Werttypen in objektorientierten Programmiersprachen: Anforderungen an eine Sprachunterstützung

Beate Ritterbach
Axel Schmolitzky

Universität Hamburg, Department Informatik,
Vogt-Koelln-Str. 30, D-22527 Hamburg, Germany
beate.ritterbach@studium.uni-hamburg.de
schmolit@informatik.uni-hamburg.de

Abstract: In der objektorientierten Modellierung von Anwendungssystemen werden *Werte* und *Objekte* häufig als unterschiedliche Abstraktionen aufgefasst. Durch die im softwaretechnischen Umfeld dominierenden objektorientierten Programmiersprachen fällt die Abbildung von Objekten eines Anwendungsbereichs auf Objektklassen dieser Sprachen inhärent leicht, während wertartige Abstraktionen umständlich repräsentiert werden müssen. Eine Programmiersprache, die neben *Objekttypen* auch *Werttypen* durch explizite Mechanismen unterstützt, könnte Abstraktionen, die konzeptionell als Werte einzustufen sind, klarer, knapper und sicherer abbilden. In diesem Artikel geben wir eine Definition von Werttypen und diskutieren, welchen Anforderungen eine Sprache genügen sollte, die dieses Konzept von Werttypen geeignet unterstützt.

1 Einführung

Etliche Veröffentlichungen im Bereich der softwaretechnischen Modellierung, beispielsweise [Zü04], [Bä98], [He00], [We98], [EK95] und [Ho07], weisen darauf hin, dass *Werte* (teilweise auch bezeichnet als „value objects“ oder „Fachwerte“) ein eigenständiges und für die Modellierung relevantes Konzept darstellen. Offensichtlich ist nicht „alles ein Objekt“, häufige fachliche Abstraktionen wie *Datum, Geldbetrag, kartesischer Punkt, Zeitraum* oder *IP-Adresse* weisen Eigenschaften auf, die sie näher an die primitiven Typen objektorientierter Programmiersprachen heranrücken als an deren Objekttypen. Letztere bilden typischerweise zustandsbehaftete und veränderbare Objekte ab, wie etwa *Kunde*, *Konto*, *Bauteil* oder *Vertrag*, die auch einen eindeutigen Erzeugungszeitpunkt aufweisen.

Obwohl es etliche Hinweise auf die Relevanz von Werten in der Modellierung gibt, gehört deren explizites Erkennen und Modellieren nicht zum Standardrepertoire. Dies kann daran liegen, dass oft keine klare Trennlinie zwischen Werten und Objekten gezogen werden kann. Es kann aber auch sein, dass die dominierende Sicht der Objektorientierung in der Modellierung zu einer gewissen „Wertblindheit“ geführt hat.

Doch selbst wenn bei den Architekten eines Softwareprojektes ein klares Verständnis für wertartige Abstraktionen im Anwendungsbereich vorhanden ist und sie geübt und gezielt fachliche Konzepte als Werte modellieren, besteht in der technischen Realisierung das Problem, dass üblicherweise objektorientierte Programmiersprachen wie Java, C++ oder C# zum Einsatz kommen. In diesen Sprachen lassen sich Objekttypen (der Modellierung) direkt in *Objektklassen* (der Programmiersprache) abbilden; Werttypen hingegen passen nicht in das Grundschema. Deshalb wurden, um Werttypen zu programmieren, spezielle Regeln und Konventionen empfohlen, z. B. das Value object pattern [Ri06]. Es kommt beispielsweise in [Ev04] und [Fo08] zum Einsatz.

Diese Ansätze stimmen darin überein, *wie* ein Werttyp abgebildet werden sollte: durch eine Klasse, die auf verändernde Methoden verzichtet (Code-Beispiele sind u. a. In [Fo08] zu finden.) Sie legen allerdings nicht klar fest, *was* unter Werten zu verstehen ist. Es bleibt offen, ob Werte dasselbe sind wie unveränderbare Objekte, d. h. Ob Unveränderbarkeit genügt, um Werte zu charakterisieren. Auch Arbeiten, die sich aus einer anderen Perspektive mit Werten befassen (z.B. ihrer effizienten Implementierung [Ba03], als Spezialfall eines allgemeineren Konzeptes „relation types“ [Va07], oder als Teil des Programmiermodells von X10, einer Sprache für Non-Uniform Cluster Computing [Ch05]) nennen nur Unveränderbarkeit als ihre kennzeichnende Eigenschaft.

Mit diesem Artikel möchten wir zwei Beiträge leisten: Zum einen geben wir in Abschnitt 2 eine Definition von Werttypen, die klarer und umfassender ist als alle informellen Beschreibungen, die wir bisher zu diesem Thema finden konnten. Diese Definition kann bereits für sich genommen hilfreich sein, ein klareres Verständnis in der Modellierung zu entwickeln. Darüber hinaus stellen wir in Abschnitt 3 dar, welche Probleme sich ergeben, wenn Werttypen durch Objektklassen abgebildet werden müssen. Aus diesen Problemen leiten wir in Abschnitt 4 Anforderungen an ein Sprachmodell ab, das Werttypen als gleichberechtigte Abstraktionen ansieht. Abschnitt 5 diskutiert verwandte Ansätze und Abschnitt 6 fasst die Arbeit zusammen.

2 Werte als explizites Konzept

Die u. a. in [Ev04], [Fo08] und [Ri06] aufgeführten Beispiele - *Zahl*, *Zeichenkette*, *Datum*, *Punkt*, *Geldbetrag*, etc. - und die Art ihrer Verwendung legen nahe, dass Werte mehr gemeinsam haben als Unveränderbarkeit. Motiviert durch die *Fachwerte* im *Werkzeug&Material-Ansatz* [Zü04], befassen wir uns am Arbeitsbereich Softwaretechnik der Universität Hamburg seit geraumer Zeit systematisch mit Werttypen ([Wi10], [Ra07], [Ra06], [He05], [Ri04], [FS02], [Sa01], [Mü99]). Dabei wurde immer deutlicher, dass eine präzise Definition des Wertbegriffs notwendig ist, um eine technische Unterstützung (per Framework, per Sprache oder nur per Konvention) für sie anbieten zu können.

2.1 Definition: Wert und Werttyp

Die Erkenntnisse aus den genannten Vorarbeiten führten zu folgender Definition:

Definition:

Werte sind die Elemente eines **Werttyp**s. Ein Werttyp beschreibt die zum Typ gehörenden **Elemente** und die für sie aufrufbaren Operationen.

1. Werte sind **unveränderbar**.
2. Werte werden **nicht erzeugt** und nicht vernichtet.
3. Wert-Operationen sind **seiteneffekt-frei**.
4. Wert-Operationen sind **referentiell transparent**, d. h. wenn sie mehrmals mit denselben Parametern aufgerufen werden, liefern sie immer dasselbe Ergebnis.

Mit diesem Begriff von Werten und Werttypen stützen wir uns insbesondere auf die grundlegende Arbeit von MacLennan [Ma82], der Werte als *zeitlose*, *zustandslose Abstraktionen* beschreibt, die unabhängig von Raum und Zeit existieren. Die Definition baut auf einem abstrakten Typbegriff auf, d. h. einer Menge von Elementen, die durch die für sie verfügbaren Operationen charakterisiert sind [LZ74]. Dementsprechend sind die Eigenschaften 1. bis 4. als *konzeptionelle Eigenschaften* zu verstehen; sie beschreiben das Verhalten von Werten und Wertoperationen aus der Sicht von Klienten, unabhängig von der zugrunde liegenden Implementation. Diese Definition gründet nicht mehr ausschließlich auf Unveränderbarkeit. Unerzeugbarkeit ist nach dieser Definition ebenfalls eine wesentliche, in den bisherigen Arbeiten vernachlässigte Eigenschaft von Werten. Sie impliziert u. a., dass auch die Elementmenge eines Werttyps unveränderlich ist.

2.2 Rationale

Die Wahl der genannten Eigenschaften als charakterisierend für Werte wurde insbesondere durch die beiden folgenden Gesichtspunkte motiviert:

a.) Primitive Datentypen wie Ganzzahlen oder Gleitkommazahlen können als Prototypen für Werttypen angesehen werden. Sie weisen die genannten Eigenschaften auf: Ihre Elemente sind unveränderbar und werden nicht erzeugt, ihre Operationen (syntaktisch oft in der Form von Infix-Operatoren wie +, -, <, ...) sind seiteneffektfrei und referentiell transparent. Wenn unsere Definition von Werttypen eine Klasse von Typen beschreibt, dann sind die primitiven Typen objektorientierter Programmiersprachen Exemplare dieser (Meta-)Klasse.

b.) Zeitlosigkeit und Zustandslosigkeit sind eher vage Begriffe, die präzisiert werden müssen. Erzeugung und Veränderung sind Vorgänge, die sich in der Zeit abspielen - Unerzeugbarkeit und Unveränderbarkeit beschreiben die Zeit-Losigkeit von Werten. Seiteneffekte und *Referentielle Opakheit* (das Gegenteil von Referentieller Transparenz) beruhen auf änderbarem Zustand. Die jeweiligen Gegenstücke, Seiteneffektfreiheit und Referentielle Transparenz, beschreiben zusammen die Zustands-Losigkeit von Werten: Eine Operation ohne Seiteneffekte *bewirkt* keine Zustandsänderungen, eine referentiell transparente Operation macht keine Zustandsänderungen *sichtbar*. Zustandslosigkeit ist essentiell in einem Umfeld, das grundsätzlich änderbaren Zustand zulässt, z. B. in objektorientierten Sprachen.

3 Implementierung von Wert-Typen mit Objekt-Klassen

Die vordefinierten primitiven Datentypen, die von gängigen objektorientierten Sprachen wie Java, C++ oder C# zur Verfügung gestellt werden (int, float, char, ...), genügen unserer Definition von Werttypen. Doch die jeweilige Sprachdefinition gibt Art und Anzahl der primitiven Datentypen fest vor, es können nicht problemlos weitere derartige Typen definiert werden. Wenn in einem Projekt *fachlich motivierte* Werttypen (*Datum*, *Uhrzeit*, *Punkt*, *KomplexeZahl* oder *Geldbetrag*) benötigt werden, ist man in den genannten Sprachen gezwungen, sie durch Objektklassen abzubilden, mit Hilfe von Mustern und Codierungs-Konventionen. Um einige Konsequenzen dieses Vorgehens aufzuzeigen, wird im folgenden die Programmierung eines Typs „Date“ in Java mit Hilfe des Value object patterns ([Ri06]) skizziert. Er soll eine simple Abbildung eines Datums, festgelegt durch Tag, Monat und Jahr, darstellen, Datumsarithmetik ermöglichen, ungültige Werte (wie den 31. Februar 2012) ausschließen, etc. Die Klasse „Date“ definiert ausschließlich Operationen, die ihre Elemente nicht ändern, z. B. liefert `addDays` ein anderes Datum statt das bestehende zu ändern. Die Operation `equals` implementiert die fachliche Gleichheit von Datumswerten, zusammen mit `equals` ist auch `hashcode` zu implementieren.

Code-Beispiel: Definition einer Java-Klasse „Date“

```
class Date {
  private int day, month, year;
  public Date(int year, int month, int day) {
    this.year = year;
    this.month = month;
    this.day = day;
  }
  public Date(int year, int dayOfYear) {
    this.year = year;
    this.month = ...;
    this.day = ...;
  }
  public int getDay() { return day; }
  public int getMonth() { return month; }
  public int getYear() { return year; }
  public int dayOfYear() { return ...; }
  Date addDays(int days) {
    int day = this.day + days;
    if (day > 31) ...
  }
  public boolean equals(Object obj) {
    if (this == obj) {return true;}
    if (obj == null) ...
  }
  public int hashCode() { ... }
  }
}
```

Eine mögliche Benutzung der Klasse Date kann z. B. folgende Anweisungen beinhalten:

Code-Beispiel: Benutzung der oben definierten Klasse „Date“

```
1   Date d1 = new Date(2010,1,30);     // Jan. 30th 2010
2   d1.addDays(7);                     // no effect
3   Date d2 = d1.addDays(7);           // Feb. 6th 2010
4   Date d3 = new Date(2010, 37);      // Feb. 6th 2010
5   if (d2 == d3) {}                   // false
6   if (d2.equals(d3)) {}              // true
```

Der isolierte Aufruf der Methode `addDays` in Zeile 2 ist wirkungslos und darum sinnlos; er führt aber nicht zu einer Fehlermeldung. In Zeile 5 liefert der Vergleich mittels „==“ false; es ist anzunehmen, dass der Klient prüfen wollte, ob sich `d2` und `d3` auf das gleiche Datum beziehen. Die Benutzung von „==“ ist demnach fachlich nicht korrekt, der Code müsste stattdessen wie in Zeile 6 aussehen. Die Konstruktor-Aufrufe in Zeile 1 und 4 suggerieren, dass Elemente von `Date` erzeugt werden; das aber steht im Widerspruch zum Konzept von Werten.

Eine Ursache dieser Schwierigkeiten ist die konzeptionelle Unerzeugbarkeit von Werten. Sie kann im Code nicht zum Ausdruck gebracht werden, da Elemente von Objektklassen immer erzeugt werden müssen. Ein weiteres Problem ist, dass die „Wert-Artigkeit“ durch Kombination mehrerer Einzel-Maßnahmen erreicht wird (Verzicht auf ändernde Methoden, Überschreiben der equals-Methode, ...). So ist sie schwer erkennbar und bei Code-Überarbeitung leicht zerstörbar. Und schließlich gibt es keine Sprachunterstützung für referentielle Transparenz und Seiteneffektfreiheit, da die Sprache Seiteneffekte grundsätzlich zulässt.

Aus diesen Gründen ergeben sich auch bei Einsatz anderer Patterns (z. B. flyweight [Ga95]) oder bei Verwendung anderer objektorientierter Sprachen wie C++ oder C# die gleichen Nachteile: umfangreicher Code, fehleranfällige Benutzung, keine Sicherheit bezüglich der Werteigenschaften. Untermauert wird diese Erkenntnis u. a. durch die Arbeit von Winkler, welche die Umsetzung von Wertypen mit den Sprachmitteln von C++ untersucht [Wi10]. Sie macht deutlich, dass die Programmierung von Klassen, die gemäß der obigen Kriterien als Werttypen anzusehen sind, möglich ist, aber durch die Sprachmechanismen von C++ stark erschwert wird. Der entsprechende Code ist umständlich, nicht als Werttyp erkennbar und in vielen Fällen ineffizient.

Zur Vermeidung von Mißverständissen weisen wir darauf hin, dass C# über den Typkonstruktor „`struct`“ verfügt und die damit definierten Typen im C#-Umfeld als „value types“ bezeichnet werden. Doch der Typkonstruktor `struct` gewährleistet keine der o. g. Eigenschaften von Werten; er bietet lediglich eine Unterstützung für Wertsemantik anstatt – wie bei reference types üblich – für Referenzsemantik.

4 Anforderungen an eine Sprachunterstützung für Werttypen

Werte nach unserer Definition sind rein funktionale Abstraktionen. Wie eben gezeigt, sind objektorientierte Sprachen per se schwach darin, Werte zu unterstützen. Ihre Stärke besteht in der Abbildung von Objekten – veränderbaren, erzeugbaren Abstraktionen und

ihren Zuständen. Objekte und Werte werden gleichermaßen benötigt; *beide* sollten durch geeignete Abstraktionen in einer Programmiersprache unterstützt werden. Aus softwaretechnischer Sicht fordern wir dabei:

1. *Klarheit*: Werttypen sollen im Quelltext deutlich erkennbar und von Objekttypen unterscheidbar sein.
2. *Sicherheit*: Die charakteristischen Eigenschaften von Werten sollen durch Sprachregeln garantiert werden (und möglichst bereits vom Compiler überprüfbar sein).
3. *Einfachheit*: Einfache Werttypen sollen mit geringem Aufwand definiert und benutzt werden können.

Wenn man das Klassenkonstrukt objektorientierter Sprachen (üblicherweise durch das Schlüsselwort `class` eingeleitet) als einen *Typkonstruktor* für Objekttypen bezeichnet (denn es wird u.a. ein Typ konstruiert), dann sollten Werttypen durch einen eigenen, zusätzlichen Typkonstruktor unterstützt werden. Klassen, die durch den postulierten Typkonstruktor für Werttypen definiert werden, bezeichnen wir als *Wertklassen*. Wir setzen voraus, dass Wertklassen ebenso wie Objektklassen nach dem Prinzip der Kapselung aufgebaut werden: durch Festlegung von *Datenfeldern*, welche die interne Repräsentation bilden, und von Operationen, die auf die Repräsentation zugreifen dürfen und für Klienten aufrufbar sind. (In speziellen Fällen ist ein anderes Konstruktionsprinzip möglich: die Definition eines Werttyps mittels Aufzählung. Sie wird im Rahmen dieser Arbeit nicht betrachtet.)

[Ra07] beschreibt Entwurf und Implementation der Sprache VJ, einer Erweiterung von Java um Wertklassen. VJ zeigt, dass und wie ein Werttypkonstruktor in eine objektorientierte Sprache eingebunden werden kann. Ein VJ-Compiler ist unter [VJ] zu finden.

Die Anforderungen an einen Werttypkonstruktor werden im folgenden anhand eines Code-Beispiels illustriert. Wie in Abschnitt 3 bildet es einen Werttyp „Date“ ab. Für Wertklassen gibt es eine Vielzahl möglicher syntaktischer Gestaltungen; deren Pro und Kontra soll im Rahmen dieser Arbeit nicht diskutiert werden. Der Verständlichkeit halber kommt für das Code-Beispiel eine an Java orientierte Syntax zum Einsatz, ähnlich der in VJ verwendeten. (Der Deutlichkeit halber sind andere Schlüsselworte gewählt als in VJ, dort waren aus Kompatibilitätsgründen pragmatische Kompromisse eingegangen worden.) Das Beispiel setzt voraus, dass ein Werttyp Int bereits existiert. Dabei ist irrelevant, ob es sich um einen vordefinierten Typ handelt oder ob er ebenfalls als Wertklasse definiert wurde.

Code-Beispiel: Definition einer Wertklasse „Date“

```
valueclass Date {
  Int _day, _month, _year;
  Int day() {return _day;}
  Int month() {return _month;}
  Int year() {return _year;}

  static Date dateYMD (Int year, Int month, Int day) {
    return select(day, month, year)
  }
```

```
  static Date dateYearDayOfYear (Int year, Int ddd) {
    Int month = ...
    Int day = ...
    return dateYMD(year, month, day)
  }
  Date addDays(Int days) {
    Int day = _day + days
    if (day > 31) ...
    return select(day, month, year)
  }
  Int dayOfYear() {return ... }
  ...
}
```

Code-Beispiel: Benutzung der o.g. Wertklasse „Date“

```
1 Date d1 = Date.dateYMD(2010,1,30);          // Jan. 30th 2010
2 d1.addDays(7);                              // Compile-Fehler!
3 Date d2 = d1.addDays(7);                    // Feb. 6th 2010
4 Date d3 = Date.dateYearDayOfYear(2010, 37); // Feb. 6th 2010
5 if (d2 == d3) ...                           // true
```

Um hervorzuheben, dass Werte und Objekte verschiedene Abstraktionen darstellen und keine ein Spezialfall der anderen ist, sollten jeweils dedizierte Schlüsselworte verwendet werden, z. B. `objectclass` und `valueclass`. Der Suffix „class“ deutet darauf hin, sowohl ein Typ als auch seine zugehörige Implementation definiert wird, eine pragmatische, in objektorientierten Sprachen übliche Verschmelzung.

Wir stellen einen Satz von Sprachregeln auf, welche die charakteristischen Eigenschaften von Werten sicherstellen:

- Restriktionen für die Datenfelder (4.1)
- Selektoren statt Konstruktoren (4.2)
- Gleichheit basierend auf den Datenfeldern (4.3)
- Kein Zugriff auf Objekte (4.4)

4.1 Restriktionen für die Datenfelder

Wir fordern, dass die Datenfelder einer Wertklasse unveränderbar und ihre Typen ebenfalls Werttypen sind. Zusammen bewirken diese beiden Bedingungen, dass Elemente einer Wertklasse unveränderbar sind.

4.2 Selektoren statt Konstruktoren

Die Unerzeugbarkeit von Werten und die Forderung nach Klarheit der Sprachunterstützung implizieren, dass eine Wertklasse keine Konstruktoren haben sollte. Eine Anweisung wie `...new Date(2010,5,3)...` wäre verwirrend und würde die Bedeutung des Typs verschleiern. Es besteht aber auch für Werttypen der Bedarf, auf einen Wert (der konzeptionell bereits vorhanden ist) gezielt zuzugreifen. Das wird mit

Operationen erreicht, die einen Wert des eigenen Typs liefern, im Beispiel etwa `dateYMD` oder `dateYearDayOfYear`. Wir nennen eine derartige Operation *Selektor*, denn sie dient dazu, einen spezifischen Wert aus dem „Werte-Universum" dieses Typs zu selektieren; ihre Parameter legen fest, welchen.

Selektoren unterscheiden sich grundlegend von Konstruktoren: Mehrere Aufrufe eines Selektors mit denselben Parametern liefern immer denselben Wert. Denn Selektoren sind Operationen eines Werttyps und müssen deshalb referentiell transparent sein. Im Gegensatz dazu liefern mehrere Aufrufe eines Konstruktors, auch mit denselben Parametern, immer verschiedene Objekte.

Der Code eines Selektors muss ein Element der eigenen Wertklasse liefern. Gemäß den bisher genannten Voraussetzungen gibt es kein Sprachmittel, womit das bewerkstelligt werden kann; ein solches muss postuliert werden. Wir schlagen dafür eine spezielle, für jede Wertklasse automatisch verfügbare Operation vor: sie wird durch ein eigenes Schlüsselwort aufgerufen („`select`"), ihre Parameter entsprechen in Anzahl und Reihenfolge den Datenfeldern der Wertklasse, und sie liefert dasjenige Element der Wertklasse, dessen Datenfelder genau diese Werte annehmen. Diese spezielle Operation bezeichnen wir als *primären Selektor*. Er sollte nur innerhalb der Wertklasse aufrufbar sein, weil seine Parameter Aufschluss über die interne Repräsentation der Wertklasse geben. Im Beispiel wird der primäre Selektor von `dateYMD` und `addDays` aufgerufen.

4.3 Gleichheit basierend auf den Datenfeldern

Bei einer Wertklasse muss die Gleichheit auf ihren Datenfeldern beruhen. Wenn alle Datenfelder übereinstimmen, muss es sich um denselben Wert handeln - anderenfalls wäre der primäre Selektor nicht referentiell transparent. Im Gegensatz dazu beruht der Vergleich von Objekten auf dem Konzept der Identität, zwei verschiedene Objekte können denselben Zustand haben.

(Wert-)Gleichheit und (Objekt-)Identität können als ein einheitliches Konzept aufgefasst werden: Hinter beiden Bezeichnungen steht die Aussage, dass zwei Ausdrücke dasselbe Element bezeichnen. Dementsprechend fordern wir, dass die Sprache nur einen einzigen Operator (oder eine Operation) für den Vergleich anbietet; er sollte bei Wertklassen die Wertgleichheit, bei Objektklassen die Identität repräsentieren. Bei einer solchen Gestaltung gibt es – anders als im Java-Beispiel aus Abschnitt 3 – keine Möglichkeit, zu prüfen, ob für einen konzeptionellen Wert mehrere Speicherrepräsentationen vorliegen – eine Aussage, die fachlich nicht relevant ist, sondern die zugrunde liegende technische Implementation auf der Ebene der Anwendungsprogrammierung durchscheinen lässt. Das Verhalten von Wertklassen in Bezug auf den Vergleich von Ausdrücken würde damit dem von primitiven Datentypen gleichgestellt, die ebenfalls nur einen Vergleichsoperator kennen. Und es verhindert potentielle Programmierfehler, die aus der versehentlichen Benutzung des „falschen" Vergleichs resultieren – z. B. in Java beim Vergleich von String-Ausdrücken mit „==" statt mit „`equals`".

4.4 Kein Zugriff auf Objekte

Die formalen Parameter und das Ergebnis einer Wertoperation sollten ebenfalls Werttypen sein. Denn wären Objekte als Parameter zulässig, dann könnte die Wertoperation Methoden dieser Objekte aufrufen, die aber können Seiteneffekte hervorrufen oder referentiell opak sein. Objekttypen scheiden somit als mögliche Parametertypen aus. Ein Objekt als Ergebnis der Wertoperation könnte dann nur durch einen Konstruktoraufruf produziert werden, was bei jedem Aufruf ein neues Objekt ergibt und damit die referentielle Transparenz zerstört. Auf ähnliche Art und Weise kann begründet werden, dass auch die Typen lokaler Variablen keine Objekttypen sein sollten.

Die vorgeschlagenen Regeln für den Werttypkonstruktor erfüllen zusammengenommen die geforderten softwaretechnischen Kriterien. Sie bewirken, dass die vorgegebenen Eigenschaften von Werttypen garantiert werden und sind zur Übersetzungszeit überprüfbar („Sicherheit“). Aufgrund des dedizierten Typkonstruktors sind Werttypen im Quelltext als solche erkennbar und ihre intendierte Benutzung für Klienten festgelegt („Klarheit“). Die Regeln sind so gewählt, dass sie für den Anwendungsprogrammierer leicht nachvollziehbar sind („Einfachheit“).

Um die Eigenschaften von Werttypen sicherzustellen, sind sie *hinreichend,* in dieser Hinsicht aber nicht notwendig: Beispielsweise wäre es möglich, das Verbot des Zugriffs auf Objekttypen zu lockern. Dann aber müsste ein komplexeres Regelwerk erstellt werden, welches verhindert, dass Wertoperationen durch Aufruf von Objektoperationen Seiteneffekte hervorrufen oder nicht mehr referentiell transparent sind.

Auch weitere wert-spezifische Konsistenzprüfungen sind möglich, so z. B. das Abfangen von isolierten und damit wirkungslosen Aufrufen von Wertoperationen, wie im Beispiel in Zeile 2.

Die Regeln, konsequent in einer Programmiersprache umgesetzt, führen zu einem asymmetrischen Abhängigkeitsverhältnis zwischen Werten und Objekten: Werte haben keinen Zugriff auf Objekte. Objekte können umgekehrt Werte benutzen, z. B. als Parameter oder Rückgabe-Typen, als Datenfelder oder lokale Variablen. Im Ergebnis wird dadurch jedes System unterteilt in einen „*funktionalen Kern*“ und eine „*objekt-orientierte Schale*“. Der funktionale Kern besteht aus den Werten und ihren zugehörigen Operationen (=Funktionen), er hat keinerlei Kenntnis von oder Zugriff auf die objektorientierte Schale. Diese wiederum umfasst Objekte und deren Operationen und kann beliebig Werte benutzen und Wert-Operationen aufrufen.

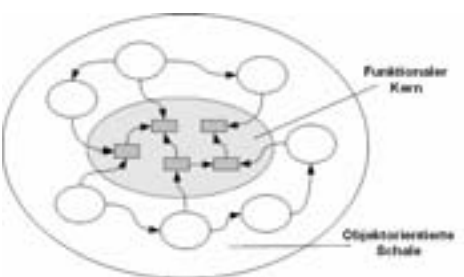

Fig. 1. Funktionaler Kern und objektorientierte Schale

Die Trennung von Objekttypen und Werttypen und ihre konsequente Unterstützung kann so die Ausgangsbasis für den Entwurf einer objekt-funktionalen Sprache bilden. Innerhalb des funktionalen Kerns ist rein funktionale Verarbeitung gewährleistet, was Programmverifikation, Parallelisierung und Optimierung erleichtert. In der objektorientierten Schale wird die Simulation von Objekten und ihren Zustandsänderungen einfach und geradlinig durch passende Abstraktionen unterstützt.

5 Verwandte Arbeiten

Eine objekt-funktionale Sprache, die auf der getrennten Unterstützung von Objekttypen und Werttypen aufbaut, wäre gemäss der in [C2] vorgenommenen Einteilung als *„pure-hybride“* zu klassifizieren. Anders als in einer „hybriden“ objekt-funktionalen Sprache (wie Ocaml [OC] oder Python [Py]) oder einer „puren“ objekt-funktionalen Sprache (wie O'Haskell [OH]) ist darin nicht eines der beiden Paradigmen das führende, und das jeweils andere durch zusätzliche Konstruktionen aufgesetzt, sondern beide Paradigmen stehen gleichberechtigt nebeneinander. Werte und Objekte - respektive Funktionen und Objekt-Operationen - bleiben jeweils als solche erkennbar, die spezifischen Eigenheiten beider Paradigmen bleiben erhalten.

Scala und Fortress sind zwei objekt-funktionale Forschungssprachen, die einige Ähnlichkeiten mit den in diesem Artikel vorgeschlagenen Konzepten aufweisen.

Scala [Od08] wurde an der EPFL von einer Arbeitsgruppe um Martin Odersky entwickelt. Es ist das explizite Ziel dieser Sprache, objektorientierte und funktionale Programmierung zu vereinen. Scala läuft auf der Java Virtual Machine, mittlerweile (seit Juli 2010) ist sie in Release 2.8 veröffentlicht. Scalas Typsystem kennt keine Trennung von Werten und Objekten. In Scala ist jedes Element ein Objekt, d. h. die Instanz einer (Objekt-)Klasse, und jede Operation eine Methode. Damit gibt es in Scala keinen funktionalen Kern, in dem pur funktionale Verarbeitung garantiert ist. Im Prinzip kann jede Operation Seiteneffekte haben oder sichtbar machen.

Es gibt in Scala Parallelen zu der hier vorgeschlagenen Unterstützung von Werttypen. Bei Case Classes erhalten die drei Methoden `toString`, `equals` und `hashcode` automatisch eine Implementation, welche die Struktur der Klassen-Member berücksichtigt, und ihr Konstruktor kann von Klienten ohne das Schlüsselwort „`new`“ aufgerufen werden. Damit weist ihre Benutzung äußerlich Ähnlichkeit mit Werttypen auf, allerdings ohne ihre charakteristischen Eigenschaften zu garantieren.

In Scala delegiert der Operator „==“ immer an die Methode `equals`. Wenn `equals` bei wert-artigen Abstraktionen passend implementiert ist, verhält sich „==“ so, wie oben vorgeschlagen: bei wert-artigen Klassen als Wertgleichheit und bei Objektklassen als Objektidentität. Allerdings gibt es neben „==“ bzw. „`equals`“ noch den Vergleich mit „`eq`“. Damit kann ein Referenzvergleich erzwungen werden, was bei wert-artigen Abstraktionen nicht angebracht ist und die zugrunde liegende Implementation durchscheinen lassen würde.

Fortress [Al08] ist eine unter der Leitung von Guy Steele in den SUN Labs entwickelte Sprache für das High Performance Computing. Sie ist ausgerichtet auf Erweiterbarkeit - viele ihrer Konstrukte sind in Bibliotheken statt im Sprachkern verankert - und die direkte Darstellung mathematischer Notationen. Fortress wurde sowohl von objektorientierten Sprachen wie Java und Eiffel als auch von funktionalen Sprachen wie ML und Haskell inspiriert.

Das Typsystem von Fortress ähnelt dem von Scala, es kennt Objekte, Traits und Vererbung. Es bietet keine dedizierte Unterstützung für Werttypen, sondern betrachtet alle Abstraktionen als Objekte - auch z. B. die Elemente primitiver Typen wie Zahlen und Zeichen, die nicht durch die Sprache selbst, sondern in Bibliotheken definiert werden. Es gibt kein Mittel, um wert-artige Abstraktionen als Werte kenntlich zu machen und die charakteristischen Werteigenschaften für sie zu garantieren.

In Fortress werden Ausdrücke üblicherweise mit dem Operator „=“ verglichen. Anders als z. B. in Java, C#, C++ oder Eiffel gibt es keine weiteren Vergleichsmethoden; der Operator „=“ kann in jeder Klasse nach Bedarf überschrieben werden kann. Damit gibt es nur einen einzigen Vergleich, was die Sprache in dieser Hinsicht einfach macht und unserer oben aufgestellten Forderung nach einem einheitlichen Vergleichsverfahren entspricht.

6 Zusammenfassung

Werttypen spielen eine wichtige Rolle in der Modellierung. Die Programmierung von Werttypen mit den Mechanismen bestehender objektorientierter Sprachen ist möglich, doch der entstehende Code ist umständlich und schwer wartbar. In dieser Arbeit haben wir, basierend auf einer ausführlich reflektierten Definition von Werttypen, einen Vorschlag entwickelt, wie solche Werttypen explizit in objektorientierten Sprachen unterstützt werden können.

Literaturverzeichnis

[Al08] Allen, E. et. al.: The Fortress Language Specification, Version 1.0. Sun Microsystems Inc. 2008, http://labs.oracle.com/projects/plrg/fortress.pdf

[Ba03] Bacon, D.F.: Kava: A Java dialect with a uniform object model for lightweight classes. Concurrency—Practice and Experience 15(3–5), 185–206, 2003

[Bä98] Bäumer, D., et al.: Values in Object Systems. Ubilab Technical Report 98.10.1, UBS AG, Zurich, Switzerland, 1998.

[Ch05] Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th OOPSLA, San Diego, CA, USA, pp. 519–538. ACM Press, New York, 2005

[C2] Cunningham & Cunningham, Inc., Wiki, ObjectFunctional languages. URL: http://www.c2.com/cgi/wiki?ObjectFunctional.

[EK95] Eckert, G., Kempe, M.: Modeling with Objects and Values: Issues and Perspectives. In: ROAD (Report on Object Analysis & Design), vol. 1, no. 5, pp. 20-27, Jan-Feb 1995.

[Ev04] Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley, Boston, 2004
[FS02] Fürter, M., Spreckelmeyer, A.: Konzepte und Ansätze zur Unterstützung der Implementierung fachlicher Werte in objektorientierten Programmiersprachen. Diplomarbeit, Universität Hamburg, Arbeitsbereich Softwaretechnik, 23.08.2002
[Fo08] Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley, Boston, 2008
[Ga95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)
[He00] Henney, K.: Patterns in Java: Value Added. Java Report 5(4), April 2000
[He05] Heiden, M.: Generierung von Fachwerten aus einem abstrakten Sprachmodell. Studienarbeit, Universität Hamburg, Arbeitsbereich Softwaretechnik, 14.07.2005
[Ho07] Hoogendoorn, S.: Die Implementierung von Value-Objekten. In: ObjektSpektrum, pp. 52-56, November 2007
[Lo93] Louden, K.: Programming Languages: Principles and Practice. Wadsworth Publ. Co., 1993
[LZ74] Liskov, B., Zilles, S.: Programming with Abstract Data Types. In: SIGPLAN Notices, 9:4, pp. 50-59, 1974.
[Ma82] MacLennan, B.J.: Values and Objects in Programming Languages. ACM SIGPLAN Notices 17,12, pp. 70-79, 1982
[Mü99] Müller, K.: Konzeption und Umsetzung eines Fachwertkonzeptes, Studienarbeit, Universität Hamburg, Arbeitsbereich Softwaretechnik, 30.08.1999
[OC] OCaml, URL: http://caml.inria.fr/ocaml/
[Od08] Odersky, M. ; Spoon, L., Venners, B.: Programming in Scala. Mountain View, Calif., 2008.
[OH] O'Haskell, URL: http://www.haskell.org/haskellwiki/O'Haskell
[Py] Python. URL: http://www.python.org
[Ra06] Rathlev, J.: Ein Werttyp-Konstruktor für Java, Diplomarbeit, Universität Hamburg, Arbeitsbereich Softwaretechnik, 11.08.2006
[Ra07] Rathlev, J., Ritterbach, B., Schmolitzky, A.: Auf der Suche nach Werten in der Softwaretechnik (Kurzbeitrag), In: Lecture Notes in Informatics - Proceedings SE 2007, Hamburg, pp. 261-262, 2007
[Ri04] Ritterbach, B.: Support for Value Types in an Object-Oriented Programming Language. In: Net.ObjectDays 2004, Erfurt, Germany, September 27-30, Proceedings, Volume 3263 of Lecture Notes in Computer Science, pp. 9-23, Springer, 2004
[Ri06] Riehle, D.: Value object. Proceedings of the 2006 conference on Pattern languages of programs, pp. 1-6, 2006
[Sa01] Sauer, J.: Generierung von Fachwerten aus XML-Beschreibungen. Studienarbeit, Universität Hamburg, Arbeitsbereich Softwaretechnik, 09.07.2001
[Va07] Vaziri, M., Tip, F., Fink, S., Dolby, J.: Declarative Object Identity Using Relation Types. ECOOP 2007: pp. 54-78, 2007
[VJ] VJ Research Language. URL: http://sourceforge.net/projects/vj-lang/
[We98] Werf, P. v. d.: Values and Objects Revisited. In: Journal Of Object Oriented Programming, pp. 25-34, 7/1998
[Wi10] Winkler, F.: Benutzerdefinierte Werttypen in C++. Diplomarbeit. Universität Hamburg, Arbeitsbereich Softwaretechnik, 08.09.2010
[Zü04] Züllighoven, H.: Object-Oriented Construction Handbook. Dpunkt-Verlag, Copublication with Morgan-Kaufmann, 2004