Fight that State: verlässlichere Softwaresysteme durch funktionale Ansätze

von Benjamin Brunzel, iteratec GmbH

Aus verschiedenen praktischen Projekten habe ich gelernt, dass es sich lohnt, über den Tellerrand etablierter Programmierpraktiken hinauszuschauen. Als professioneller Softwareentwickler liegt meine Aufgabe nicht einzig und allein darin, Softwaresysteme technisch umzusetzen. Im Kern soll ein Problem des Nutzers gelöst werden. Hierfür gilt es zunächst, dieses zu verstehen, ein schlüssiges Softwarekonzept aufzustellen, in einzelne Schritte zu gliedern, zu priorisieren und schließlich schrittweise umzusetzen. Software wächst. So wird häufig aus einem performanten Prototypen ein träger Koloss: fehleranfällig, schwer zu warten und daher teuer in der Anpassung.

Komplexität ist das größte Problem bei der Entwicklung und Wartung von nicht trivialen Softwaresystemen. Oft ist es nicht leicht zu verstehen, wie die Abläufe in einem solchen System zusammenhängen. Der einzelne Entwickler kennt nicht alle Bereiche und Module gut genug, um selbstbewusst die ihm aufgetragenen Änderungen am Programm vorzunehmen. Das Verständnis der Software wird durch interne Zustände der Software zusätzlich erschwert. Die Vermeidung von zusätzlichen, nicht essenziellen Features ist wichtig. Das Besinnen auf einen minimal notwendigen Funktionsrahmen hat sich in der Softwareentwicklung, auch in Form von Mantras wie KISS („Keep it simple, stupid!“) oder YAGNI („You ain’t gonna need it“) bereits stark durchgesetzt. Dennoch ist es mit erheblichem Aufwand verbunden, ein System auch langfristig am Prinzip der Einfachheit auszurichten.

Zustand als Treiber der Komplexität

Durch eine Vielzahl von internen Zuständen wird ein System oder Teilsystem komplexer. Es wird nicht nur unübersichtlicher für Entwickler, sondern auch schwieriger zu testen. Warum ist das so? Um sicherzustellen, dass eine Funktion, ein Feature oder ein ganzes Programm das gewünschte Verhalten zeigt, kann der Programmcode auf zwei verschiedene Arten verifiziert werden. Erstens gibt es mit Test-Frameworks geschriebene Softwaretestfälle. Hierbei werden entweder einzelne Komponenten (Units) separat für sich oder im Zusammenspiel mit anderen Komponenten und Systemen getestet. Diese Tests sind eigene Programme und werden von den Entwicklern selbst verfasst. Sie stellen sicher, dass eine Komponente für einen bestimmten Satz an Eingabeparametern ein definiertes Set an Ausgaben erzeugt.

Das Problem bei solchen Unit Tests ist jedoch, dass ein Test für ein gewisses Set an Eingabeparametern nichts über das Verhalten mit anderen Parametern aussagt. Somit können diese Tests immer nur zeigen, dass Programmfehler vorhanden sind und nicht, dass keine vorhanden sind.

Auch die zweite Verifikationsmethode wenden Entwickler täglich an. Bei ihr geht es schlicht darum, den geschriebenen Programmcode zu lesen, nachzuvollziehen und zu verstehen. Je einfacher und verständlicher der Programmcode ist, desto schneller versteht ein Autor den Programmcode und findet die Stellen, an denen gezielte Änderungen zum gewünschten neuen Verhalten des Programms führen.

Veränderbarer globaler Zustand

Was zunächst banal klingt, kann nach meiner Erfahrung einen signifikanten Anteil des Zeitaufwands für eine Änderung ausmachen. Ein Beispiel:

```
class GlobalExample{
  static int sum = 0;

  static void add(int x){
    sum += x;
  }

  public static void main(String[] args)
  {
    add(2);
    add(5);
    System.out.println(“Summe: “+sum);
  }
}
```

Dieses simple Java-Programm war eine meiner ersten Begegnungen mit der Java-Entwicklung. Der Autor des Programms wollte damit Methodenaufrufe veranschaulichen. Wir begeben uns nun in den Kopf eines Entwicklers, der – wie oben erwähnt – das Programm liest und versucht, den Ablauf zu verstehen.

In Gedanken versucht er, den Programmablauf zu simulieren. Daher startet er wie die ausführende Java Virtual Machine (JVM) auch bei der *main*-Methode. Anders als diese wird er jedoch zunächst in einer höheren Flughöhe ansetzen und die gesamte *main*-Methode überfliegen, um ein grobes Verständnis der Funktionalität zu bekommen. Offensichtlich existiert eine *add*-Methode, welche zunächst mit Zwei, dann mit Fünf aufgerufen wird. Rückgaben dieser Methode werden nicht berücksichtigt. Der Methodenname suggeriert, dass eine Addition oder ein irgendwie geartetes Hinzufügen erfolgt (zum Beispiel wie beim Hinzufügen zu einer Datenstruktur). Schließlich wird eine Summe auf der Konsole ausgegeben. Das erhärtet diesen Verdacht. Doch wo wird diese Summe definiert? Über eine globale Variable. Zum Schluss muss die Funktion *add(int x)* im Detail betrachtet werden, um das Rätsel zu lüften. *add* erhöht die global Zustandsvariable *sum* um den übergebenen Wert *x*.

Das Programm tut, was es tun soll. Wo ist also der Haken? Unser Entwickler hat sich zunächst schwer getan zu verstehen, was hier eigentlich passieren soll. Die *main*-Methode ist nicht aussagekräftig, denn im obigen Beispiel musste zunächst das gesamte Programm betrachtet werden, um den Ablauf zu verstehen. Ein zweiter Kritikpunkt wird deutlich, wenn wir einen Softwaretestfall für die *add*-Methode schreiben wollen. Durch die Nutzung eines globalen Programmzustands, der Variable *sum*, lässt sich die Methode nicht mehr losge-

löst betrachten. Anstatt die Methode einfach isoliert testen zu können, muss zunächst die komplette Klasse instanziiert werden. Anschließend wird mithilfe von Reflection das statische Feld modifiziert. Das erhöht die Komplexität des Tests und den Aufwand für dessen Implementierung. Wir sehen also, dass die Funktion noch einen impliziten Eingabeparameter besitzt, nämlich den aktuellen Wert des *static-int*-Attributs *sum*.

Funktionale Ansätze

Im Beispiel haben wir gesehen, wie die Methode *add()* Speicherbereiche außerhalb ihres eigenen Geltungsbereichs beeinflusst. Neben dem Verarbeiten von Eingabeparametern zu Ausgaben wird der Zustand des Programms verändert. Solche Effekte werden als Seiteneffekte bezeichnet.

Was mit den weitverbreiteten prozeduralen und objektorientierten Programmiersprachen jederzeit möglich ist, wird in rein funktionalen Sprachen wie Haskell oder Idris bereits im Sprachdesign unterbunden. Hier erzeugt eine Funktion immer ausschließlich Ausgaben basierend auf den Eingabeparametern. Innerhalb der Funktion ist es nicht möglich, auf irgendetwas anderes als die expliziten Funktionsparameter zuzugreifen.

Ähnlich wie mathematische Funktionen sind somit Seiteneffekte in diesen rein funktionalen Programmteilen unmöglich. Das geht so weit, dass Programmteile, die mit der Außenwelt kommunizieren wollen (zum Beispiel Werte auf der Konsole ausgeben, Netzwerk-Sockets öffnen oder auf das File-System zugreifen), diese Operationen nur in speziellen als nicht rein funktional gekennzeichneten Umgebungen tun können.

Warum das Ganze? Wird hier nicht die Flexibilität des Entwicklers eingeschränkt? Natürlich. Die Vorteile sind jedoch immens. Betrachten wir doch einmal eine Haskell-Implementation unseres Code-Beispiels.

```
module GlobalExample where

add :: (Num a) => a -> a -> a
add x y = x + y

printSum :: (Show a) => a -> IO ()
printSum sum = do
    putStrLn ("Summe: " ++ (show sum))

main :: IO ()
main = do printSum ( 2 `add` 5)
```

Dieses Beispiel definiert drei Funktionen. Neben einer *add*- und einer *main*-Funktion gibt es eine *printSum*-Funktion. Außer diesen sind keine weiteren Symbole auf dem Toplevel definiert, das heißt, es gibt keine globale *sum*-Variable oder einen anderweitigen veränderbaren Zustand des Programms. Dies wäre auch nicht möglich, denn in Haskell gibt es überhaupt keine veränderbaren Variablen.

Es gibt keine Variablen in Haskell? Wie können dann überhaupt sinnvolle Programme erstellt werden? Anstatt veränderbarer Variablen nutzt Haskell sehr stark das Konzept der Rekursion, mit dem sich in Verbindung mit hohen Abstraktionsleveln (zum Beispiel mit *fold* oder *map*) dieselben Ergebnisse erzeugen lassen, wie es prozedurale Programme mit Schleifen und veränderbaren Variablen ermöglichen.

Wenn unser Entwickler nun dieses Programm analysiert, stellt er recht schnell fest, dass die *main*-Funktion deutlich aussagekräftiger ist als im vorhergehenden Beispiel. *main = do printSum (2 `add` 5)* sieht auf den ersten Blick so aus, als würde dort die Summe aus der Addition von zwei und fünf ausgegeben. Und das ist ja auch der Fall.

Als zweiten Schritt wird er sich die Typsignaturen der Funktionen ansehen. *add :: (Num a) => a -> a -> a* sagt, dass es sich um eine Funktion handelt, die zwei Werte desselben Typs kombiniert und einen Wert desselben Typs *(a)* zurückliefert. Dies wird mit *a -> a -> a* zum Ausdruck gebracht. *(Num a)* besagt, dass es sich beim Typ *a* um einen numerischen Wert handelt. Somit beschreibt diese eine Zeile alles, was die Funktion tun kann. Der Entwickler kann sich zu 100 Prozent sicher sein, dass diese Funktion keine versteckten Seiteneffekte auslöst.

Auch ein Softwaretest lässt sich nun deutlich einfacher implementieren (zum Beispiel *assertEqual "zwei plus fünf" 7 (add 2 5))*. Aus der Signatur der *printSum*-Funktion wird ersichtlich, dass diese einen Wert vom Typ *a* entgegennimmt und damit eine Seiteneffekt-behaftete Input-Output-Operation ausführt – in diesem Fall eine Ausgabe auf dem Terminal. Der Entwickler weiß somit, dass er hier unter Umständen genauer hinschauen sollte, da dieser Codebereich seiteneffektbehaftete Funktionen ausführen kann.

Die einzelnen Programmteile sind hier deutlich entkoppelt worden. Zudem wurde die *add*-Funktion auf einem generischen Level geschrieben. Dadurch kann sie – anders als im ersten Beispiel – zum Beispiel auch für Gleitkommazahlen verwendet werden. Diese Abstraktion sowie die Tatsache, dass jede Funktion nur von ihren Eingabewerten abhängt, sorgen dafür, dass rein funktionale Funktionen leichter an anderen Stellen wiederverwendet werden können. Zudem können solche Funktionen leicht kombiniert werden, um Neues einzubauen oder die bestehende Funktionalität eines Programms zu verändern. So kann beispielsweise unsere *add*-Funktion verwendet werden, um eine Liste von Zahlen um je einen Wert zu erhöhen.

```
addToListItems :: (Num a) => a -> [a] -> [a]
addToListItems x list = map (add x) list
```

Eine alternative Implementierung in Java

Können wir einige dieser Vorteile auch in unsere Java-Implementierung übernehmen? Tatsächlich! Zunächst wird der Methodenschnitt angepasst. Die *add*-Methode soll wie ihr funktionales Pendant zwei Input-Parameter bekommen. Dadurch kann die Zustandsvariable entfernt werden. Für die Ausgabe wird ebenfalls eine eigene Methode geschaffen.

```
class GlobalExample{

  static int add(int x, int y){
    return x + y;
  }

  static void printSum(int sum){
    System.out.println("Summe: " + sum);
  }

  public static void main(String[] args)
  {
    printSum(add(2 + 5));
  }
}
```

Fazit

Um ein wartbares und verlässliches Softwaresystem zu entwickeln und zu betreiben, muss die Komplexität so stark wie möglich reduziert werden. Die oben gezeigten funktionalen Ansätze helfen, den Code lesbar und wartbar zu halten.

Ich habe in verschiedenen Projekten durch Code-Reviews, Pair Programming und das schlichte Erweitern von Funktionalitäten bereits viel Programmcode gelesen. Ich habe mit Kundenvertretern und Nutzern um simple Lösungen gekämpft. Unsere Welt ist komplex genug, unser Handwerk als Softwareentwickler ebenso. Daher finde ich es enorm wichtig, die Eigenkomplexität des Programms so gering wie möglich zu halten. Es geht um die Lösung eines Nutzerproblems. Ich habe für mich festgestellt, dass funktionale Programmierung mir hilft, mich auf das Wesentliche zu konzentrieren.

Programmiersprachen wie Haskell nutzen diese Ansätze als Kerneigenschaften der Sprache und sind daher bestens geeignet, verlässlichere Software zu entwickeln. Doch auch mit prozeduralen Sprachen lassen sich viele der Vorteile nutzen. Ich versuche in meiner täglichen Arbeit – so weit wie möglich – auf Funktionen ohne Rückgabewert (*void*) zu verzichten. Zudem kann der Compiler helfen, fehlerhafte Nutzung anhand der Typen bereits beim Übersetzen der Software aufzudecken. Dafür müssen aber zunächst die Parameter einer Signatur in deren Funktion explizit sein. Ich kann jedem Java-Entwickler ans Herz legen, sich einmal mit den in Java 1.8 hinzugefügten Sprach-Features auseinanderzusetzen. Viele davon (Streams, Lambdas, Optionals) transportieren weitere Ideen aus der funktionalen Programmierung.

Erst in die Branche eintauchen und dann die IT-Lösung an die Oberfläche bringen

von Heiko Packwitz, Lufthansa Industry Solutions AS GmbH

Personalpläne mithilfe eines Selfservice-Systems optimieren

Für das Universitätsklinikum Heidelberg (UKHD) hat Lufthansa Industry Solutions ein modernes Selfservice-System entwickelt. Das neue Organisations-Tool unterstützt sowohl das Personalmanagement als auch die Mitarbeiter der Klinik, indem es eine effizientere Planung ermöglicht. So bietet es unter anderem die Option, individuelle Wünsche bei der Erstellung von Dienstplänen zu berücksichtigen und besser zu koordinieren.

Das Universitätsklinikum Heidelberg hatte zuvor bereits eine systemunterstützte Personaleinsatzplanung für den ärztlichen Bereich eingeführt. Es zeigte sich dann aber, dass sie über die Einführung von Selfservices weiter optimiert werden könnte. So waren zum Beispiel die Prozessabläufe in den unterschiedlichen Klinikbereichen hinsichtlich Urlaubsplanung oder auch Wunschdiensten samt Genehmigungsverfahren inhomogen.

Durch eine klassische Personaleinsatzplanung, primär gesteuert von Einsatzplanern, lassen sich standardisierte, zentral ausgerichtete Abläufe realisieren. Mitarbeiter und Führungskräfte können jedoch nur wenig flexibel in den gesamten Gestaltungsprozess einbezogen werden. Die Folge: Routineaufgaben wie die Erstellung von Dienstplänen waren umständlich und zeitintensiv.

Von der Analyse über das Konzept bis zur Umsetzung

Im vorliegenden Fall hat sich das UKHD entschlossen, Employee- und Manager-Selfservices zu etablieren, um damit den Selbstgestaltungsanteil bei der Erstellung der Dienstpläne signifikant zu erhöhen. Gemeinsam mit dem externen Dienstleister hat die zentrale IT des UKHD in einem mehrstufigen Projekt zahlreiche Selfservices eingerichtet und sukzessive in die einzelnen Klinikbereiche ausgerollt. Die Vorgabe des UKHD war, eine Lösung zu entwickeln, die für alle Klinikbereiche einheitliche Standards bietet und die Effizienz des bisherigen Prozesses deutlich verbessert.

Das ambitionierte Projekt begann mit einer umfangreichen Analysephase. In der anschließenden Konzeptionsphase wurde zunächst ein SAP-basiertes Soll-Konzept entwickelt. Gemeinsam mit Projektmitarbeitern aus dem Bereich Human Resources und IT haben die Experten der Lufthansa-Tochter anschließend entschieden, wie die Szenarien aufgesetzt werden sollen. Dabei wurden auch die entsprechenden Soll-Prozesse gemeinsam konkretisiert. Es folgte eine Realisierungsphase, bei der eine iterative Entwicklungsmethodik zum Einsatz kam.