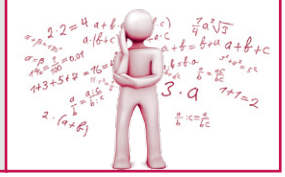
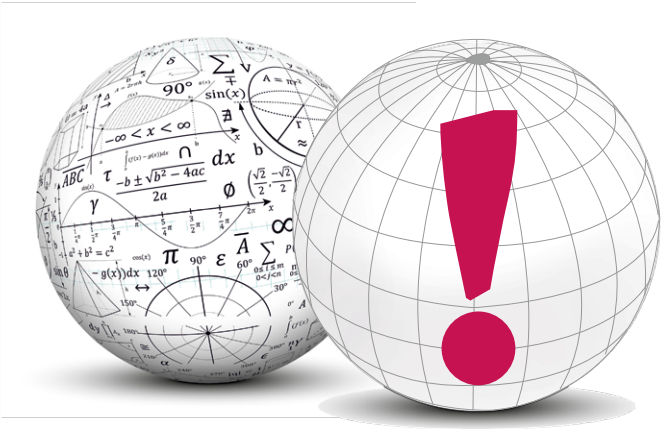


Das Beste beider Welten

Funktional versus imperativ

Beate Ritterbach, Axel Schmolitzky

In der Praxis der Softwareentwicklung dominieren objektorientierte Sprachen (wie Java oder C#) und prozedurale Sprachen (wie C oder Pascal). Damit überwiegen Vertreter des imperativen Programmierparadigmas. Sprachen wie Scala oder Clojure und die Diskussion um Lambdas in Java ab Version 8 haben aber jetzt auch jenseits der akademischen Welt ein Interesse an funktionaler Programmierung geweckt. Um von den Vorzügen beider Welten zu profitieren, erscheint es naheliegend, die beiden Paradigmen zu verbinden.



Dieser Artikel beschreibt die Kernkonzepte der funktionalen Programmierung und vergleicht sie mit denen der imperativen Programmierung. Er arbeitet die Stärken und Schwächen der beiden Paradigmen heraus und stellt Sprachen vor, welche die beiden Welten miteinander kombinieren. Eine solche Vereinigung führt jedoch in ein Dilemma, das verhindert, von den jeweiligen Vorteilen gleichermaßen zu profitieren. Der Artikel skizziert einen Weg, wie die grundlegenden Merkmale und Vorzüge beider Paradigmen in einer Sprache erhalten bleiben können.

Funktionale Programmierung

Das Leitbild der funktionalen Programmierung sind Funktionen *im mathematischen Sinne* – eindeutige Abbildungen von einer Definitionsmenge auf eine Zielmenge. Beispiele für Funktionen sind Sinus, Quadratwurzel, Determinante (einer Matrix) oder die Addition (die zwei Zahlen auf eine dritte, nämlich ihre Summe, abbildet).

Funktionen sind zustandslos. Funktionale Programmierung kennt keine änderbaren Variablen und damit keine Zuweisungen. Eine Funktion verändert nicht ihre Parameter, sondern sie ermittelt ausgehend von den Parametern ein Ergebnis. Die Berechnung einer Quadratwurzel wandelt zum Beispiel nicht die Eingabe 9 in die 3 um, sondern sie ordnet der 9 das Ergebnis 3 zu. Funktionale Programmierung besteht im Wesentlichen aus der Anwendung („Applikation“) von Funktionen auf ihre Parameter, weswegen sie auch als *applikativ* bezeichnet wird.

Die Ein- und Ausgaben von Funktionen sind *Werte*, beispielsweise mathematische Abstraktionen wie Zahlen, Vektoren oder Matrizen. Auch viele nicht-mathematische Abstraktionen sind Werte: Zeichen und Zeichenketten, Längen- und Zeitangaben, Datum oder Geldbetrag. Werte zeichnen sich durch Unveränderlichkeit und durch eine feste Wertemenge aus: Da jede Verarbeitung zustandslos ist, kann es keine Veränderungen geben. Das Hinzufügen oder Löschen eines Wertes würde ebenfalls eine Zustandsänderung darstellen und ist damit ebenfalls nicht möglich.

Ausgehend von Zustandslosigkeit haben sich in funktionalen Sprachen spezifische Idiome und Sprachkonstrukte etabliert. Zum Beispiel werden Wiederholungen typischerweise mit Hilfe von Rekursion ausgedrückt. Schleifen – kennzeichnend für imperative Sprachen – wären wirkungslos, weil es keinen Zustand gibt, der sich mit jedem Schleifendurchlauf ändern könnte. Weil Funktionen im Mittelpunkt des Ansatzes stehen, unterstützen viele funktionale Sprachen sie als „Werte erster Klasse“, das heißt, sie unterliegen keinen Einschränkungen und können auf die gleiche Weise verarbeitet werden wie Zahlen, Tupel oder andere, einfachere Werte. Insbesondere können Funktionen Eingabe oder Ausgabe anderer Funktionen sein. Das ermöglicht die Bildung von „Funktionen höherer Ordnung“, wie die Listenfunktionen map, select oder fold.

Als Initialzündung für funktionale Programmierung gilt ein Vortrag, den John Backus 1977 anlässlich der Verleihung des Turing-Awards hielt (für seine Arbeit an der imperativen(!) Sprache Fortran). Darin kritisiert er die Einschränkungen, welche die implizite Ausrichtung imperativer Programmiersprachen auf die vorherrschende Von-Neumann-Rechnerarchitektur mit sich bringt [Back77]. Er schlägt als Gegenentwurf ein radikal anderes, nämlich das funktionale Verarbeitungsmodell vor, und illustriert es am Beispiel der von ihm entwickelten Sprache *FP*.

Bereits vor Backus' Rede gab es Sprachen mit funktionalen Merkmalen. Beispielsweise zählt das in den 1950er Jahren entwickelt *LISP* zu den funktionalen Sprachen. Es ist bis heute in verschiedenen Dialekten in Gebrauch, dazu zählen *Common-Lisp*, *Scheme* und die JVM-Sprache *Clojure*. Nachfolgende Sprachen haben die funktionale Programmierung um zusätzliche Konzepte erweitert: *Hope* führte algebraische Datentypen und das zugehörige Pattern-Matching ein. *ML* enthielt als erste Sprache ein Verfahren, um Typen aus dem Kontext zu erschließen und so die Notwendigkeit expliziter Typannotationen zu reduzieren („Typinferenz“). *Miranda* entwickelte eine sprachgestützte verzögerte Auswertung von Operationen und Datenstrukturen („lazyness“). *Haskell* vereinigte diese und weitere Konzepte in einer Sprache, fügte einige neue hinzu (z. B. Typklassen) und kann heute als „state of the art“ im Bereich funktionaler Programmierung angesehen werden.

Funktionale Sprachen gibt es in zwei Varianten:

- *Impure* Sprachen – wie ML und die meisten Lisp-Dialekte – unterstützen Funktionen, Werte und viele der eben genannten Konzepte; sie fördern einen funktionalen Programmierstil, erlauben daneben aber auch Zustand (z. B. durch Zuweisungen und änderbare Variablen).
- *Rein* funktionale („*pure*“) Sprachen – wie Hope, Miranda oder Haskell – verzichten konsequent auf alle zustandsbezogenen Sprachkonstrukte, sie erzwingen zustandslose Verarbeitung und lassen keinen anderen Programmierstil zu.

Imperative/objektorientierte Programmierung

Imperative Sprachen sind inhärent zustandsorientiert. Ein Programm in einer imperativen Sprache besteht typischerweise aus einer Abfolge von Anweisungen (imperare = lat. befehlen, anweisen). Anweisungen bewirken Zustandsänderungen – auch als „Seiteneffekte" bezeichnet. Die elementaren Sprachbausteine dafür sind Zuweisungen. Sie ändern den Inhalt einer Variablen und bilden ein universelles Mittel, eine Zustandsänderung abzubilden, zum Beispiel den Anstieg eines Aktienkurses oder die Gewichtsabnahme eines Patienten. Variablen in imperativen Sprachen sind Abstraktionen von Speicherplätzen; der Zustand eines Programms besteht aus dem aktuellen Inhalt aller seiner Variablen. Mit diesem auf den Speicher ausgerichteten Modell lehnen sich imperative Sprachen eng an die zugrunde liegende Von-Neumann-Architektur des Rechners an.

Mit Zustand als Grundlage des Sprachmodells gehen typisch imperative Steuerungsanweisungen einher. Beispielsweise werden Wiederholungen in der Regel durch Iteration abgebildet (while- oder for-Schleifen). Sie sind erst durch Zustand möglich: Die Steuerung selbst – die Prüfbedingung oder die Variable zum Hochzählen der Schleifendurchläufe – muss änderbar sein; auch ist eine mehrfache Ausführung desselben Schleifenrumpfs nur deshalb sinnvoll, weil sie aufgrund fortschreitender Zustandsänderungen bei jedem Durchlauf etwas anderes bewirkt.

Viele imperative Sprachen sind *prozedural*, wie *Fortran*, *Cobol*, *PL/I*, *Algol*, *C* oder *Pascal*. Eine Prozedur (Routine, Unterprogramm, ...) ermöglicht dem Programmierer, einen zusammengehörenden Block von Anweisungen zusammenzufassen, als Einheit aufzurufen und die Details seiner Implementierung vor Aufrufern zu verbergen („information hiding").

Auch objektorientierte Sprachen sind im Kern imperativ und damit zustandsorientiert: Jedes Objekt verfügt über einen internen Zustand. Er ist gekapselt, das heißt, andere Objekte können nicht direkt auf ihn zugreifen, sondern nur mit Hilfe der dafür vorgesehenen Operationen (die in der objektorientierten Terminologie nicht mehr als Prozeduren, sondern als „Methoden" bezeichnet werden). Information hiding erstreckt sich in objektorientierten Sprachen nicht nur auf das Verbergen der Implementierung einer Operation, sondern auch auf das Verbergen der internen Datenstruktur der Objekte. Diese „Partitionierung des Zustandes", seine „Aufteilung in autonome, gekapselte Einheiten (=Objekte)", ist das wesentliche Merkmal der objektorientierten Programmierung [Weg90].

Typisch objektorientierte Sprachkonzepte, wie Objekte, Klassen, Identität, Kapselung, Vererbung usw., sind kein Gegensatz zu prozeduraler Programmierung, sondern vielmehr eine darauf aufbauende Erweiterung. Objektorientierung ist ein Unterparadigma der imperativen Programmierung (s. Abb. 1). Das zeigt sich auch darin, dass die Grundbausteine imperativer Sprachen – Variablen, Zuweisungen, Schleifen usw. – auch in objektorientierten Sprachen zu finden sind. Die zusätzlichen Konzepte haben allerdings in so beträchtlichem Umfang Abstraktionsniveau, Modularität und Verständlichkeit der Sprachen gesteigert, dass der Übergang von prozeduraler zu objektorientierter Programmierung vielfach als „Paradigmenwechsel" empfunden (und vermarktet) wurde.

Simula67, auf Algol aufbauend und für Simulation entworfen, gilt als erste objektorientierte Sprache. Alan Kay entwickelte in den 1970er Jahren *Smalltalk* und etablierte den Begriff der „objektorientierten Programmierung". Die darin enthaltenen Ideen sind auf breite Akzeptanz gestoßen und haben nachfolgende Sprachen maßgeblich beeinflusst: *C++*, *Objective-C* und *Eiffel*. Basierend auf den Erfahrungen damit wurden *Java*, *C#* und weitere Sprachen entwickelt, unter anderem das kürzlich herausgekommene *Ceylon*. Zu mehreren prozeduralen Sprachen gibt es objektorientierte Erweiterungen (z. B. *ObjectPascal/Delphi*). Auch Skriptsprachen verfügen über objektorientierte Bestandteile (*Ruby*, *Python*, *Groovy*).

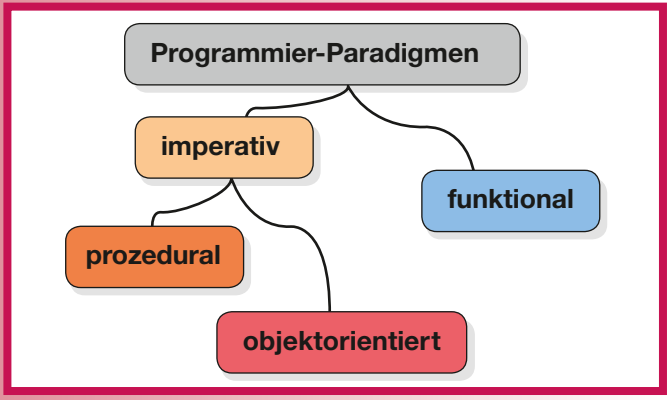


Abb. 1: Programmierparadigmen

Gegenüberstellung

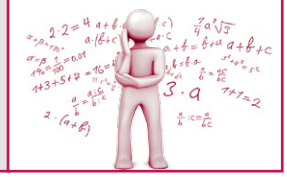
Das grundlegende Merkmal funktionaler Programmierung ist Zustandslosigkeit, imperative Programmierung dagegen basiert auf (prinzipiell änderbarem) Zustand.

Zustandslosigkeit ist mehr als nur Unveränderlichkeit. Beispielsweise bewirkt der Aufruf einer Operation wie „Summe der eingezahlten Beiträge aller Lebensversicherungen von Klaus Meyer" keine Seiteneffekte – es handelt sich um eine Query (zum Prinzip der Trennung von Commands und Queries s. [Mey97]). Dennoch ist die Operation zustandsabhängig. Sie macht eine Aussage über den aktuellen Zustand der Welt, und das Ergebnis kann zu verschiedenen Zeitpunkten unterschiedlich ausfallen – abhängig davon, ob Klaus weiter einzahlt, neue Verträge abschließt oder einer seiner Verträge fällig wird. Eine Funktion im Sinne der funktionalen Programmierung, wie „Quadratwurzel aus 625", ist dagegen zeit- und zustandslos. Sie liefert unabhängig von Geschehnissen in der realen Welt bei gleichen Eingaben immer das gleiche Ergebnis – eine Eigenschaft, die als *referentielle Transparenz* bezeichnet wird.

Funktionale und imperative Abstraktionen – Werte respektive Objekte – befinden sich auf verschiedenen Daseinsebenen. Werte, wie Zahlen, Zeichen, Farben oder Formen, abstrahieren vollständig von den Gegenständen, die durch sie beschrieben werden können. Werte existieren deshalb nicht in Raum und Zeit. Objekte beschreiben dagegen Gegenstände, Sachverhalte oder sonstige Aspekte einer „realen" Welt, wie Personen oder Fahrzeuge, sie können auch ungegenständlich sein, wie Versicherungsverträge oder Urlaubsreisen. Objekte können prinzipiell ihren Zustand ändern und haben damit einen Bezug zu Raum und Zeit.

Objekte können in Einzelfällen auch unveränderlich sein. Trotzdem werden sie dadurch nicht zu Werten. Zum Beispiel sind Buchungssätze in einem revisionssicheren System per Definition unveränderlich. Dennoch bilden sie Zustände der realen Welt ab. Die Erzeugung eines solchen Buchungssatzes verändert die Gesamtmenge der vorhandenen Buchungen, sie kann sich auf das Ergebnis von Operationen (z. B. Kontensalden) auswirken und ist damit eine Zustandsänderung.

Tabelle 1 stellt die essenziellen Merkmale funktionaler und imperativer Programmierung gegenüber.



funktionale Programmierung auch bezeichnet als: applikativ, wertorientiert	**imperative Programmierung** Unterparadigmen: prozedural, objektorientiert
Zustandslosigkeit	Zustand, prinzipiell änderbar
keine Zuweisungen, keine Anweisungen	Zuweisungen, Anweisungen
Variable = Name für eine feste Größe	Variable = änderbarer Speicherplatz
Mathematische Funktionen	Prozeduren bzw. Methoden
Werte	Objekte
Mathematische Abstraktionen	Gegenstände der „realen" Welt
Verarbeitung = Anwenden von Funktionen auf ihre Parameter = Auswertung von Ausdrücken	Verarbeitung = Ausführung von Anweisungen = schrittweise Änderung des Zustandes
Reihenfolge der Auswertung ist irrelevant	Reihenfolge der Ausführung ist entscheidend

Tabelle 1: Gegenüberstellung funktionale vs. imperative Programmierung

Stärken und Schwächen der beiden Paradigmen

Funktionale und imperative/objektorientierte Programmierung sind grundverschieden. Beide Paradigmen haben spezifische Stärken und Schwächen.

Der konsequente Verzicht auf Zustandsänderungen macht (rein) funktionale Programmierung sicher und verringert ihre Fehleranfälligkeit. Es ist leichter, funktionale Programme nachzuvollziehen, zu analysieren und zu testen; es gibt sogar Ansätze, ihre Korrektheit durch formale Verfahren nachzuweisen. Weil das Ergebnis einer Funktion nicht von einem Zustand und nicht von der Reihenfolge der Auswertung der Teilausdrücke abhängt, sind funktionale Programme inhärent „thread-safe". Sie eignen sich damit ideal für nebenläufige Programmierung bzw. parallele Verarbeitung.

Gleichzeitig ist die fehlende Unterstützung für änderbaren Zustand die größte Schwäche funktionaler Programmierung. Gegenstände der realen Welt und ihre Veränderungen im Laufe der Zeit passen nicht ins funktionale Schema. Um sie dennoch abzubilden, muss der Programmierer Umwege gehen, die den Code verkomplizieren und seine Verständlichkeit beeinträchtigen: „*Purely applicative languages are poorly applicable*" [Per82].

Imperative Sprachen sind von Haus aus für die Abbildung von Zustand konstruiert. Objekte bilden auf einfache und natürliche Weise Gegenstände der realen Welt ab, seien das nun Fahrzeuge, Dampfdruckkessel oder Bankkonten. Darum können viele unterschiedliche Gegenstandsbereiche geradlinig auf ein programmiertechnisches Modell abgebildet werden; es gibt keinen methodischen Bruch zwischen Wirklichkeit und Programm. Das prädestiniert objektorientierte Sprachen für vielfältige Anwendungsgebiete. Die mit der Objektorientierung eingeführten Konzepte, insbesondere Kapselung, haben dazu beigetragen, die mit Zustand einhergehende Komplexität zu reduzieren („to tame the state"). Aus all diesen Gründen haben sich objektorientierte Sprachen in der kommerziellen Softwareentwicklung auf breiter Front durchgesetzt.

Mit dem Glaubensgrundsatz „Alles ist ein Objekt" haben objektorientierte Sprachen Werte vernachlässigt beziehungsweise sie implizit zu (speziellen) Objekten erklärt. Werte und Funktionen sind zwar in den meisten objektorientierten Sprachen rudimentär vorhanden (z. B. in Form von vordefinierten Basisdatentypen wie bool, int oder float und ihren zugehörigen Operatoren). Aber fachlich motivierte Werte, die nicht als Basisdatentypen vorliegen – beispielsweise Geldbetrag, Datum oder Komplexe Zahl –, müssen zwangsläufig mit Hilfe von Objektklassen abgebildet werden. Wert-artige Abstraktionen als Objekte abzubilden, ist jedoch ähnlich umständlich und fehleranfällig, wie umgekehrt der Zwang, Objekte in ein Schema von wertorientierter Verarbeitung zu pressen.

Hybride Sprachen: eine Synthese?

Das imperative Paradigma, insbesondere die Objektorientierung, dominiert bis heute die kommerzielle Softwareentwicklung. Funktionale Ansätze haben lange ein weitgehend unbeachtetes Dasein in akademischen Nischen gefristet. Historisch haben sich imperative und funktionale Sprachen parallel zueinander entwickelt und nur in geringem Umfang gegenseitig beeinflusst.

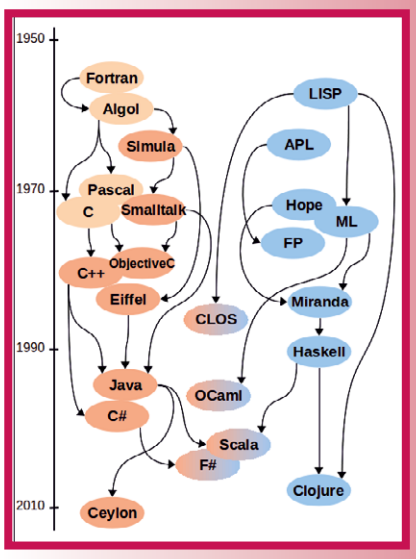


Abb. 2: Historische Sprachlinien

Eine der ersten Sprachen, die funktionale und objektorientierte Programmierung vereinte, war *CLOS* (Common Lisp Object System, 1988). Ein weiteres Beispiel ist *O'Caml* (1996), eine Erweiterung von ML um objektorientierte Konstrukte wie Klassen und Vererbung. Auch *F#* und *Scala* verfolgen das ausdrückliche Ziel, funktionale und objektorientierte Programmierung miteinander zu verbinden. In solchen hybriden Sprachen finden sich sowohl „typisch funktionale" Merkmale (z. B. Funktionen höherer Ordnung, Lazyness, Pattern-Matching) als auch „typisch objektorientierte" (z. B. Klassen, Vererbung und die damit verbundene Subtyppolymorphie). Abbildung 2 zeigt die historische Entwicklung der Sprachlinien. Aus Gründen der Übersichtlichkeit beschränkt sie sich auf eine repräsentative Auswahl von Sprachen und deren Einflüssen aufeinander.

Hybride Sprachen, die mehr sein wollen als ein Konglomerat unterschiedlicher Konzepte und Mechanismen, stehen vor einem Dilemma: Funktionale und objektorientierte (bzw. generell imperative) Programmierung sind nicht nur verschieden – sie schließen einander aus. Zustandslosigkeit ist das logische Gegenteil von änderbarem Zustand. Die beiden Paradigmen erscheinen unvereinbar.

Die meisten hybriden Sprachen, auch die eben aufgeführten von CLOS bis Scala, schließen bei der Kombination der beiden Welten Zustand ein, sie werden dadurch „impure". Damit verlieren sie die wesentlichen Vorteile funktionaler Program-

mierung: die mit der garantierten Abwesenheit von Zustand einhergehende Sicherheit und alle daraus resultierenden Eigenschaften, von Nachvollziehbarkeit bis zur Eignung für parallele Verarbeitung (s. o.).

Es gibt auch den umgekehrten Ansatz: „pure" Sprachen, die die Definition von Klassen sowie Kapselung und Vererbung unterstützen, dabei aber Zustandsänderungen ausschließen. Er findet sich vorwiegend im akademischen Umfeld (z. B. *O'Haskell* oder *Clover*). Auch hier wirkt sich die Unvereinbarkeit der beiden Welten aus, nun in umgekehrter Richtung. Eine solche Sprache profitiert zwar von den Vorteilen des funktionalen Paradigmas. Sie eignet sich aber gerade wegen ihrer fehlenden Unterstützung von Zustand nicht für die Abbildung von Gegenständen der realen Welt und ist damit für den Einsatz in der Praxis kaum geeignet.

Ausblick: Ein alternativer Ansatz

Aktuell befasst sich ein Forschungsprojekt an der Universität Hamburg mit der Integration von Werttypen in objektorientierte Programmiersprachen. Es zielt darauf ab, funktionale (wertorientierte) und objektorientierte Programmierung zu verbinden und setzt dazu an den grundlegenden Merkmalen der beiden Paradigmen an: Zustandslosigkeit versus Zustand:

- Werte repräsentieren die zustandslose,
- Objekte die zustandsbehaftete Welt.

Der vorgeschlagene Ansatz besteht darin, die beiden verschiedenen Welten auf der Ebene der Programmiersprache voneinander zu trennen und für Wert- und Objekttypen jeweils eigene Typkonstruktoren bereit zu stellen. Die Sprache erhält dadurch ein zweigeteiltes Typsystem. (Die vorgeschlagenen Werttypen bilden nicht dasselbe Konzept wie z. B. sogenannte „value types" in C#. Diese unterstützten direkte Speicherung, wie auch expanded types in Eiffel, sie sind aber nicht zustandslos.)

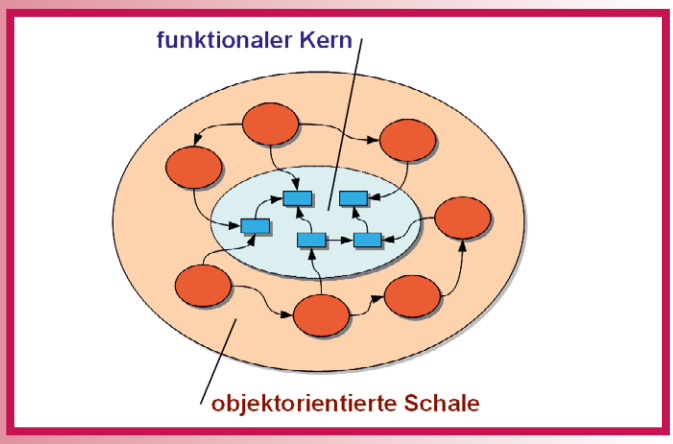


Abb. 3: Das Beste beider Welten – zweigeteiltes Typsystem

Die Zweiteilung des Typsystems entfernt sich von dem Credo „Alles ist ein Objekt", das nicht nur objektorientierten, sondern auch hybriden Sprachen wie Scala zugrunde liegt. Es folgt statt dessen der Devise „Es gibt Objekte, und es gibt Werte". Ein solches Typsystem teilt jede Anwendung in zwei Bereiche auf, die hier als „funktionaler Kern" und „objektorientierte Schale" bezeichnet werden (s. Abb. 3). Der funktionale Kern besteht aus der Menge aller Werte und ihrer zugehörigen Operationen (=Funktionen). Er bildet ein in sich abgeschlossenes Teilsystem, innerhalb dessen rein funktionale Verarbeitung durch die Sprache garantiert wird. In der „objektorientierten Schale" befinden sich Objekte und ihre Methoden. Dieser Bereich repräsentiert Zustand und unterstützt die von objektorientierten Sprachen gewohnte zustandsbezogene Verarbeitung. Die beiden Bereiche stehen in einem asymmetrischen Abhängigkeitsverhältnis: Objekte können sich auf Werte beziehen, Werte aber nicht auf Objekte.

Der Ansatz verbindet funktionale und objektorientierte Sprachelemente so, dass die spezifischen Merkmale und Stärken beider Paradigmen erhalten bleiben. Im funktionalen Kern ist Zustandslosigkeit mit allen daraus resultierenden Vorteilen garantiert. In der objektorientierten Schale können Gegenstände der realen Welt und ihre Veränderungen geradlinig und ohne Umwege abgebildet werden. Die Sprache ist hybrid und bleibt trotzdem in jedem Teilbereich „pur" (s. [C2wiki]).

Das vorgestellte Projekt befindet sich im Forschungsstadium. Es lotet mögliche Gestaltungen aus, erkundet das Zusammenspiel von Werten und Objekten und untersucht die Auswirkungen des zweigeteilten Typsystems auf andere Sprachkonstrukte.

Literatur und Links

[Back77] J. W. Backus, Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs, in: Commun. ACM, August 1978, s. a.
`http://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf`

[C2wiki] Object Functional,
`http://c2.com/cgi-bin/wiki?ObjectFunctional`

[Mey97] B. Meyer, Object-oriented software construction (2nd ed.), Prentice-Hall Inc., 1997

[Per82] A. J. Perlis, Epigrams in Programming, in: SIGPLAN Notices Vol. 17, No. 9, 1982

[Weg90] P. Wegner, Concepts and Paradigms of Object-oriented Programming, in: SIGPLAN OOPS Mess., 1990

Dipl.-Math. **Beate Ritterbach** arbeitet seit über zwanzig Jahren freiberuflich als Softwareentwicklerin und Methodenberaterin, schwerpunktmäßig im Umfeld Objektorientierung. Aktuell schreibt sie an ihrer Dissertation zum Thema „Werttypen in objektorientierten Programmiersprachen".
E-Mail: ritterbach@informatik.uni-hamburg.de

Dr. Axel Schmolitzky wird demnächst von der Universität Hamburg auf eine Professur für Softwareentwicklung an die Hochschule für angewandte Wissenschaften (HAW) in Hamburg wechseln.
Er beschäftigt sich seit zwanzig Jahren mit dem Design objektorientierter Sprachen sowie ihrem Nutzen in der Anwendungspraxis. In den letzten Jahren hat er an der Uni Hamburg systematisch die einführende Programmierausbildung nach dem Objects First-Ansatz aufgebaut. Außerdem will er die Ausbildung in den Bereichen Agile Softwareentwicklung und Softwarearchitektur verbessern.
E-Mail: schmolit@informatik.uni-hamburg.de