DEKONSTRUKTION: VON INTEGRATIONS- ZU UNIT-TESTS

Dieser Artikel nimmt das Verhältnis zwischen leichtgewichtigen Tests (Unit-Tests) und schwergewichtigen Tests (Integrations- oder Systemtests) in den Blick. In der Entwicklungspraxis entsteht oft ein Übergewicht der letzteren, obwohl es eigentlich andersherum sein sollte. Woran liegt das?

die autoren

Dr. Andreas Arnold
(andreas.arnold@andrena.de)
ist Softwareentwickler für Java und .NET bei der andrena objects ag in Karlsruhe. Besondere Erfahrungen hat er mit der Migration alter Anwendungen auf neue Technologien. Er interessiert sich für die automatisierte Messung von Softwarequalität.

Leif Frenzel
(leif@andrena.de)
ist Softwareentwickler und agiler Coach bei der andrena objects ag. Er arbeitet seit mehr als zehn Jahren mit XP-Praktiken und interessiert sich für Aspekte der sauberen, nachhaltigen und verantwortungsbewussten Softwareentwicklung.

In der agilen Softwareentwicklungs-Community herrscht weitgehend Einigkeit darüber, dass konsequentes und flächendeckendes Unit-Testen das Fundament automatisierter Tests bilden und erst zusätzlich durch Integrationstests zwischen Komponenten flankiert werden sollte. In der theoretischen Diskussion sind automatisierte Tests nach vielfältigen Gesichtspunkten klassifiziert worden; für den praktischen Gesichtspunkt, der hier im Vordergrund steht, ist der relevante Unterschied der zwischen leichtgewichtigen und schwergewichtigen Tests.

Schwergewichtige Tests *laufen oft zu lange*, um direkt während der Arbeit in einer Entwicklungsumgebung ausgeführt zu werden. Da sie unterschiedliche Stellen der Codebasis betreffen, sind sie oft *spröde*, müssen also häufig an Änderungen im Produktivcode angepasst werden, ohne dass neuer Testnutzen entsteht. Außerdem sind *Rückschlüsse auf Fehler* bei fehlschlagenden Tests vielfach schwieriger als bei leichtgewichtigen Tests.
Ein weniger offensichtlicher, aber gravierender Nachteil schwergewichtiger Tests ist, dass sie die *aktuelle Modulstruktur zementieren*: Da sie einen Großteil des Systems auffahren müssen, machen sie häufig Voraussetzungen über die Verteilung einzelner Klassen auf Module. Bei größeren Systemen führt dies zu einer höheren Hemmschwelle, Module neu zu organisieren und Funktionalität (einschließlich seiner Unit-Tests) zwischen Modulen zu bewegen.

Kasten 1: *Nachteile von schwergewichtigen Tests.*

Leicht- und schwergewichtige Tests

Leichtgewichtige Tests sind aufgrund ihrer sehr kurzen Laufzeit (wenige Sekunden für mehrere hundert Testfälle) schnell ausführbar. Sie können in wenigen Codezeilen formuliert werden und erfordern insbesondere keine langwierigen Setups. Sie sind auf spezifische Codestellen ausgerichtet und daher im Fall des Fehlschlagens leicht zu interpretieren. Der paradigmatische Fall sind Unit-Tests, die kurze und isolierte Aussagen über einzelne Methoden, Eigenschaften oder das Verhalten einer Klasse (oder ein bestimmtes Zusammenspiel weniger Klassen) treffen und diese am Produktivcode überprüfen.

Ihnen gegenüber stehen *schwergewichtige Tests* (siehe Kasten 1), paradigmatisch Integrationstests, die auf die Integration einzelner Komponenten miteinander zielen, oder Systemtests, bei denen die Integration mit Drittsystemen im Blickpunkt steht. Solche Tests starten eine signifikante Portion des Systems oder gar das ganze System (gegebenenfalls sogar Instanzen von Drittsystemen). Weniger einfach als schwergewichtig zu erkennen sind solche, die komplexe Initialisierungen erfordern und dafür auf aufwändige Konstruktionen wie In-Memory-Datenbanken (oft für Persistenztests verwendet) oder in Mocks „nachgebaute" Aufrufsequenzen auf Abhängigkeiten (üblicherweise Service-Objekte) zurückgreifen. Sie teilen jedoch oft die Sprödigkeit, Unübersichtlichkeit, und gelegentlich auch die verhältnismäßig lange Laufzeit mit den erwähnten Integrationstests, die produktive Komponenten oder Drittsystem-Instanzen als Testumgebung auffahren[1]).

[1]) *Angelehnt an den Sprachgebrauch in der Praxis verwenden wir im Folgenden gelegentlich die Begriffe für die paradigmatischen Fälle, „Unit-Test" und „Integrationstest", beispielhaft für die allgemeineren Klassen leicht- bzw. schwergewichtiger Tests.*

Ideal und Alltagswirklichkeit

Während leichtgewichtige Tests nahezu flächendeckend für die gesamte Codebasis vorhanden sein sollten und idealerweise auch kontinuierlich entwicklungsbegleitend ausgeführt werden (sowohl in der Entwicklungsumgebung als auch im *Continuous Integration-Server*), gilt für schwergewichtige Tests eher der Grundsatz: „so viel wie nötig, aber so wenig wie möglich." Als quantitative Faustregel dafür ist die Testautomatisierungs-Pyramide (vgl. [Cohn] und siehe Abbildung 1) bekannt: Die breite Basis der Pyramide steht für eine hohe Abdeckung mit Unit-Tests, die höheren Schichten bis zur Spitze (Integrations-, System-, und manuelle Tests) werden graduell schmaler und symbolisieren so eine geringere Abdeckung, ein schrittweises Abrücken von einem Vollständigkeitsanspruch und einen immer stärkeren stichprobenartigen Charakter.

In der Praxis lässt sich jedoch oft genau der gegenläufige Trend beobachten. Die

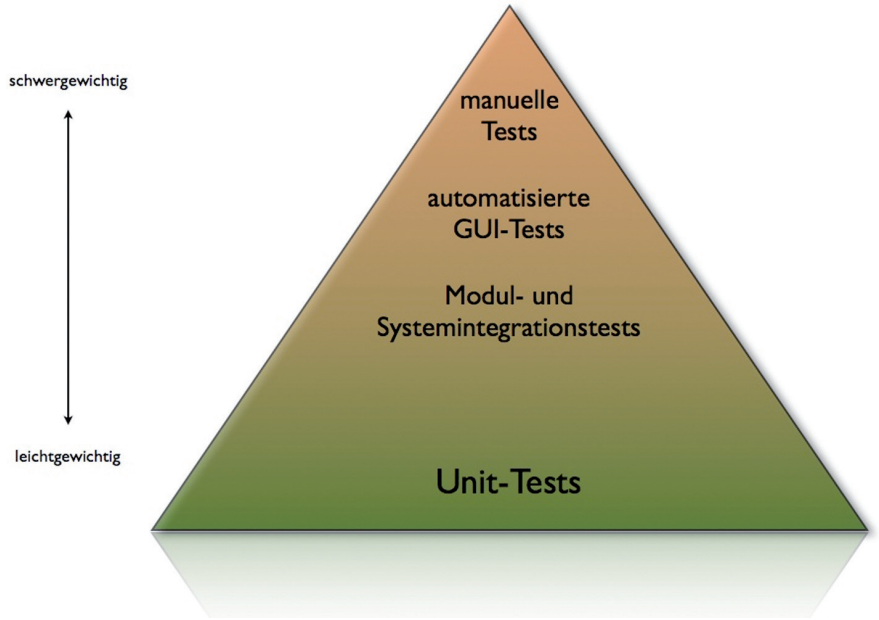

Abb. 1: Die Testautomatisierungs-Pyramide nach Cohn.

Pyramide steht Kopf, wie es gelegentlich zu hören und lesen ist: Viel Zeit und Arbeit wird in aufwändige Test-Setups und schwergewichtige Tests investiert, die ganze Komponenten (vielfach sogar das gesamte System und Testinstanzen von Drittsystemen) hochfahren und ansprechen, um einen Testfall auszuführen. Das quantitative Übergewicht in der Zahl der Testklassen und -methoden, Codezeilen oder hinsichtlich von Abdeckungskriterien liegt bei schwergewichtigen Tests. Nach unseren Beobachtungen aus der Beratungstätigkeit kann man Entwickler außerdem einfacher und stärker motivieren, schwergewichtige automatisierte Tests zu entwerfen und zu implementieren, als durchgängig entwicklungsbegleitend Unit-Tests zu schreiben.

Überdies gilt auch für das Verhältnis zwischen leicht- und schwergewichtigen Tests, was für so vieles in der Softwareentwicklung gilt: Es entwickelt sich *dynamisch* während der Lebenszeit einer Codebasis. Auf die eben angeführte Beobachtung bezogen bedeutet dies, dass oft nicht nur ein Übergewicht schwergewichtiger Tests vorzufinden ist, sondern dass sich dieses Übergewicht im Laufe der Zeit noch verstärkt: Mit dem Wachstum der Codebasis sollte die Basis an leichtgewichtigen Tests eigentlich stärker wachsen als die Basis schwergewichtiger Tests; das Gegenteil scheint jedoch der Fall zu sein. Was sind die Gründe für diese Entwicklung?

Technische und andere Faktoren

Wenn es auch auf den ersten Blick überraschen mag: Es handelt sich nicht nur um simple technische Faktoren. Natürlich gibt es gute Gründe, zumindest einen Grundstock an schwergewichtigen Tests zu haben. Bei der Sanierung von Altsystemen, vor allem wenn sie ursprünglich ohne automatisierte Tests entstanden sind, sind sie oft ein sinnvoller oder sogar notwendiger erster Schritt zur Grundabsicherung von Umarbeiten. Und natürlich erfordert jedes umfangreiche System die stichprobenartige Sicherstellung der Integration von Komponenten, Dritt-Frameworks und externen Systemen.

Schwergewichtige Tests an sich sind also nicht das Problem – das ist vielmehr das Übergewicht, das sich im Laufe der Zeit bildet. Die Ursachen sind vielfältig:

- Schwergewichtige Tests, die zunächst zur Grundabsicherung von Umbauten oder Sanierungen geschrieben wurden, werden im zweiten Schritt nicht durch leichtgewichtige Tests ergänzt oder ersetzt.
- Schwer testbarer Code wird nicht umgearbeitet, sondern lediglich von außen mit schwergewichtigen Tests versehen und damit als hinreichend getestet betrachtet.
- Neu zu entwickelnder Code, dessen Feinstruktur bei Beginn der Implementierung noch nicht feststeht, wird nicht testgetrieben entwickelt, sondern nach der Implementierung – wiederum von außen – noch um Tests ergänzt.

Ein häufiges Motiv in allen diesen Fällen ist, dass Änderungen am schon bestehenden Produktivcode vermieden werden können. Handelt es sich um ein bereits umfangreiches und komplexes System, das möglicherweise schon mit technischen Schulden belastet ist, dann ist das ein Anreiz zur wahrgenommenen Risikovermeidung: Man kann automatisierte Tests schreiben und damit die Qualitätssicherung verbessern, läuft jedoch nicht Gefahr, durch ungewollte Effekte der Änderung neue Fehler zu produzieren. Wird diese Vorsicht zur Methode, so kann sich über längere Zeiträume hinweg unbemerkt eine defensive Präferenz für schwergewichtige Tests etablieren.

Schwergewichtige Tests suggerieren eine höhere Testabdeckung. Das betrifft nicht nur die messbare Überdeckung von Codezeilen oder -zweigen beim Testlauf, sondern auch die intuitive Sicht des Entwicklers. Eine Aussage, die sich auf eine spezifische Stelle im Code bezieht, wird als Aussage über das Verhalten einer Komponente oder gar des Gesamtsystems formuliert. Ein häufig angeführtes Argument für dieses Vorgehen ist der Gedanke: „Dann teste ich gleich noch x, y, und z mit." So entsteht oft ein Test, der sich auf eine bestimmte Codestelle bezieht (beispielsweise eine einzelne Berechnung in der Geschäftslogik), diese Codestelle aber über einen längeren Weg anspricht (via externer Schnittstelle über interne Services bis hin zum eigentlichen Berechnungscode). Das suggeriert einen höheren Wert des geschriebenen Tests, nämlich mehr getestete Funktionalität, bei vergleichbarem Implementierungsaufwand gegenüber einem Unit-Test. Dieser Gedanke hat eine hohe momentane Plausibilität; die Nachteile, die sich aus der Schwergewichtigkeit des einzelnen Tests und dem kumulativen Effekt vieler ähnlicher Entscheidungen ergeben, werden häufig erst später deutlich und gehen deshalb nicht in die Abwägung mit ein.

Weitere motivatorische Faktoren im Projektalltag können eine Rolle spielen. Integrations- und Systemtests erfordern oft projektinterne technische Arbeiten (z. B. das Aufbauen einer dedizierten Testinfrastruktur), die eine willkommene Abwechslung bieten mögen, das Ausprobieren neuer Technologien oder Trends ermöglichen und überdies als technische Arbeiten

unter der Hoheit des Entwicklungsteams bleiben und daher keine Abstimmung oder Auseinandersetzung mit den Auftraggebern erfordern. Je nach Projektsituation mag dies eine attraktive Erhöhung des Spaßfaktors bedeuten und die Entscheidung für aufwändige Testmechanismen begünstigen.

Eine Kombination solcher Motive führt zu der beobachteten Tendenz, dass schwergewichtige automatisierte Tests über das erforderliche Maß hinweg entstehen. Im Laufe der Zeit verschiebt sich so das Verhältnis von leicht- zu schwergewichtigen Tests häufig unbemerkt in eine unerwünschte Richtung.

Dekonstruktion: Integrations- zu Unit-Tests

Hier ist also ein aktives Entgegenwirken gefragt. Es besteht weitgehend Einigkeit darüber, dass Refactoring und Code-Hygiene ein beständiges Element in der Wartung und Weiterentwicklung einer Codebasis sein sollten – unserer Meinung nach muss dies ein kontinuierliches Dekonstruieren von schwergewichtigen Tests einschließen. Ein solches Dekonstruieren umfasst die folgenden Schritte:

- Das Verhältnis zwischen schwer- und leichtgewichtigen Tests wird regelmäßig betrachtet und korrigiert.
- Integrationstests werden daraufhin untersucht, ob sie mehr tun, als nur einen Integrationspunkt abzusichern – falls dies der Fall ist, werden Unit-Tests für diejenigen Codestellen geschrieben, die vom Integrationstest mitgetestet werden.
- Anschließend wird der Integrationstest darauf reduziert, wirklich nur noch sicherzustellen, dass der Integrationspunkt korrekt bedient wird (d. h. er wird zu einer bloßen Stichprobe).

Dekonstruktion bedeutet – im Unterschied zu Destruktion – kein völliges Verwerfen, sondern ein Zerteilen in funktionsfähige Bestandteile: In diesem Fall sind das Unit-Tests und (wenige) Tests für Integrationspunkte, die in Summe den früheren schwergewichtigen Test ersetzen. Das Ergebnis ist eine positive Veränderung des Verhältnisses von schwer- zu leichtgewichtigen Tests zugunsten letzterer.

Dekonstruktion in der Praxis

Der geeignete Pfad von Integrations- zu Unit-Tests ist freilich im Programmieralltag nicht unbedingt einfach zu beschreiten – und auch nicht notwendigerweise immer offensichtlich. Wie in vielen Aspekten der Entwicklungspraxis müssen Teams hier ihren eigenen Weg finden. Leider kann man das Ziel nicht mit einfachen technischen Tricks erreichen, auch der Entwicklungsprozess muss angepasst werden. Zu diesen beiden Aspekten möchten wir im weiteren Verlauf des Artikels noch einige Hilfestellungen geben.

```
public IList<Entity> ReadObjectList(string sqlQuery)
{
  var result = new List<Entity>();
  using (var command = connection.CreateCommand())
  {
    connection.Open();
    using (var command = connection.CreateCommand())
    {
      command.CommandText = sqlQuery;
      using (var dataReader = command.ExecuteReader())
      {
        if (dataReader.HasRows)
        {
          OracleDataReader reader = dbReader as OracleDataReader;
          while ( reader.Read() )
          {
            var item = new EndurDealEntity();
            item.Property = reader.GetString(0);
            result.Add(item);
          }
        }
        dataReader.Close();
      }
    }
  }
  return result;
}
```

Listing 1: Schlecht testbare Implementierung eines Datenlesers für Objekte aus der Datenbank. Es ist unmöglich, das Einlesen der Daten vom Umwandeln in Objekte getrennt zu testen.

Schwer testbarer Code

Am eher technischen Ende des Spektrums gilt es vor allem, einen Blick dafür zu entwickeln, wie solche Codestellen leichtgewichtig getestet werden können, die auf den ersten Blick eher schwergewichtige Tests nahelegen. Viele der notwendigen Fertigkeiten liegen im Bereich des „Refactoring zur besseren Testbarkeit". Langfristig erhöht sich dadurch nicht nur die Testbarkeit, sondern auch die Wartbarkeit, sodass dieser Schritt auch aus anderen Gründen zu empfehlen ist. Natürlich kann man nicht beliebig schnell die Qualität steigern, aber auf jeden Fall sollte man schnellstmöglich damit beginnen.

Es gibt viele Gründe für schlechte Testbarkeit. Sehr schwer testbar ist z. B. die GUI. Deshalb sollte man darauf achten, dass in ihr keinerlei Berechnungslogik steckt, sondern wirklich nur die Anzeige. Berechnungen müssen in andere Klassen ausgelagert werden – und schon sind sie testbar. Da der Zugriff auf Hardware nur in Integrationstests möglich ist, sollte die

```
public class Database
{
  public DataTable ReadDataTable(string sqlQuery, string
tableName)
  {
    var result = new DataTable(tableName);
    using (var connection = GetConnection())
    {
      connection.Open();
      using (var adapter =
_factory.CreateDataAdapter())
      {
        using (var command =
_factory.CreateCommand())
        {
          adapter.SelectCommand = command;
          command.CommandText = sqlQuery;
          command.Connection = connection;
          adapter.Fill(result);
        }
      }
    }
    return result;
  }
}
public class DatenleserBesserTestbar
{
  public Database Database { get; set; }
  public IList<Entity> ReadObjectList(string sqlQuery,
string tableName)
  {
    var table = Database.ReadDataTable(sqlQuery,
tableName);
    return table.Rows.Select(row => new Entity
                    {
                      Property = row[0] as string
                    } ).ToList();
  }
}
```

Listing 2: *Besser testbare Lösung: Hier ist das Einlesen der Daten aus der Datenbank getrennt von dem Umwandeln in Objekte, sodass man das Einlesen an anderer Stelle gut wiederverwenden kann und das Umwandeln problemlos in Unit-Tests testen kann.*

Schnittstelle möglichst einfach gehalten werden. Komplizierte Logik innerhalb dieser Schnittstelle ist nur schwer testbar. Alle Zugriffe auf andere Systemkomponenten – ob nun Datenbanken, File-System oder Web – sollten auch hinter Schnittstellen

```
public class SchlechtTestbaresLadenImHintergrund
{
  public void Initialize()
  {
    Parallel.Invoke(() => LadeWerte1(), () =>
LadeWerte2(), () => LadeWerte3());
  }
  private void LadeWerte1()
  {
  }
}
```

Listing 3: *Schlecht testbarer Code zum parallelen Laden von Daten. Hier ist es nicht möglich, das Laden der Daten ohne das Multi-Threading zu testen.*

abstrahiert werden, um einfache Unit-Tests zu ermöglichen.

Die Prinzipien des *Clean Code* (vgl. [Mar09]) helfen hierbei sehr. Schwierig zu testen sind z. B. große Klassen, lange Methoden oder tiefe Vererbungshierarchien. *Clean Code* gibt Hinweise, wie man diese Probleme umgehen und korrigieren kann (z. B. *Extract Method-Object, Dependency Inversion*). Die folgenden zwei Beispiele sollen verdeutlichen, wie das Refactoring aussehen kann.

Datenbankzugriffe

Heutzutage ist es üblich, einen OR-Mapper zu verwenden, um sich den Zugriff auf die Datenbank so leicht wie möglich zu machen. Glücklicherweise liefern diese auch schon alle eine Schnittstelle für die Session mit, sodass man hier einfach mit einem Mocking-Framework Unit-Tests erreichen kann. Am besten kapselt man alle direkt zur Datenbank abhängigen Methoden in einer eigenen Klasse Repository und testet nur diese gegen die Datenbank, alle restlichen Klassen werden nun gegen einen Mock getestet. Das ist nicht nur viel schneller, man hat auch keine Abhängigkeiten zwischen den Tests, was bei einer richtigen Datenbank doch leicht passieren kann.

Schwieriger wird es bei Zugriffen beispielsweise über ADO.NET (siehe Listing 1). Hier gibt es keine praktischen Interfaces. Deshalb ist es das erste Ziel, alle Datenbankzugriffe hinter einer Schnittstelle zu verstecken, um dann alle weiteren Methoden mittels eines Mocks testbar zu machen. Nur diese Klasse mit dem direkten Datenbank-Zugriff braucht nun eine wirkliche Datenbank. Gibt man z. B. nur noch eine DataTable für die Ergebnisse aus der Klasse zurück, kann man diese leicht testen (siehe Listing 2).

Nebenläufigkeit

Ein weiterer Problemfall ist Nebenläufigkeit (*Concurrency*) (siehe Listing 3). Häufig werden noch neue Threads direkt im Code erzeugt, sodass es unmöglich ist, einen Unit-Test zu schreiben, ohne auf mehrere Threads aufpassen zu müssen. Hier ist es das erste Ziel, die Nebenläufigkeit von der Berechnung zu trennen (siehe Listing 4). Wenn die gesamte Berechnungslogik in einer Klasse ist, die nicht von der Nebenläufigkeit weiß, kann man sie auch leicht testen. Bleibt also nur noch die Klasse, die sich um die Verdrahtung der einzelnen Methoden kümmert. Hat man häufige Probleme, wie z. B. ein gegenseitiges Sperren der Ressourcen oder noch nicht geladene Daten, muss man hier wohl auf ein Framework zum Testen der Nebenläufigkeit zurückgreifen, um dieses sicher zu testen. Aber immerhin braucht man sich nicht mehr um das Testen der

```
public interface IParallelService
{
  void RunParallel(params Action[] actions);
}
public class BesserTestbaresLadenImHintergrund
{
  public IParallelService ParallelService { get; set; }
  public void Initialize()
  {
    ParallelService.RunParallel(() => LadeWerte1(), ()
=> LadeWerte2(), () => LadeWerte3());
  }
  private void LadeWerte1()
  {
  }
}
```

Listing 4: *In diesem Beispiel wurde das parallele Laden der Daten hinter ein Interface verlagert. So kann man nun in Tests kontrollieren, was in welcher Reihenfolge passieren soll, um Fehler aufzudecken.*

Berechnung zu kümmern – das hat man schon in eigenen Unit-Tests erledigt.

Bewusstheit und Motivation

Neben technischen Aspekten gilt es auch, die verschiedenen Motive zu adressieren, die zu einer Bevorzugung schwergewichtiger Tests führen – diejenigen Motive also, die wir im ersten Teil dieses Artikels diskutiert haben. Häufig hilft bereits ein klares Bewusstsein solcher Tendenzen, um sie in der Teamarbeit einzudämmen. Ein wichtiger Schritt dazu ist sicher, im Team zu diskutieren, wie das Verhältnis leicht- und schwergewichtiger Tests zueinander aussehen soll und welche Schritte eingeleitet werden können, um dieses Ziel zu erreichen.

Auch wenn die nötigen analytischen und technischen Fertigkeiten gegeben sind, um schwergewichtige in leichtgewichtige Tests zu dekonstruieren, bleibt immer noch eine gewisse Schwelle zu überwinden, um diese Fertigkeiten im Entwicklungsalltag auch konsequent einzubringen. Hierzu können Teams die Mittel der agilen Vorgehensweise einsetzen, um Dekonstruktion im Prozess zu verankern. Beispielsweise kann es zu einem Kriterium von Code-Reviews innerhalb des Entwicklungsteams oder in der *Definition of Done* gemacht werden, dass neu entstandene Tests daraufhin geprüft werden, ob sie tatsächlich leichtgewichtig sind oder nicht. Es kann auch eine Regel oder Richtlinie im Team beschlossen werden, die vorgibt, dass stets leichtgewichtige und schwergewichtige Tests in einem gesunden Verhältnis geschrieben werden. Praktiken wie *testgetriebene Entwicklung (TDD)* oder *Pair Programming* fördern hierbei nahezu automatisch ein klareres Bewusstsein und eine Fokussierung auf leichtgewichtige Tests. Je nach verwendeten Test-Frameworks können gegebenenfalls einfache Metriken automatisiert auf dem *Continuous Integration-Server* erhoben werden: Alle Testfälle eines Testlaufs etwa könnten in der Reihenfolge ihrer Laufzeit aufgelistet werden und das Verhältnis aller Tests unterhalb einer Schwelle, die gerade noch als leichtgewichtig gilt, zu den anderen könnte bestimmt werden. (Wenn beispielsweise mehr als die Hälfte aller Tests länger als eine Zehntelsekunde läuft, besteht sicher ein Potenzial zur Dekonstruktion.) Dieses Verhältnis kann regelmäßig am Sprint-Ende betrachtet und gegebenenfalls zielgerichtet verbessert werden. Die genaue Form und Intensität solcher Maßnahmen hängt allerdings von der Entwicklungskultur im Team ebenso ab wie von konkreten technischen und organisatorischen Gegebenheiten.

Zusammenfassung

Schwergewichtige automatisierte Tests sind oft ein sinnvolles, manchmal ein notwendiges Mittel und haben daher einen festen Platz in der Werkzeugsammlung der Softwareentwicklung. Im Projektalltag gibt es jedoch vielfache Faktoren – oft nichttechnischer Natur –, die dazu führen, dass schwergewichtige Tests in höherem Maße entstehen, als es notwendig und vorteilhaft wäre. Dieser Tendenz sollte aktiv und kontinuierlich entgegengewirkt werden – durch Dekonstruktion schwergewichtiger Tests: Integrationstests sollten beständig in Unit-Tests überführt werden. Refactoring zur Testbarkeit, das Schreiben von Unit-Tests und eine weitgehende Auflösung der Integrationstests gehen dabei Hand in Hand. Wie andere Aktivitäten zur Wahrung der Qualität und Wartbarkeit einer Codebasis sollte dies zum festen Bestandteil der Code-Hygiene gemacht werden. ■

Literatur & Links

[Cohn] Mike Cohns Blog, siehe:
blog.mountaingoatsoftware.com/the-forgotten-layer-of-the-test-automation-pyramid

[Mar09] R.C. Martin, Clean Code – A Handbook of Agile Software Craftmanship, Prentice Hall 2009