren darüber, welche Gefühle wir mit schnellem Fahren verbinden. Wir reden nicht darüber, wie sehr wir uns auf den Motorenlärm freuen, wie deutlich wir den Fahrtwind in den Haaren spüren, was genau FREIHEIT für uns bedeutet. Genau hier liegt aber die Chance, uns selbst und andere zum Erreichen der Ziele zu motivieren. Wenn wir uns klarmachen und darüber austauschen, welche Gefühle wir damit verbinden,

- … dass wir eine bestimmte Datenbankstruktur umgebaut haben,
- … dass wir eine bestimmte Wirkung im Marketing erzielt haben,
- … dass wir ein bestimmtes Feature beim Kunden aktiv haben,
- … und das dann alles vorher entsprechend aufladen, erscheint das Erreichen des Ziels, der Kennzahl, selbst nicht mehr wichtig. Ab jetzt leitet die Vorfreude auf das Gefühl unser Handeln. **Das Erlebnis der Umsetzung ist immer mehr wert als das Ergebnis der Umsetzung!**

Natürlich haben fünf Menschen unterschiedliche Gefühle beim Erreichen eines Meilensteins. Aber es befeuert unsere Selbstwirksamkeit, wenn wir wissen, dass wir dafür sorgen können, dass unsere Kollegen ein Gefühl der Freiheit überkommt, wenn zum Beispiel endlich die alte Frontendtechnologie abgeschaltet werden kann. Auch hier ein paar Dinge zum Ausprobieren:

- Mit den Kollegen über die Ziele sprechen. Zu jedem Ziel sollten alle mit der Zielerreichung verbundenen Gefühle ausgetauscht werden. Jetzt wissen alle, wie viel Gutes an dem Ziel hängt.
- Eine Zielwand gestalten, auf der neben dem Ziel – gerne auch SMART-Ziele oder OKR-Ziele – auch die Gefühle hängen. Es geht nicht mehr nur um sterile KPIs. Es wird auch deutlich, dass das Team Emotionen mit der Zielerreichung verbindet, zum Beispiel Stolz, Befriedigung und Anerkennung.
- Bei Teilerfolgen, wichtigen Erkenntnissen und gelösten Problemen im Team die Gefühle thematisieren. So wird „das Erlebnis ist mehr wert als das Ergebnis" in der Zusammenarbeit verankert und Motivation generiert *(siehe Abbildung 6)*.

Bei der Remote-Arbeit können wir deutlicher über unsere Emotionen sprechen. Um das zu erleichtern, folgende Vorschläge:

- Hinter dem Monitor (wo niemand außer dem Teilnehmer selbst hinschauen kann) mit Post-its oder einem Plakat Emotionen visualisieren (zum Beispiel ein Foto von der letzten erfolgreichen Projektabschlussparty, die Gefühle aufschreiben, die man mit dem Projekt verbindet, etc.). Jetzt kann ich während des Online-Meetings meinen Blick schweifen lassen und mir in Erinnerung rufen, warum dieses Projekt wichtig und erstrebenswert ist.
- Bei erreichten Teilerfolgen ganz bewusst auch darüber sprechen, dass sich bereits Emotionen einstellen: „Ich kann schon jetzt eine deutliche Erleichterung in der Arbeit spüren… wir sind zwar noch nicht ganz fertig, aber durch das Datenbankupdate schlafe ich besser!"
- Da in der Remote-Session die Emotion vielleicht nicht klar transportiert wird, hilft auch hier ein kleiner Check-in am Anfang des Meetings: „Welches Gefühl verbindest Du mit dem Projekt an dem wir jetzt gleich arbeiten?"

Viele dieser Tipps sind selbstverständlich und gehören in anderen Lebensbereichen zum Alltag. Leider gibt es im Berufsalltag Bereiche und Firmen, in denen Gefühle keine Rolle spielen (dürfen). Gerade dort können durch die Thematisierung der Gefühle große positive Veränderungen erreicht werden. Ganz leicht.

Referenzen

[1] Wikipedia: *https://de.wikipedia.org/wiki/Selbstwirksamkeitserwartung*
[2] Buch „Positivity" von Barbara Fredrickson
[3] Website: *https://bulletjournal.com*

Armin Schubert
Emendare GmbH & Co. KG
armin.schubert@emendare.de

Armin Schubert ist einer der Geschäftsführer der Emendare GmbH & Co. KG, einer agilen Unternehmensberatung in Karlsruhe. Er arbeitet seit 2001 in Führungspositionen und setzt seit 2011 agile Methoden ein. Er schafft als Sprecher, Coach und Teamflüsterer eine Umgebung, in der die agile Transformation funktioniert.

In seinen Vorträgen und Impulsen zeigt er, wie Positivität und Dankbarkeit den Alltag der Mitarbeiter verbessern und dadurch eine große Veränderung möglich machen. Mehr über seine Vorträge erfahren Sie hier: https://www.emendare.de/team/armin-schubert/

Sag doch einfach mal Nö!

Sabine Wojcieszak, getNext IT

Wer kennt das nicht? Immer wieder gibt es Situationen, in denen wir eigentlich „Nein" sagen möchten. Doch was passiert? Wir sagen „Ja" und wissen schon in dem Moment, dass das ein Fehler war. Besonders im beruflichen Umfeld gilt das „Nein" häufig als unprofessionell und wird als Verweigerungshaltung gesehen. Doch ist das wirklich so oder sind wir alle es uns selbst, unserem Team, dem Unternehmen, der Qualität und auch dem Kunden schuldig, zum richtigen Zeitpunkt einfach auch mal „Nein" zu sagen?

Ein x-beliebiger Vormittag an einem x-beliebigen Tag in einem x-beliebigen Unternehmen irgendwo auf der Welt: Claudia (Peter, Stefanie oder eine andere x-beliebige Person) sitzt an ihrem Schreibtisch und ist vertieft in ihre Arbeit. Der Chef kommt zu ihr und sagt: „Claudia, du musst mir helfen. Ich habe morgen früh um 9 Uhr ein Meeting mit einem Kunden. Kannst du dir bitte mal schnell die Anforderungen hier anschauen und mir sagen, wie lange ihr dafür braucht? Den Auftrag wollen wir ganz unbürokratisch morgen unter Dach und Fach kriegen. Ist auch nicht so viel, was da drinsteht. Außerdem kennst du dich ja super damit aus – genau dein Thema! Ich habe das Dokument mit dir geteilt." Einen kurzen Moment lang zögert Claudia, da der Realist ihres inneren Teams laut schreit: „Du musst Nein sagen! Du hast so viel Arbeit auf dem Tisch, die fertig werden muss, damit das Team den Termin halten kann. Martin braucht deine Ergebnisse, um weitermachen zu können!" Doch dann überlegt Claudia kurz und hört sich Ja sagen. Ihr Chef ist zufrieden, bedankt sich höflich und geht. Und er lässt eine umgehend gestresste Claudia zurück, die weiß, dass sie genau drei Möglichkeiten hat:

1. Das Team hängen lassen und nicht pünktlich liefern.
2. Den Chef hängen lassen und nicht pünktlich liefern.
3. Den Arbeitstag über das Maß hinaus verlängern und Überstunden machen.

Da Claudia weder Stress mit dem Team noch mit ihrem Chef haben möchte, entscheidet sie sich für Variante 3 – zum fünften Mal in diesem Monat, der noch nicht einmal halb rum ist.

Damit sie sich mit dem Ja besser fühlt, sucht sie sich gute Gründe dafür: „Das ist manchmal einfach so, also muss ich mich auch nicht so anstellen. Was ist schon dabei, Überstunden zu machen. Andere machen das doch auch. Außerdem weiß der Chef dann doch auch, was er an mir hat. Er wird auch schon wissen, warum er zu mir damit kommt – die anderen kriegen das nicht so schnell hin wie ich. Das zahlt sich bestimmt irgendwann einmal aus. Und dann bekomme ich meine Belohnung dafür, dass ich mich immer so reinhänge. Außerdem hat er ja auch gesagt, dass das nicht so viel ist. Ich will ihn ja auch nicht enttäuschen, denn er ist ja echt ein netter Chef. So ist das eben, wenn man ein Profi ist. Und wenn es mal hart auf hart kommt – auf mich werden sie hier nicht verzichten können."

Zufrieden mit diesen Argumenten und dem dadurch entstandenen Heldengefühl öffnet sie die Datei und muss tief Luft holen: 66 Seiten warten darauf, von ihr gelesen und durchgearbeitet zu werden. Tapfer fängt sie an zu lesen und stellt spätestens auf Seite zehn fest, dass sie viele Fragen hat, ohne deren Klärung eine vernünftige Aussage über eine Umsetzungsdauer nicht einmal annähernd möglich ist. Seufzend macht sie sich auf den Weg zum Büro ihres Chefs, doch der ist schon weg. Feierabend, Zeit mit dem Freund und ausreichend Schlaf – davon kann sie sich verabschieden, Gleiches gilt für eine wirklich fundierte Antwort für ihren Chef.

Das ist erst der Anfang…

Dieses unrealistische Ja von Claudia hat einige Konsequenzen – nicht nur für Claudia selbst. Sie hat nicht ausreichend Zeit, um die offenen Fragen zu klären und eine belastbare Aussage für ihren Chef zu erarbeiten. Ihr Chef wiederum verlässt sich auf sie und nimmt ihre Antwort wahrscheinlich als aussagekräftig hin. Damit geht er in das Gespräch mit dem Kunden, aus dem er sich unbürokratisch einen Auftrag erhofft. Kommt es dann tatsächlich zu dem Auftrag, sieht sich das Team unter Umständen mit einer Aufwandsschätzung konfrontiert, die in keiner Weise die Realität darstellt. Das Projekt enthält somit schon ein hohes, nicht notwendiges Risiko, bevor es überhaupt gestartet ist. Gleichzeitig steht der Chef mit seinem Wort im Ernstfall beim Kunden schlecht da und verliert das Vertrauen.

Daran, wie diese Schätzung zustande gekommen ist, wird sich später niemand mehr erinnern – außer vielleicht Claudia. Dann sind Schuldzuweisungen wie „Ich hatte ja gar keine Chance. Du hast mich ja dazu gedrängt!" nicht selten das Ergebnis.

Vielleicht hat Claudia jedoch auch gemerkt, dass sie zu wenig über das Projekt weiß, und einfach noch einen Zeitpuffer auf ihre Schätzung raufgerechnet. Aber ist das Ergebnis dann noch marktfähig? Oder wird der Auftrag wegen der hohen Schätzung dann nicht zustande kommen?

Gleichzeitig gerät Claudia allerdings auch mit der im Team abgestimmten Arbeit unter enormen Druck. Führt sie ihre Aufgaben sorgfältig und mit der entsprechenden Qualität zu Ende? Oder wird sie nachlässig, um den mit Martin abgestimmten Termin halten zu können? Vielleicht wird sie auch einfach nur nachlässig, weil sie müde ist und sich nicht mehr richtig konzentrieren kann. Wie auch immer – eine schlechtere Qualität kann später durchaus zu Nacharbeiten, nicht bestandenen Tests und so weiter führen. Der Termin, den das Team halten soll, ist damit möglicherweise gefährdet oder führt zu Mehrarbeit beim Rest des Teams. Oder Claudia schafft ist trotz aller Bemühungen gar nicht erst, Martin die Arbeit entsprechend zu übergeben. Damit reicht sie den Druck sowohl an Martin als auch an den Rest des Teams weiter.

Aber auch Familie und Freunde sind von ihrem sich selbst überschätzenden Ja ebenso betroffen wie Claudia selbst. Jeder Mensch braucht Zeit, sich zu erholen und Kraft zu tanken für neue Aufgaben und Herausforderungen. Auch wenn wir in der IT nicht unbedingt körperlich arbeiten, sondern zu den sogenannten Kopfarbeitern, den kreativen Köpfen, gehören, braucht unser Körper Erholung. Freunde, Familie und Hobbies sind wichtig für uns, um uns zu erholen und zu regenerieren. Wenn dafür keine Zeit mehr bleibt oder auch im privaten Umfeld durch die Arbeitsbelastung Stress entsteht, fehlt dieser Erholungs- und Rückzugsort.

Besonders unser Gehirn benötigt die Auszeit und vor allem den Schlaf. Es arbeitet den ganzen Tag über auf Hochtouren und erlebt einen kontinuierlichen Strom von äußeren Reizen. Nach zirka 16 Stunden ist es Zeit für das menschliche Gehirn zu schlafen. In den verschiedenen Schlafphasen finden unterschiedliche, sehr wichtige Prozesse statt. Zum einen wird im Schlaf das Gehirn aufgeräumt. Es werden wichtige von unwichtigen Informationen getrennt und der Ballast wird sozusagen entsorgt, um Platz für neue Informationen zu schaffen. Da tagsüber durch die stetige Reizüberflutung keine Zeit ist, Informationen „sicher" im Gehirn abzulegen, werden diese Informationen zunächst einmal „zwischengeparkt". Im Schlaf dann werden emotionale, prozessuale und erklärende Erinnerungen verarbeitet und abgelegt. Nur dann können wir später wieder auf sie zugreifen und sie nutzen. Gleichzeitig zeigen Forschungsergebnisse, dass sich die Körpertemperatur im Schlaf um bis zu einem Grad absenkt und die Zellabstände im Gehirn sich vergrö-

ßern. So können Giftstoffproteine besser abtransportiert werden. Derzeit wird vermutet, dass durch diesen regelmäßigen Abtransport von Giftstoffen Krankheiten wie Alzheimer abgewehrt werden können. Gleichzeitig werden aktuell Schlafstörungen und zu wenig Schlaf mit Krankheiten wie Schizophrenie, Angststörungen und anderen psychischen Problemen in Verbindung gebracht. Dies wird stetig weiter erforscht.

Fakt ist jedoch, dass zu wenig Schlaf der Grund dafür ist, dass wir weniger leistungsstark und unkreativer sind. Es kommt zu verlangsamten Reaktionszeiten – und dafür reicht schon eine einzige Nacht mit zu wenig Schlaf aus! Unsere Leistungsfähigkeit am Folgetag ist somit eingeschränkt – Fehler, längere Bearbeitungszeiten und schlechtere Entscheidungen sind daher wahrscheinlich und wirken sich auf die aktuellen Arbeiten aus.

Das heißt also, dass, auch wenn ein Ja nur ein kleines Wort mit zwei Buchstaben ist, dieses mit Bedacht gesprochen werden sollte, denn die möglichen Kosten für ein unbedachtes Ja können enorm sein.

In seinem Buch „The Clean Coder" widmet Robert C. Martin (Uncle Bob) der Bedeutung, auch mal Nein sagen zu können, ein ganzes Kapitel. Wer sich wirklich professionell verhalten will, ist auch bereit, Nein zu sagen, wenn Nein die richtige Antwort ist. Dafür sind wir die Experten für unsere Arbeit.

Selbst wenn die „Kannst du mal schnell…"-Frage nicht zu Überstunden und Co führt, birgt sie viele Risiken für den späteren Projektverlauf. Und mal ganz ehrlich: Die Frage „Sag mal schnell, was schätzt du, wie lange ihr dafür braucht, nur so über den Daumen gepeilt?" ist in vielen Unternehmen im Alltag etabliert. Ebenso etabliert ist es, auf diese Frage schnell mit einer Einschätzung zu antworten, obwohl wir noch nicht die Details kennen und uns noch nicht damit auseinandergesetzt haben. Die Antwort, die eine Einschätzung enthält, ist ein implizites Ja, müsste aber eher ein „Nein, das kann ich so nicht beantworten" sein. Was hier am Anfang eines Projektes leichtfertig als Annahme in den Raum gestellt wird, kann am Ende den Projekterfolg kosten.

Typisch!?

„Typisch Chef!", lässt sich leicht zu diesem Beispiel sagen. Und ja, selbstverständlich hätte er erst einmal fragen können, ob Claudia überhaupt Zeit dafür hat. Und ja, er hätte vielleicht auch daran denken können, dass das Team einen Termin halten will. Und er hätte selbstverständlich auch wissen müssen, dass eine vernünftige Schätzung eben nicht so „schnell mal nebenbei" gemacht werden kann. Hätte, hätte…

Er hat diese Fragen jedoch nicht gestellt und lediglich versucht, sein Gespräch für den folgenden Tag vorzubereiten. Vielleicht hat er den Termin tatsächlich sehr kurzfristig mit dem Kunden vereinbart, vielleicht hat er aber auch vergessen, dass er für das Gespräch mit dem Kunden etwas vorbereiten muss. Oder der Chef ist es gewohnt, dass seine Aufgaben immer ganz kurzfristig von Claudia oder ihren Kollegen ohne Murren und Widerspruch erfüllt werden, egal wie hoch der Workload ist. Mit einem Nein von Claudia an der Stelle hätte sich die Gelegenheit zur zielgerichteten Kommunikation eröffnet, um entweder alte Muster zu durchbrechen oder eine bessere Lösung für das Problem zu finden.

Doch ähnliche Situationen finden sich nicht nur in dieser hierarchischen Struktur wieder. Auch zwischen Kollegen wird immer wieder das Ja ausgesprochen, obwohl ein Nein gemeint ist. Und ja, es kann durchaus auch der Eindruck „typisch Frau" entstehen. Vielleicht ist es auch so, dass Frauen grundsätzlich eher Ja sagen, obwohl sie Nein meinen. Nicht selten haben Frauen das Gefühl, mehr leisten zu müssen als ihre männlichen Kollegen, um dem Chef etwas zu beweisen. In vielen Umgebungen ist das auch der Fall. Dennoch ist es eben nicht „typisch Frau". Auch Männer finden sich regelmäßig in der Situation wieder, trotz besseren Wissens mit Ja geantwortet zu haben.

Es geht bei dieser Frage weder um Mann oder Frau noch um den Aspekt der Schuld – es geht vielmehr darum, Verantwortung für die eigene Arbeit zu übernehmen! Wer ein Profi sein will, muss für sich realistisch abschätzen können, was machbar ist und was nicht. Wer akzeptiert, dass die Qualität der Arbeit durch noch mehr Arbeitslast leidet, handelt nicht wie ein Profi. Wer trotz besseren Wissens Ja sagt zu Lösungen, die suboptimal sind, verhält sich nicht wie ein Experte.

Verantwortung übernehmen heißt eben nicht, einfach alles zu akzeptieren. Es heißt vielmehr, das zu tun, was gut ist, zum Ziel und zum Erfolg führt. In seinem Buch „The Responsibility Process" beschreibt Christopher Avery, dass wir erst dann Dinge verändern und verbessern können, wenn wir bereit sind, Verantwortung zu übernehmen. In unserem Beispiel ist die Wahrscheinlichkeit hoch, dass keine der beiden Arbeiten, die Claudia nun erledigen muss, wirklich nachhaltig erfolgreich sein werden. Sie opfert ihr Commitment mit Martin und dem Team für ihren Wunsch, als Heldin gesehen zu werden und Anerkennung zu bekommen. Dabei nimmt sie auch in Kauf,

Abbildung 1: Der Responsibility Process nach Christopher Avery (© Christopher Avery)

Abbildung 2: Die Matrjoschka als Symbol für das Innere Team (© Sabine Wojcieszak)

dass ihr professionelles Experten-Ich sich von weniger rationalen Gründen ruhigstellen lässt. Doch neben Gründen wie Anerkennung und dem Streben nach Heldentum gibt es auch noch weitere Gründe, die eng miteinander zusammenhängen: fehlender Mut und Angst.

Viele Menschen haben Angst, dass sie nicht respektiert werden, wenn sie Nein sagen. Sie haben Angst davor, andere zu verärgern oder sie vor den Kopf zu stoßen. Ihnen fehlt der Mut, sich für die professionelle Überzeugung stark zu machen, besonders wenn es sich dabei um eine vorgesetzte Person handelt. Solange wir allerdings nicht bereit sind, die Verantwortung voll zu übernehmen, suchen wir nach Erklärungen und Gründen, die uns ein besseres Gefühl geben. Denn eines ist klar: Wenn wir zwar wissen, dass ein Nein richtig wäre, wir aber trotzdem Ja sagen, fühlt sich das nicht gut an. Um dieses negative Gefühl loszuwerden, reagieren Menschen sehr unterschiedlich *(siehe Abbildung 1)*.

In der Phase des Leugnens versuchen wir, einfach darüber hinwegzusehen, dass ein Nein die bessere Antwort wäre. Andere suchen jemanden, den sie dafür beschuldigen können, dass sie selbst das Nein nicht sagen konnten. „Er hat mich überrascht und unter Druck gesetzt. Da konnte ich nicht Nein sagen!", ist die Entschuldigung, um sich besser zu fühlen. Die Verantwortung liegt somit nicht bei einem selbst, sondern bei anderen. Eine weitere Möglichkeit ist, dass die Umstände die Rechtfertigung dafür sind, dass das falsche Ja im Raum steht. „Ich war so in meine Arbeit vertieft, dass ich gar nicht richtig nachdenken konnte.", ist hier eine mögliche Aussage. Doch das Ergebnis ist ähnlich: Nicht wir haben das fehlende Nein zu verantworten, sondern die äußeren Umstände. Damit haben wir es noch immer nicht in der Hand, etwas zu verändern. Weitere Phasen sind das Schämen – „Es tut mir leid, dass ich Ja gesagt habe, obwohl ich es besser wusste" – und die Verpflichtung – „Ich muss dann beim nächsten Mal Nein sagen!". Auch sie führen noch nicht dazu, dass wir Verantwortung für unser Handeln übernehmen. Erst wenn wir erkennen, dass wir es sind, die Ja oder Nein sagen, und wir es damit in der Hand haben, können wir diese Verantwortung übernehmen.

Ja sagen kann jeder!

Um ehrlich zu sein: Es ist nicht immer leicht, Nein zu sagen. Und genau deshalb sind wir gefordert – die Profis. Ja sagen kann jeder – der Profi weiß, wann ein Nein die bessere Aussage ist! Dennoch ist ein Profi eben nicht nur ein Profi, sondern auch ein Mensch, der laut Friedemann Schulz von Thun, einem führenden Kommunikationswissenschaftler und Psychologen, „mehrere Seelen in der Brust" trägt. Er beschreibt in seinem Buch „Miteinander reden 3 – Das Innere Team und situationsgerechte Kommunikation" (1998) unter anderem das Beispiel einer fleißigen Studentin, von der sich ein wenig engagierter Kommilitone eine Mitschrift ausborgen will. Ein Teil von ihr möchte Nein sagen, weil sie das Gefühl hat, ausgenutzt zu werden, und sich darüber ärgert. Ein anderer Teil in ihr, der von Kollegialität geprägt ist, sagt, dass man sich gegenseitig helfen muss. Dadurch entsteht in ihr ein Zwiespalt. Von diesen Mitgliedern im Inneren Team gibt es noch mehr *(siehe Abbildung 2)*.

Da kann die ängstliche Seele neben der mutigen Seele, der professionellen und der neugierigen stehen. Die Frage ist immer, welche dieser Persönlichkeiten gerade die stärksten Argumente und die lauteste Stimme hat. Vielleicht bilden sie auch Koalitionen und gewinnen so an Macht. Diese „innere Pluralität" ist absolut menschlich. Und je nach unserer Fähigkeit, das „innere Betriebsklima" zu gestalten, kann diese Pluralität wertvoll oder zerstörerisch sein. Wer sich seines Inneren Teams bewusst ist, die einzelnen Mitglieder kennt, kann mit sich in die innere Diskussion einsteigen und so gegebenenfalls wertvolle Hinweise für weitere Entscheidungen identifizieren. Die besondere Kunst liegt darin, diese verschiedenen Stimmen wertzuschätzen und schnell wahrzunehmen, dass sich gerade eine Diskussion in uns anbahnt.

Der Preis des späten Neins

Im beruflichen Alltag haben wir allerdings oft gar nicht die Zeit, lange in eine Diskussion mit unserem Inneren Team einzusteigen. Wir müssen schnell entscheiden. Je länger wir ein Nein hinauszögern, desto schwieriger wird es am Ende, tatsächlich Nein zu sagen. Hinzu kommt, dass ein nicht ausgesprochenes Nein von Gesprächspart-

nern häufig als ein implizites Ja gewertet wird. Verschieben wir also unser Nein, ohne dem anderen genau klar zu machen, was wir benötigen, um eine qualifizierte Aussage treffen zu können, entsteht mehr Druck. Der Gesprächspartner geht davon aus, dass das fehlende Nein ein Ja bedeutet, und plant entsprechend weiter. Kommt dann das Nein zu einem späteren Zeitpunkt, ist der Frust groß: „Aber dann hättest du doch gleich Nein sagen können! Jetzt habe ich schon alles vorbereitet!", kann eine mögliche Antwort sein. Und was passiert? Schnell meldet sich das schlechte Gewissen – auch ein Teil des Inneren Teams – und wir sitzen wieder in der Zwickmühle und fühlen uns schuldig (Shame-Phase im Responsibility Process in *Abbildung 1*).

Nun kann die Idee sein, statt des Neins mit einem Vielleicht zu antworten. „Ich sehe mal, was ich machen kann", als gedachte langsame Hinführung zum Nein funktioniert in der Regel nicht. Die anderen verlassen sich darauf, dass das schon klappen wird! Wertvolle Zeit wird auch hier verschwendet. Das Ende ist ebenso unrühmlich und kann geprägt sein von Blame, Shame und Enttäuschung.

Sag, was du wirklich meinst!

„Do or do not. There is no trying!", sagte einst Yoda in Star Wars. Wenn wir eine Antwort geben, sollten wir uns sicher sein, dass wir auch wirklich das sagen, was wir meinen und auch halten können.

Ein Ja ist ein Commitment und bedeutet nach Roy Osherove: Sagen. Meinen. Machen! Haben wir Zweifel, dass wir dieses Commitment wirklich einhalten können, so sollten wir das entsprechend äußern und nicht einfach nur hoffen. Eine hilfreiche Möglichkeit ist zum Beispiel, ganz klar zu äußern, welche Voraussetzungen erfüllt sein müssen, um das Commitment einhalten zu können. So hätte Claudia ihrem Chef beispielsweise Folgendes sagen können: „Ich kann morgen mit in deine Besprechung mit dem Kunden kommen und meine Fragen dort stellen. Gerne erkläre ich dann auch, dass wir im Sinne eines erfolgreichen Projektes mehr Informationen benötigen, um eine verlässlichere Schätzung abgeben zu können. Dazu müsstest du allerdings den Termin für unser Projekt um einen Tag verschieben. Dann kann ich mich jetzt in die Unterlagen einlesen, morgen mit dir in das Gespräch kommen und mich danach weiter um unser derzeitiges Projekt kümmern."

Domenica DeGrandis beschreibt in ihrem Buch „Making Work Visible: Exposing Time Theft to Optimize Work & Flow", dass die Visualisierung von Aufgaben oft sehr hilfreich sein kann, besonders den Workload sichtbar und für andere nachvollziehbar zu machen. Kanban ist dabei das Mittel ihrer Wahl und hilft dabei, anderen zu zeigen, welche Auswirkungen es auf bereits abgestimmte Arbeiten hat, wenn wir kurzfristig noch weitere Aufgaben annehmen. Gegebenenfalls kann dann eine abgestimmte Priorisierung erfolgen.

Es ist immer hilfreich, ein Nein nicht allein im Raum stehen zu lassen, sondern eine mögliche Lösung mitzuliefern. So ist das Nein zwar zunächst wirklich ein Nein, kann aber immer noch in ein Ja überführt werden. Und genau das ist es, was von Profis erwartet wird: Echte Lösungen schaffen!

Übrigens: Wer Angst hat, seine Karriere durch ein Nein zu gefährden, irrt gewaltig! Karriereberater raten sogar dringend davon ab, immer nur Ja zu sagen. Die Hoffnung, dass die Ja-Sagerei am Ende die Karriere beflügelt, ist meist vergeblich. Ja-Sager, die dann alles auf sich nehmen, um das Ja auch einigermaßen möglich zu machen, werden häufig übersehen. Wer sich also Gedanken um seine Karriere macht, sollte sich professionell verhalten und damit auch dann einmal Nein sagen, wenn ein Nein angebracht ist – für mehr Zeit, mehr Qualität und weniger Risiko!

Sabine Wojcieszak
getNext IT
sabine@getnext-it.com

Als Enthusiastic Agile und DevOps Enabler bei getNext IT führt Sabine Wojcieszak Projekte für Unternehmen erfolgreich durch. Sie adressiert Thematiken wie Kommunikation, Führung, Meetings und Innovation und coacht sowohl Teams als auch Führungskräfte. Als Sprecherin ist sie auf internationalen Konferenzen unterwegs und schreibt Artikel für führende Fachmagazine und Blogs. Weiterhin lehrt und motiviert sie an der Fachhochschule internationale Master-Studierende in den Bereichen DevOps, Agiles PM und Open Source. Auf Twitter ist sie unter @SabineBendixen zu finden.

Mitglieder des iJUG

01 Android User Group Düsseldorf
02 BED-Con e.V.
03 Clojure User Group Düsseldorf
04 DOAG e.V.
05 EuregJUG Maas-Rhine
06 JUG Augsburg
07 JUG Berlin-Brandenburg
08 JUG Bremen
09 JUG Bielefeld
10 JUG Bonn
11 JUG Darmstadt
12 JUG Deutschland e.V.
13 JUG Dortmund
14 JUG Düsseldorf rheinjug
15 JUG Erlangen-Nürnberg
16 JUG Freiburg
17 JUG Goldstadt
18 JUG Görlitz
19 JUG Hannover
20 JUG Hessen
21 JUG HH
22 JUG Ingolstadt e.V.
23 JUG Kaiserslautern
24 JUG Karlsruhe
25 JUG Köln
26 Kotlin User Group Düsseldorf
27 JUG Mainz
28 JUG Mannheim
29 JUG München
30 JUG Münster
31 JUG Oberland
32 JUG Ostfalen
33 JUG Paderborn
34 JUG Passau e.V.
35 JUG Saxony
36 JUG Stuttgart e.V.
37 JUG Switzerland
38 JSUG
39 Lightweight JUG München
40 SOUG e.V.
41 SUG Deutschland e.V.
42 JUG Thüringen

iJUG Verbund

www.ijug.eu

Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, *www.ijug.eu*) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Stefan Kinnen. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Mylène Diacquenod
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Melanie Feldmann, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

Titel, Gestaltung und Satz:
Caroline Sengpiel,
DOAG Dienstleistungen GmbH

Fotonachweis:
Titel: Bild © stories | *https://de.freepik.com*
S. 9: Bild © rawpixel | *https://de.freepik.com*
S. 14: Bild © tohey | *https://de.123rf.com*
S. 22: Bild © Pop Nukoonrat | *https://de.123rf.com*
S. 23: Bild © sergign | *https://stock.adobe.com*
S. 24: Bild © Denis Ismagilov | *https://de.123rf.com*
S. 26: Bild © Oleksandr Koltukov | *https://de.123rf.com*
S. 28: Bild © Bakhtiar Zein | *https://de.123rf.com*
S. 30: Bild © Artem Egorov | *https://de.123rf.com*
S. 37: Bild © Sentavio | *https://stock.adobe.com*
S. 43: Bild © maximmmmum | *https://de.123rf.com*
S. 48: Bild © Teerapat Seedafong | *https://de.123rf.com*
S. 53: Bild © Ico Maker | *https://stock.adobe.com*
S. 56: Bild © vchalup | *https://de.123rf.com*
S. 60: Bild © Roman Samborskyi | *https://de.123rf.com*
S. 65: Bild © happystock | *https://de.123rf.com*

Anzeigen:
Simone Fischer, DOAG Dienstleistungen GmbH
Kontakt: anzeigen@doag.org

Mediadaten und Preise unter:
www.doag.org/go/mediadaten

Druck:
WIRmachenDRUCK GmbH,
www.wir-machen-druck.de

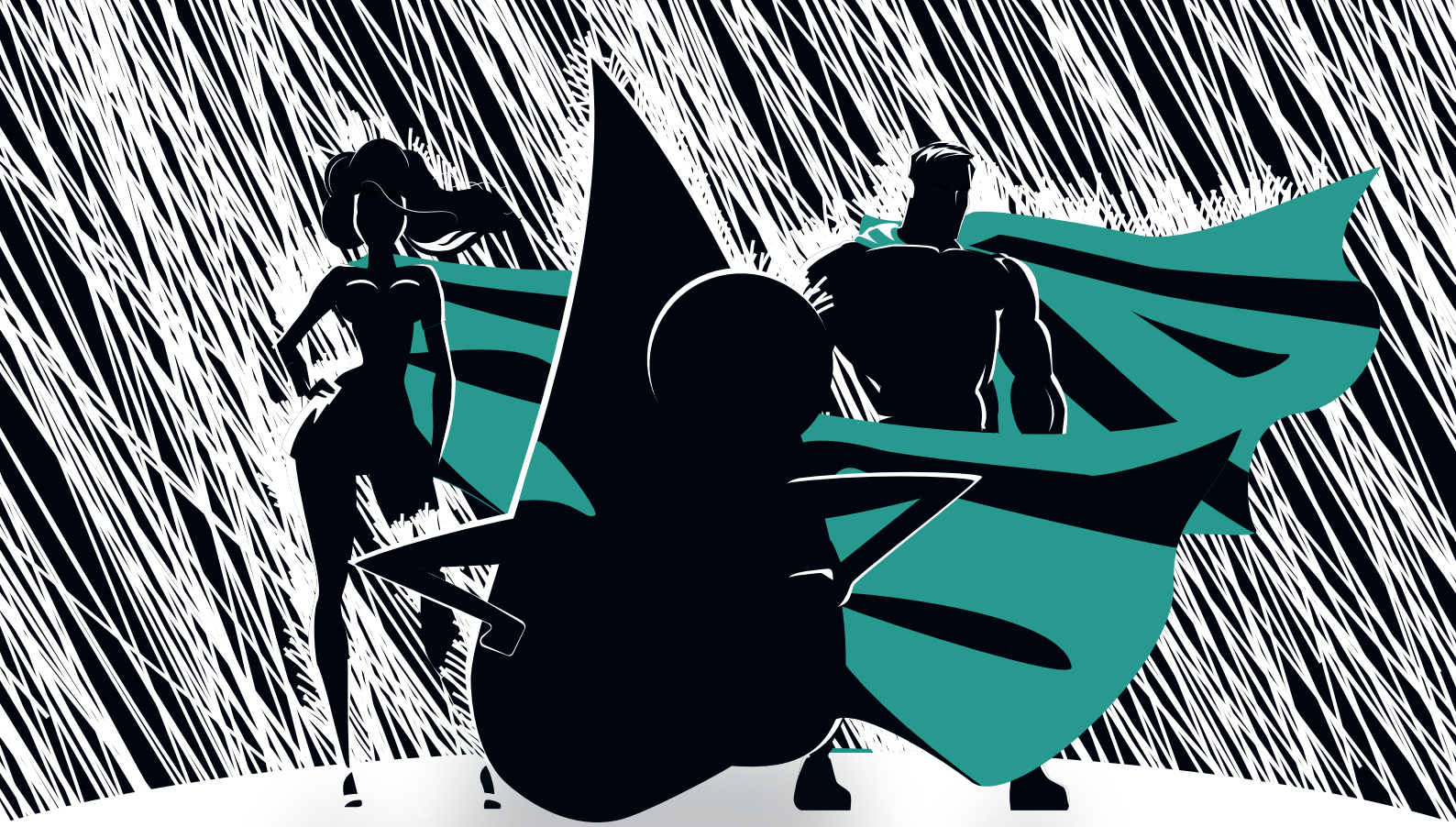
Inserentenverzeichnis

Java aktuell

Mehr Informationen zum Magazin und Abo unter:

https://www.ijug.eu/de/java-aktuell

iJUG Verbund
www.ijug.eu



IHR SEID UNSERE JavaLand HEROES!

Zahlreiche Teilnehmer, Referenten und Sponsoren haben entschieden, die JavaLand in der Krise mit einem finanziellen Beitrag zu unterstützen.

Wir sind überwältigt von eurer Großzügigkeit.
Schön, dass wir auf eine starke Community zählen können!

Bisher konnten wir dadurch über 46.000 Euro sammeln.
Insgesamt müssen aufgrund des Veranstaltungsausfalls mehr als 200.000 Euro gedeckt werden.

JavaLand

Besucht unsere Hall of Fame: