Strategien und Techniken zur Fehlerprävention

Wider die kognitive Last

Joachim Hofer

Irren ist menschlich, und Fehler passieren – insbesondere in der Softwareentwicklung. Qualitätssicherung soll dafür sorgen, dass die Software trotzdem fehlerarm zum Kunden geht. Aber vielleicht geht es ja besser? Können Entwickler Vorsorge betreiben, um das Entstehen von Fehlern zu verhindern?

Die klassische Qualitätssicherung ist oft hauptsächlich auf das Finden von Fehlern ausgerichtet, die bereits passiert sind. Im Idealfall findet sie die Fehler durch Einsatz mannigfaltiger Techniken wie statischer Codeanalyse, Reviews und Tests bereits so früh wie möglich nach dem Entstehungszeitpunkt. Aber wenn man Fehler von vornherein vermeiden will, muss man sich als Erstes fragen, wie Fehler überhaupt zustande kommen.

Zur Klärung dieser Frage lohnt es sich, zunächst zu untersuchen, wodurch allgemein menschliche Fehler verursacht werden. Hierfür ist die Kognitionspsychologie eine Fundgrube: Die wesentliche Erkenntnis ist, dass Fehler entstehen, wenn der Mensch kognitiv überlastet ist. Ein typisches Beispiel dafür ist die Überlastung des Arbeitsgedächtnisses: Dieses kann nur etwa fünf bis neun Dinge gleichzeitig parat halten [1]. Das ist sehr wenig, sodass so gut wie jeder Entwickler bei der täglichen Arbeit schnell an diese Grenze stoßen wird. Beispielsweise wenn er sich bei Codeänderungen merken muss, wo genau er schon etwas geändert hat, an welchen Stellen welche Änderungen noch ausstehen und was jeweils die zugehörigen Anforderungen sind, während er gleichzeitig noch die E-Mails im Auge behalten will oder gar zwischendurch von hilfesuchenden Kollegen unterbrochen wird. Ganz zu schweigen davon, dass er vielleicht nicht immer alle Tastenkürzel der Entwicklungsumgebung, alle Syntaxkonstrukte der Programmiersprache und die Semantik der verwendeten Bibliotheken in- und auswendig kennt.

Ein Ansatz für das Verhindern von Fehlern ist also, die Komplexität der Arbeitsumgebung zu senken. Dazu gehört neben dem eigentlich bearbeiteten Programmcode das Umfeld, in dem er entsteht, wie am obigen Beispiel zu sehen.

Komplexität im Arbeitsumfeld lässt sich oft durch organisatorische Maßnahmen senken: Weiterbildungen können dafür sorgen, kognitive Last vom Arbeits- ins Langzeitgedächtnis zu verlagern, wenn es um das Programmierhandwerk geht (Syntax, Bedienung der Entwicklungsumgebung, Tastaturkürzel, Schnittstellen zu Fremdbibliotheken oder -frameworks). Prozesse können helfen, unnötige Störungen von außen zu reduzieren. Zeitmanagement-Techniken wie Pomodoro [a] oder „Getting Things Done“ [2] unterstützen dabei, Ablenkungen und unnötigen kognitiven Ballast loszuwerden, der nicht die aktuell bearbeitete Aufgabe betrifft. Das simple Beenden des Mailprogramms während der Arbeit beseitigt einen der größten Störfaktoren. Und nicht zuletzt hat die Gestaltung des Arbeitsplatzes große Auswirkungen darauf, ob man fokussiert arbeiten kann.

Zwiefache Komplexität

Eine das Arbeitsumfeld betreffende Ergänzung zum alleinigen Senken der kognitiven Last wäre, die eigene Leistungs-/Konzentrationsfähigkeit zu steigern. Dafür gibt es vor allem ein erprobtes Mittel ohne Nebenwirkungen: das hinreichende Entspannen in der Freizeit und das Vermeiden exzessiver Überstunden. Das oberste Ziel dabei ist immer: den Kopf für die eigentliche Arbeit frei zu haben. So profan das auch klingen mag, es hilft bereits dabei, Fehler zu vermeiden.

Wenn das Arbeitsumfeld erst einmal stimmt, bleibt die für den Entwickler spannendere Frage übrig: Welche Techniken gibt es in der Softwareentwicklung, die helfen, die Komplexität des entwickelten Codes zu reduzieren? Die Frage stellt sich zwar

nur, wenn sich der Entwickler überhaupt für die Qualität der erstellten Software verantwortlich fühlt, statt das der QS-Abteilung zu überlassen. An der Stelle ist aber glücklicherweise in letzter Zeit ein starkes Umdenken in Gang gekommen, gerade auch in Zusammenhang mit dem Einzug agiler Methoden in die Softwareentwicklung. Entsprechend ist es weit verbreitete Praxis, dass Entwickler Tests schreiben, bevor sie den getesteten Code dazu implementieren (bekannt als „Test-Driven Development“ [3]; TDD). Tatsächlich kann dieses helfen, Fehler zu vermeiden. Denn durch TDD wird der Entwickler gezwungen, die Sicht des Testers einzunehmen, bevor er die Funktion „konstruktiv“ betrachtet, um sie zu implementieren.

Das verhindert eine andere Fehlerquelle, die beim nachträglichen Testen durch einen Entwickler immer eine Gefahr ist: den Bestätigungsfehler (auch „Confirmation Bias“). Er entsteht durch die menschliche Neigung, die Realität mit zuvor gestellten eigenen Erwartungen möglichst in Einklang zu bringen. Im Fall der klassischen Softwareentwicklung mit nachgelagertem Testen bedeutet das: Der Entwickler implementiert zunächst die Funktion, wobei er seine eigenen Erwartungen umsetzt und festigt, und danach testet er die Implementierung gegen genau diese Erwartungen. Der Confirmation Bias sorgt so beim Testen dafür, dass er unbewusst gar keine Fehler finden will.

Der klassische nachgelagerte Softwaretest kennt diese Fehlerquelle natürlich auch, weswegen dort zwischen Testern und Entwicklern meist strikt getrennt wird. Man sollte sich aber bewusst machen, dass dabei leicht über das Ziel hinaus geschossen wird, wenn von einem speziellen Tester-„Mindset“ die Rede ist, das ein Entwickler gar nicht haben könne. Ein entsprechend in Testmethoden ausgebildeter Entwickler ist durchaus in der Lage, kritisch zu testen – solange er es im Sinne von TDD tut, bevor er sich an die Implementierung setzt.

So unabdingbar das Testen für die Softwarequalität ist, und so sinnvoll TDD in dem Zusammenhang ist, bleibt jedoch ein Nachteil: Man findet Fehler immer erst nach deren Entstehen. Es ist in diesem Sinne also von der Zielsetzung her immer noch keine echte Vorsorge.

Kognitive Entlastung durch Lesbarkeit

Was kann der qualitätsbewusste Entwickler also tun, um von vornherein die Code-Komplexität zu reduzieren? Hierzu kann er sich die Frage stellen, womit er beim Implementieren die meiste Zeit verbringt. Im Normalfall dürfte das das Lesen (und nicht etwa das Schreiben) von Quelltext sein. Das bedeutet, dass Entwickler ihre kognitive Belastung am besten entlasten können, wenn sie Code so schreiben, dass er möglichst leicht lesbar ist (für ihn selbst und für die Kollegen im Team). Diese Erkenntnis ist Kernthese der „Clean Code“-Bewegung. Tatsächlich sind alle (teils provokanten) Thesen aus Robert C. Martins Buch zu Clean Code [4] darauf ausgerichtet, die Lesbarkeit und Verständlichkeit von Quelltext zu erhöhen.

Es ist alles andere als neu, dass kurze Klassen und Methoden besser verständlich und deswegen weniger fehleranfällig sind als lange oder tief verschachtelte. Hierfür gibt es klassische Metriken wie McCabes zyklomatische Komplexität, die sich auch im Rahmen einer statischen Codeanalyse automatisch prüfen

lässt. Aber dann ist es bereits wieder zu spät für die Vorsorge. Es ist auch nicht neu, dass sprechende Benennungen helfen, Code verständlicher zu machen. Die Innovation bei Clean Code liegt eher in der Forderung nach kompromissloser Entwicklerdisziplin, den eigenen Code sauber zu halten.

Ein einfacher Selbstversuch mit Clean Code kann eindrücklich aufzeigen, wie sehr es entlastet, an lesbarem Code zu arbeiten. Er verdeutlicht, dass es tatsächlich viel Disziplin erfordert, Code von vornherein lesbar zu gestalten. Und er zeigt, dass sich die Mühe auszahlt, die man beispielsweise in die möglichst passende Benennung seiner Klassen und Methoden steckt – nicht zuletzt stellt man fest, dass die Wahl sprechender Namen direkt das gesamte Systemdesign verbessern kann: Wenn der Entwickler eine Klasse einführen möchte, aber keinen guten Namen für sie findet, ist das im Normalfall kein Zeichen für fehlende Kreativität, sondern eines dafür, dass am Domänenmodell etwas noch nicht ganz stimmt.

Mit der Philosophie von „Clean Code“ wurde also ein wichtiger Baustein für Fehlerprävention identifiziert. Die Lesbarkeit oder äußere Form von Quelltext ist aber noch lange nicht alles, was zu dessen Komplexität beiträgt. Mindestens ebenso wichtig ist, auf welche Art und Weise der Code das tut, was er tun soll.

Das Typsystem – Freund und Helfer

Ein oft unterschätztes Hilfsmittel ist das Typsystem: Es ist ja gerade Aufgabe eines statischen Typsystems, Fehler bereits für den Compiler auffindbar zu machen, die sonst erst zur Laufzeit auffallen würden, sobald man den fehlerhaften Programmzweig durchläuft. In diesem Sinne ist das Typsystem einer Programmiersprache ihr wichtigstes Bordmittel zur Fehlervermeidung. Leider hat das seinen Preis: Es zwingt einen dazu, an vielen Stellen die Typen explizit hinzuschreiben, was unbequem sein und auch Boilerplate-Code erzeugen kann (also Code, der nur für den Compiler zu schreiben ist, für den Entwickler aber überflüssigen Ballast darstellt). Das lässt sich zwar je nach Programmiersprache mehr oder weniger gut mit Typinferenz abmildern, aber im Normalfall nicht ganz verhindern.

Damit es sich im Sinn einer Effizienzsteigerung beim Entwickeln lohnt, ein Typsystem einzusetzen, muss es genügend mächtig sein, dass es nicht nur triviale Fehler findet. Es muss einen in die Lage versetzen, signifikant Anwendungslogik mit domänenspezifischen Typen zu unterstützen, und zwar so, dass tatsächlich unzulässige Zustände im zu lösenden Problemraum zu Compiler-Fehlern führen.

Ein einfaches Beispiel für Situationen, in denen es sich fast immer lohnt, auf das Typsystem zurückzugreifen, sind Sonderfallbehandlungen. Und der wohl häufigste anzutreffende in der Entwicklung dürfte der *null*-Sonderfall sein, von dem Tony Hoare, der „Erfinder“ der Nullreferenz, inzwischen selbst sagt, es sei sein „billion-dollar mistake“ gewesen [b].

Vom Umgang mit Nullreferenzen

Es dürfte heute wohl kaum einen Entwickler geben, der sich nicht schon mit Nullreferenzen und den zugehörigen Laufzeitfehlern herumgeschlagen hat, ob sie nun *NullPointerException* heißen oder *NullReferenceException*. Das Grundproblem ist, dass Datentypen in den gängigen Programmiersprachen so gut wie immer zusätzlich zu ihrem „normalen“ Wertebereich noch einen Nullwert zur Verfügung stellen. Bei Verwendung also von Variablen solcher Datentypen muss sich der Entwickler grundsätzlich immer darum kümmern, beide Fälle zu behandeln.

Bei einem Newsletter als Beispiel, der an alle Personen geschickt werden soll, die über ein Webformular neben ihrem Namen ihre E-Mail-Adresse angegeben haben, sieht die Datenstruktur (etwa in Scala) naiv wie in Listing 1 aus, und das dazugehörige Versenden des Newsletters erfordert eine Abfrage auf *null*. Das Beispiel ist noch einfach genug, dass es keine Probleme bereitet, an die Abfrage auf *null* zu denken. In komplexen Programmen entstehen aber häufig Fehler genau deshalb, weil man so eine Abfrage vergessen hat. Deswegen gehen viele Entwickler dazu über, in ihren Programmen überall sicherheitshalber Nullprüfungen vorzunehmen, ob sie notwendig sind oder nicht. Dieser defensive Programmierstil verhindert zwar dann genau diese Fehlerquelle, allerdings werden dadurch die Programme deutlich unübersichtlicher und unlesbarer als nötig, was wiederum eine eigene Fehlerquelle ist.

Die Alternative ist hier, die Option der E-Mail-Adresse mit dem Typsystem zu adressieren. Dafür gibt es ein allgemeines Rezept in Form des *Option*-Datentyps. So ein Summentyp kann entweder leer sein (hat also den Untertyp *None*) oder enthält wie in Listing 2 einen String (also den Untertyp *Some[String]*. Der Trick ist hier, dass die *Option* den Entwickler zwingt, an die Sonderbehandlung für den Fall einer nicht vorhandenen E-Mail-Adresse zu denken.

Listing 1: Newsletter versenden mit Nullprüfung

```
case class Person(name: String, email: String)

trait Newsletter {
  val persons: List[Person]

  def sendNewsletterTo(email: String): Unit

  def sendNewsletter: Unit = {
    for (person <- persons)
      if (person.email != null)
        sendNewsletterTo(person.email)
  }
}

object Main extends App {
  val newsletter = new Newsletter {
    val persons = List(
      Person("Bill", "bill@somewhere.de"),
      Person("Bob", null))

    def sendNewsletterTo(email: String) =
      println(email)
  }

  newsletter.sendNewsletter
}
```

Listing 2: Newsletter versenden mit Option-Datentyp

```
case class Person(name: String, email: Option[String])

trait Newsletter {
  val persons: List[Person]

  def sendNewsletterTo(email: String): Unit

  def sendNewsletter: Unit = {
    for (person <- persons)
      person.email foreach sendNewsletterTo
  }
}

object Main extends App {
  val newsletter = new Newsletter {
    val persons = List(
      Person("Bill", Some("bill@somewhere.de")),
      Person("Bob", None))

    def sendNewsletterTo(email: String) =
      println(email)
  }

  newsletter.sendNewsletter
}
```

Wer Entwickler allerdings zu etwas zwingt, sollte tunlichst dafür sorgen, dass es trotzdem bequem bleibt, den *None*-Sonderfall zu behandeln. Und das ist es glücklicherweise in Sprachen wie Scala und F#, die einen *Option*-Datentyp enthalten, der sich dank Funktionen höherer Ordnung (wie *map* und *flatMap*) angenehm in seine Umgebung integriert. In Java (zumindest vor Java 8) ist das leider nicht der Fall, was wohl jeder schon erlebt hat, der beispielsweise mit *Optional* aus der Guava-Bibliothek arbeitet.

Aber selbst in Java gibt es einen Ausweg, den Spezialfall mit dem Typsystem abzufangen, ohne sich zu verrenken. Ganz klassisch objektorientiert kann der Entwickler die Personen per Subclassing in Personen mit und ohne E-Mail-Adresse unterteilen. Das Versenden der Newsletter-E-Mail erledigt dann die jeweilige Unterklasse via *Dynamic Dispatch* (s. Listing 3).

Dadurch lassen sich nicht nur Nullprüfungen mit dem Typsystem abfangen, sondern auch viele andere Sonderfälle. Das ist für die meisten nichts Neues. In der Praxis ist allerdings leider häufig zu beobachten, dass stattdessen *if*-Abfragen verwendet werden, die zu später entdeckten Laufzeitfehlern und aufwendiger Fehleranalyse führen, wo schon der Compiler einen Fehler hätte melden können.

Listing 3: Newsletter versenden via Dynamic Dispatch

```
sealed trait Person {
  def name: String
  def sendNewsletter(newsletter: Newsletter): Unit
}

case class PersonWithEmail(name: String, email: String)
    extends Person {

  def sendNewsletter(newsletter: Newsletter) =
    newsletter.sendNewsletterTo(email)
}

case class PersonWithoutEmail(name: String)
    extends Person {

  def sendNewsletter(newsletter: Newsletter) = {
    /* nothing to do */
  }
}

trait Newsletter {
  val persons: List[Person]

  def sendNewsletterTo(email: String): Unit

  def sendNewsletter: Unit =
    for (person <- persons)
      person.sendNewsletter(this)
}

object Main extends App {
  val newsletter = new Newsletter {
    val persons = List(
      PersonWithEmail("Bill", "bill@somewhere.de"),
      PersonWithoutEmail("Bob"))
    def sendNewsletterTo(email: String) =
      println(email)
  }

  newsletter.sendNewsletter
}
```

Komplexität durch Redundanz

Eine andere wesentliche Erscheinungsform überflüssiger Komplexität – bei der der Compiler allerdings nichts hilft – ist Komplexität durch Redundanz. Es gibt viele Arten von Redundanz, aber die meisten entstehen beim Programmieren durch Copy & Paste. Code wird kopiert, weil es bequem ist und schnell geht. Was allerdings oft vernachlässigt wird, ist, wie sich das auf die Wartbarkeit auswirkt. Eine eigentlich einfache Änderung kann bei zu viel kopiertem Quelltext zu enormem Aufwand führen: Plötzlich ist die Änderung an zig Stellen einzupflegen, und der Entwickler weiß hinterher nicht einmal mit Sicherheit, ob er an alle solchen Stellen gedacht hat. Und genau auf diese Weise entsteht aus Redundanz dann Komplexität – wieder muss man beim Entwickeln Seiteneffekte im Blick behalten („Wo muss ich das noch überall ändern?“), was das Arbeitsgedächtnis unnötig belastet.

Die Regel, Code nicht zu kopieren und stattdessen zu abstrahieren, sobald er an zwei oder mehr Stellen benötigt wird, hört sich allerdings leider einfacher einhaltbar an, als sie es in der Praxis zu sein scheint. Oft unterscheiden sich Codestellen in ein paar kleinen Details, sodass man nicht mehr einfach nur eine Unterfunktion ausgliedern kann. In Listing 4 sieht der Leser als (etwas konstruiertes) Beispiel die Ergebnisverarbeitung zweier solchermaßen „ähnlicher“ Datenbankaufrufe. Das sieht nach viel dupliziertem Code aus, aber was tun? Im hektischen Arbeitsalltag erlebt man oft, dass an dieser Stelle aufgegeben wird: zu kompliziert, zu unbequem. Je nach Programmierparadigma gibt es aber zwei einfache Lösungsansätze: das Template Method Pattern (objektorientiert) und Higher-Order Functions (funktional).

Der erste Ansatz erfordert eine abstrakte Basisklasse, in der der kopierte Code als Schablone angelegt ist. Die variierenden Codestellen bleiben abstrakt (s. Listing 5). „Higher-Order Functions“ erfordern statt der Vererbungshierarchie, dass die aufgerufene „kopierte“ Funktion die variierenden Stellen als Funktionen mit hineingereicht bekommt, die sie dann passend aufruft (s. Listing 6). Der Ansatz funktioniert nur gut, wenn die Sprache Lambda-Ausdrücke gut unterstützt, was zum Glück bei den meisten modernen Sprachen der Fall ist. Aber egal wie man es genau macht, in jedem Fall ist es alles andere als ein Ding der Unmöglichkeit, die Quelltext-Kopiererei zu vermeiden.

Listing 4: Ein klassischer Fall von Copy & Paste

```
def getNumPersons(query: PreparedStatement): Long = {
  val results = query.executeQuery
  try {
    results.next
    return results.getLong("numPersons")
  } finally {
    results.close
  }
}

def getNumPersonsNamed(
    name: String, query: PreparedStatement): Int = {
  query.setString(1, name)
  val results = query.executeQuery
  try {
    results.next
    return results.getInt("numNamedPersons")
  } finally {
    results.close
  }
}
```

Komplexität durch Zustand

Zu guter Letzt noch ein weiterer Aspekt, der starken Einfluss auf die Komplexität beim Entwickeln hat: der veränderliche Zustand. Das zustandsorientierte Programmieren stammt aus einer Zeit, in der es absolut notwendig war, in einzelnen Speicherzellen zu denken und diese direkt zu ändern. Das hat sich inzwischen allerdings stark gewandelt. In der Regel ist Speicherplatz inzwischen nicht mehr das Hauptproblem. An seine Stelle treten langsam, aber sicher Themen wie die Parallelisierung. Die Sicht auf das Programmieren ist aber immer noch sehr bestimmt von Variablen, die ständig zu ändern sind [c].

Veränderlicher Zustand erzeugt aber Komplexität. Dem menschlichen Gehirn fällt es im Allgemeinen schwer, von außen nachzuvollziehen, in welchem Zustand sich ein System befindet. Das merkt man insbesondere dann, wenn man Objekte mit kom-

Listing 5: Copy & Paste-Vermeidung per Template Method Pattern

```
abstract class NumPersons[T] {
  final def get(query: PreparedStatement): T = {
    initializeQueryParameters
    val results = query.executeQuery
    try {
      results.next
      return valueOf(results)
    } finally {
      results.close
    }
  }


  // abstract
  def initQueryParams(query: PreparedStatement): Unit

  // abstract
  def valueOf(results: ResultSet): T
}

class NumAllPersons extends NumPersons[Long] {
  def initQueryParams(query: PreparedStatement) = {
    /* nothing to do */
  }
  def valueOf(results: ResultSet): Long =
    results.getLong("numPersons")
}

class NumNamedPersons(name: String)
    extends NumPersons[Int] {

  def initQueryParams(query: PreparedStatement) =
    query.setString(1, name)
  def valueOf(results: ResultSet): Int =
    results.getInt("numNamedPersons")
}
```

plexem inneren Zustand systematisch testen möchte: Ein paar einfache Beispiele und Spezialfälle reichen plötzlich als Testfälle bei weitem nicht mehr aus. Stattdessen ist aufwendig der gesamte Zustandsgraph des Objekts abzudecken. Viel einfacher dagegen ist das Testen von Funktionen, die keine Seiteneffekte haben: Sie liefern für gleiche Eingabe- immer gleiche Ausgabewerte. So fällt auch das Nachvollziehen solcher Funktionen deutlich leichter.

Es ist von Vorteil, möglichst weite Teile des Quelltexts zustandsfrei zu halten, um die Komplexität zu senken. Das ist in der Theorie allerdings leichter gesagt, als in der Praxis getan. Ein Zustand ist in der Programmierung überall zwangsläufig erforderlich, wo Ein- und Ausgabe passiert: etwa bei Events aus einer Bedienoberfläche oder beim Zugriff auf Datenbanken.

Zum Glück schaffen hier moderne Programmierkonzepte Abhilfe. Bei Events kann der Entwickler heute ohne Callbacks auskommen und veränderlichen Zustand vermeiden, wenn er reaktiv arbeitet, wie es beispielsweise in der .NET-Welt via Rx („Reactive Extensions“) [d] möglich ist. Auch Datenbanken gehen zunehmend in eine „ereignisorientierte“ Richtung, wo Daten historisiert werden, statt direkt Änderungen daran vorzunehmen.

Onlinequellen

[a] Pomodoro
www.pomodorotechnique.com

[b] Tony Hoare; Null References:
The Billion Dollar Mistake (Vortragsmitschnitt)
www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[c] Rich Hickey; The Value of Values (Vortragsmitschnitt)
www.infoq.com/presentations/Value-Values

[d] Reactive Extensions (Rx)
msdn.microsoft.com/en-us/data/gg577609

[e] Rich Hickey; Are We There Yet? (Vortragsmitschnitt)
www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey

Listing 6: Copy & Paste-Vermeidung per Higher-Order Functions

```
def getNumAllPersons(query: PreparedStatement): Long =
  getNumPersons(query,
    _ => (),
    _.getLong("numPersons"))

def getNumPersonsNamed(
    name: String, query: PreparedStatement): Int =
  getNumPersons(query,
    _.setString(1, name),
    _.getInt("numNamedPersons"))

def getNumPersons[T](query: PreparedStatement,
    initQueryParams: PreparedStatement => Unit,
    valueOf: ResultSet => T): T = {
  initQueryParams(query)
  try {
    results.next
    return valueOf(results)
  } finally {
    results.close
  }
}
```

Das Grundprinzip ist dabei immer, Daten an einem Zeitstrahl anzuordnen, statt sie zu ändern [e]. Eine Änderung ist in diesem Sinn ein Ereignis auf dem Zeitstrahl, ab dem dann andere Fakten vorhanden sind als vorher. Die vorherigen Fakten sind aber unverändert vorhanden und abfragbar. Datenstrukturen, die diesen Mechanismus effizient unterstützen, nennt man deshalb auch persistente Datenstrukturen. Ihre Speichereffizienz ist besser, als man auf den ersten Blick vielleicht meinen könnte, insbesondere auch im Hinblick auf Szenarien mit Parallelität, wo die Alternative direkter Änderungen schnell dazu führt, dass man viele defensive Kopien vorhalten muss. Und die Komplexität, die sonst durch unkontrolliert veränderlichen Zustand entsteht, kann man dank dieser Datenstrukturen so gut wie vollständig loswerden.

Fazit

Es gibt mannigfaltige Techniken, Komplexität zu reduzieren. Sie sind im Grunde alle nicht neu. Setzt man sie aber konsequent um und behält auch sonst immer im Blick, wo Komplexität notwendig und wo sie überflüssig ist, haben Programmierer die Chance, einen großen Teil der kognitiven Last beim Entwickeln loszuwerden. Diese Entlastung wiederum hilft nicht nur, den Stress beim Entwickeln zu reduzieren, sondern auch tatsächlich Fehler von vornherein zu vermeiden. Die Tester werden es einem genauso danken wie die Mitentwickler. (ane)

Literatur

[1] George A. Miller; The magical number seven plus or minus two; Some limits on our capacity for processing informations; Psychological Review 63 8 (1956), S. 81–97

[2] David Allen; Getting Things Done; Piatkus 2002

[3] Kent Beck; Test Driven Development By Example; Addison-Wesley 2002

[4] Robert C. Martin; Clean Code; A Handbook of Agile Software Craftsmanship; Prentice Hall 2008

Joachim Hofer
studierte Informatik an der Friedrich-Alexander-Universität Erlangen-Nürnberg. Bei der imbus AG leitet er das TestBench-Entwicklungsteam.

Alle Links: www.ix.de/ix1315140