Universität Hamburg
Fachbereich Informatik
Softwaretechnik

Bachelorarbeit

Umsetzung von Werttypen in Scala

Erstgutachter:	Dipl.-Math. Beate Ritterbach
Zweitgutachter:	Dr. Axel Schmolitzky

Vorgelegt von:

Erik Witt
Matrikelnr.: 6204671
Heidmühlenweg 90c
25336 Elmshorn
0witt@informatik.uni-hamburg.de

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Die Modellierung des Anwendungsbereichs ist ein fundamentaler Bestandteil der Softwareentwicklung. Die Qualität der Abbildung von Gegenständen des Anwendungsbereichs auf Bestandteile des Softwaresystems ist entscheidend für die Qualität des Endprodukts.

Die objektorientierte Programmierung ist darauf ausgelegt, Gegenstände des Anwendungsbereichs auf Objekte des Softwaresystems abzubilden. In vielen Fällen stellt dies eine intuitive und passende Modellierung des Gegenstandes dar. So fällt beispielsweise die Modellierung eines Bankkontos als Objekt eines Softwaresystems gewünscht leicht.

Allerdings lassen sich nicht alle Gegenstände des Anwendungsbereichs zufriedenstellend abbilden. Wert-artige Abstraktionen wie beispielsweise ein Geldbetrag passen nicht auf den Objekt-Begriff der objektorientierten Programmierung. Objektorientierte Sprachen haben bisher keine Mittel zur direkten Umsetzung von Werten als Teile des Anwendungssystems. Werte müssen als Objekte mit eingeschränkten Eigenschaften modelliert werden. [RS11, Kapitel 3]

Die funktionale Programmierung hingegen ist auf den Umgang mit Werten ausgelegt. Werte wie beispielsweise Geldbeträge lassen sich also leicht abbilden. Dafür fehlt in funktionalen Programmiersprachen ein Mittel zur direkten Abbildung von Objekten wie beispielsweise Bankkonten. [Mac81]

Viele Arbeiten ([BRS+98, Mac81, RS11]) sehen in Werten und Objekten zwei fundamental verschiedene Konzepte und messen beiden eine Relevanz für die Modellierung und Implementierung bei. Es wird erwartet, dass die explizite Unterstützung und die Trennung von Werten und Objekten den Entwicklungsprozess erleichtert und zu verständlicherem und robusterem Programmcode führt.

Die Programmiersprache Scala hat sich zum Ziel gesetzt, funktionale und objektorientierte Programmierung zu verbinden. Es stellt sich die Frage, ob die Kombination der beiden Programmierstile zu einer Sprache geführt hat, in der sowohl Werte als auch Objekte unterstützt werden.

Diese Bachelorarbeit beschäftigt sich mit der Umsetzung von Werttypen in Scala und untersucht die folgenden Fragestellungen:

- Wie können Werttypen in Scala umgesetzt werden?
- Wie gut werden die Eigenschaften von Werttypen abgebildet?
- Welche Anforderungen an die Implementierung von Werttypen werden erfüllt?

1.2 Wissenschaftlicher Diskussionsstand

Der Arbeitsbereich Softwaretechnik der Universität Hamburg beschäftigt sich seit langem mit dem Unterschied von Werten und Objekten sowie der Modellierung und Implementierung von Werten in objektorientierten Programmiersprachen. Insbesondere die Untersuchungen von Beate Ritterbach im Zuge ihrer Dissertation über Werttypen stellen eine Basis für diese Bachelorarbeit dar.

In diesem Kontext wurden die charakteristischen Eigenschaften von Werten und Objekten untersucht und ihre Unterschiede und Gemeinsamkeiten dargestellt. [Rit03] Der Fokus der aktuellen Forschung liegt auf den Mitteln zur Abbildung von Werttypen in objektorientierten Programmiersprachen. Dazu wurde die Unterstützung von Werttypen in verschiedenen Sprachen (Java, C++, C#) untersucht sowie Programmiermuster zur Umsetzung der Werttypen in diesen Sprachen erarbeitet. [Win10, BRS+98] Die bei diesen Untersuchungen festgestellten Schwierigkeiten resultieren in der Forderung nach einer expliziten Sprachunterstützung. [Rat06, RS11]

1.3 Aufbau der Arbeit

Kapitel 2 stellt mit der Definition von Werttypen und dem Vergleich zu Objekttypen die begriffliche Grundlage dieser Arbeit dar. In Kapitel 3 werden Anforderungen an die Umsetzung von Werttypen formuliert. Kapitel 4 gibt einen Überblick über die Programmiersprache Scala mit den für diese Arbeit relevanten Sprachmitteln.

Die Untersuchung der Sprachmittel in Kapitel 5 stellt mögliche Umsetzungen von Werttypen in Scala vor und untersucht diese im Hinblick auf die formulierten Anforderungen. Programmierrichtlinien, die bei der Umsetzung von Werttypen einzuhalten sind, und weitere Verfahren, die für die Umsetzung hilfreich sind, werden in Kapitel 6 vorgestellt. Kapitel 7 fasst die Stärken und Schwächen von Scala im Hinblick auf die Anforderungen an die Werttyp-Umsetzungen zusammen. Kapitel 8 formuliert ein Fazit.

2 Werttypen

2.1 Werte als Abstraktionen

Beispiele für Werte sind schnell gefunden. Die Mathematik und Physik enthalten mit Zahlenmengen und Größenangaben die wohl wichtigsten Werte. Auch in vielen anderen Bereichen sind Beispiele wie Datum, Geldbetrag oder Farbe zu finden. [Rit03, S. 1] Diese Beispiele dienen dem Verständnis von Werten, können aber nicht das Konzept eines Wertes an sich beschreiben. Zur Beantwortung der Frage, welches Konzept Werte in der Softwareentwicklung darstellen, betrachtet MacLennan mathematische Größen wie die natürlichen Zahlen.

Natürliche Zahlen dienen zur Beschreibung verschiedenster Zusammenhänge. So kann die Zahl 20 das Alter eines Menschen genauso wie die Anzahl von Personen in einem Raum beschreiben. In beiden Fällen ist 20 dieselbe Zahl unabhängig vom Kontext ihrer Verwendung. Auch die Operationen auf Zahlen sind unabhängig vom Kontext. Ob die Person 5 Jahre älter wird oder 5 Personen den Raum betreten, in beiden Fällen wird die 25 mit der Addition von 20 und 5 verknüpft. Diese Aussagen sind deshalb möglich, weil Zahlen Abstraktionen sind, die über den Kontext ihrer Verwendung abstrahieren. Die Mathematik nutzt genau dies und erforscht die Eigenschaften und Zusammenhänge von Zahlen isoliert. Nach MacLennan gilt dies für alle Werte, weil alle Werte Abstraktionen entsprechen. [Mac81, S. 1f]

Die Möglichkeit, Werte unabhängig von ihrem Kontext zu betrachten, ist auch für die Softwareentwicklung von Bedeutung. Damit muss ein Werttyp nur einmal implementiert werden und kann danach in verschiedenen Kontexten verwendet werden. Viele Programmiersprachen (wie Java, C++, C# und Scala) enthalten beispielsweise Zahlenwerte als primitive Datentypen. Die primitiven Datentypen bilden unter anderem die Basis von komplexeren Objekten.

Die Charakterisierung von Werten als Abstraktionen ist sehr allgemein und verdeutlicht nicht die Unterschiede von Werten und Objekten. Vor allem, weil Objekte auch Abstraktionen sein können. Für einen konkreteren Überblick werden im Folgenden die unterschiedlichen Eigenschaften von Werten und Objekten sowie ihre Gemeinsamkeiten beschrieben.

2.2 Eigenschaften von Werttypen

Basierend auf dem Wert-Begriff von MacLennan [Mac81] werden in verschiedenen Arbeiten [Rit03, Rit04, RS11, BRS+98] Eigenschaften von Werttypen beschrieben. Die genannten Arbeiten beschreiben eine Reihe von Eigenschaften, die zum Verständnis von Werten hilfreich sind. Für die Definition von Werttypen [Rit14] ist aber eine Teilmenge der Eigenschaften ausreichend.

Im Folgenden wird die Definition von Werttypen vorgestellt und anschließend sowohl die für die Definition verwendeten als auch weitere Eigenschaften von Werttypen erläutert. Für ein besseres Verständnis werden alle Eigenschaften an den Beispielen „natürliche Zahl“ und „Datum“ veranschaulicht.

Der Werttyp-Begriff setzt auf den allgemeinen Typ-Begriff auf, der eine Menge von zugehörigen Elementen und ihre Operationen beschreibt. Der Werttyp ist ein Spezialfall des allgemeinen Typs. Er beschreibt eine Menge von zugehörigen Werten und ihre Wert-Operationen. Diese unterliegen zusätzlich den folgenden Regeln:

Werttyp Definition

Nach [Rit14, Kapitel 3]:

- Werte sind *unveränderbar*, werden *nicht erzeugt* und *nicht vernichtet*.
- Wert-Operationen sind *referenziell transparent* und haben *keine Seiteneffekte*.

Die Unveränderbarkeit und Unerzeugbarkeit von Werten kann aus der referenziellen Transparenz und der Seiteneffektfreiheit der Wert-Operationen abgeleitet werden. Deshalb würde es ausreichen, in der Definition die Eigenschaften der Wert-Operationen aufzuführen. Die Definition über die vier Eigenschaften dient der Anschaulichkeit. [Rit14, Kapitel 3]

Referenzielle Transparenz

Wert-Operationen sind referenziell transparent. [Mac81, S. 1], [RS11, S. 113], [BRS+98, S. 14] Eine Operation ist per Definition dann referenziell transparent, wenn ihr Ergebnis ausschließlich von ihren Operanden abhängt. Eine referenziell transparente Operation liefert für die selben Operanden immer dasselbe Ergebnis. [Rit14, Kapitel 3]

Für Operationen auf Zahlen und Daten entspricht diese Regel unserer Intuition. So ergibt die Differenz zweier Daten immer dieselbe Anzahl von Tagen unabhängig davon, ob die Zeit zwischen Projektbeginn und aktuellem Datum oder die Zeit zwischen zwei Prüfungsterminen berechnet wird. $20.02.2013 - 10.02.2013$ entspricht immer 10 Tagen. Aus der referenziellen Transparenz ergibt sich auch die Möglichkeit der Termersetzung. So kann in jedem Ausdruck $(3 * (7 + 4) = 33)$ ein beliebiger Term durch sein Ergebnis $(7 + 4 = 11)$ ersetzt werden, ohne das Ergebnis des Gesamtausdrucks zu beeinflussen $(3 * 11 = 33)$. Dies ist nur dann möglich, wenn sichergestellt ist, dass ein Term immer zum selben Ergebnis evaluiert.

Seiteneffektfreiheit

Wert-Operationen sind seiteneffektfrei. [RS11, S. 113], [Rit03, S. 1] Eine Operation ist per Definition dann seiteneffektfrei, wenn sie keine Zustandsänderungen bewirkt. Wert-Operationen ändern also weder den Zustand ihrer Operanden, noch den von anderen Objekten.

Operationen, wie die Addition von natürlichen Zahlen $(1+2)$, berechnen ausschließlich den Ergebnis-Wert (3) der Operation und haben ansonsten keinen Effekt. Sie ändern weder den Zustand eines der Operanden (1 oder 2), noch den Zustand eines Objekts aus ihrem Kontext.

Unveränderbarkeit

Werte sind unveränderbar. [Mac81, S. 2], [RS11, S. 113], [Rit04, S. 10], [Rit03, S. 1], [BRS$^+$98, S. 14] Unveränderbarkeit von Werten bedeutet, dass es keine Operation gibt, die eine Änderung eines Wertes bewirkt. Die Unveränderbarkeit von Werten ergibt sich aus der Seiteneffektfreiheit der Wert-Operationen. Wenn die Operationen keine Zustandsänderungen bewirken, kann durch eine Operation auch kein Wert verändert werden. [Rit14, Kapitel 3]

Die Addition zweier Zahlen $(1 + 2)$ ändert keine der Zahlen (1 oder 2) in das Ergebnis (3), sondern bildet zwei Zahlen auf ihr Ergebnis $(1 + 2 = 3)$ ab. Operationen auf Werten ändern also nicht die Werte, sondern bilden Werte aufeinander ab. So ist auch die Addition einer Anzahl von Tagen zu einem Datum ($10.02.2013 + 10$ Tage) eine Abbildung der beiden Werte auf das Ergebnis ($20.02.2013$). Vor allem ändert die Operation nicht den $10.02.2013$ in den $20.02.2013$.

Unerzeugbarkeit

Werte können nicht erzeugt und nicht zerstört werden, sie existieren per se. [Mac81, S. 1], [RS11, S. 113], [Rit04, S. 10], [Rit03, S. 1], [BRS$^+$98, S. 14] Erzeugen bedeutet, etwas Neues zu erschaffen. Würde es Operationen geben, die Werte erzeugen, so würde jeder Aufruf der Operation einen anderen Wert liefern, selbst wenn die Operanden dieselben wären. Diese Operationen würden gegen die Eigenschaft der referenziellen Transparenz verstoßen. Deshalb können Werte nicht erzeugt werden. [Rit14, Kapitel 3]

Die Unzerstörbarkeit von Werten folgt aus ihrer Unerzeugbarkeit und der referenziellen Transparenz. Könnte ein Wert w zerstört werden, so würde der Aufruf einer Wertoperation $\otimes$, der w als Ergebnis liefert ($a \otimes b = w$) diesen Wert erzeugen. Da dies der Unerzeugbarkeit von Werten widerspricht, können Werte nicht zerstört werden.

Der Operations-Aufruf $1 + 2$ erzeugt nicht das Ergebnis 3. Die 3 existiert schon vor der Operation. Auch kann die 3 nicht zerstört werden, weil wegen der referentiellen transparenz $1 + 2$ weiterhin 3 ergeben muss. Mit einem Datum verhält es sich genauso. Es gibt nur genau einen $20.02.2013$, der unabhängig davon existiert, ob er Ergebnis eines Operations-Aufrufs war.

Feste Wertmenge

Die Wertmenge (auch bezeichnet als Werte-Universum) eines Werttyps ist festgelegt. [RS11, S. 113] Die feste Wertmenge ist gleichbedeutend mit Unerzeugbarkeit und Unzerstörbarkeit von Werten. Es können weder Werte hinzugefügt noch entfernt werden.

Es ist nicht möglich eine neue natürliche Zahl zu finden, die vorher noch nicht existierte. Dasselbe gilt für ein Datum. Entweder gehört ein Wert zur Wertmenge oder nicht.

Zeitlosigkeit

Werte werden als zeitlos beschrieben. [Mac81, S. 1], [Rit04, S. 9], [BRS$^+$98, S. 14] Zeitlosigkeit bezieht sich auf verschiedene Aspekte von Werten. Zum einen hat die Zugehörigkeit zur Wertmenge keine zeitliche Komponente, weil die Wertmenge fest ist. Zum anderen sind auch einzelne Werte der Wertmenge zeitlos in dem Sinne, dass sie keinen Lebenszyklus haben. Werte werden nicht erzeugt, nicht verändert und nicht zerstört. Es macht demnach keinen Sinn, Werte zu verschiedenen Zeitpunkten zu betrachten.

Zahlen wie Daten existieren als Konzept unabhängig von dem Zeitpunkt, an dem die natürlichen Zahlen entdeckt wurden oder zum ersten Mal von einem Datum gesprochen wurde.

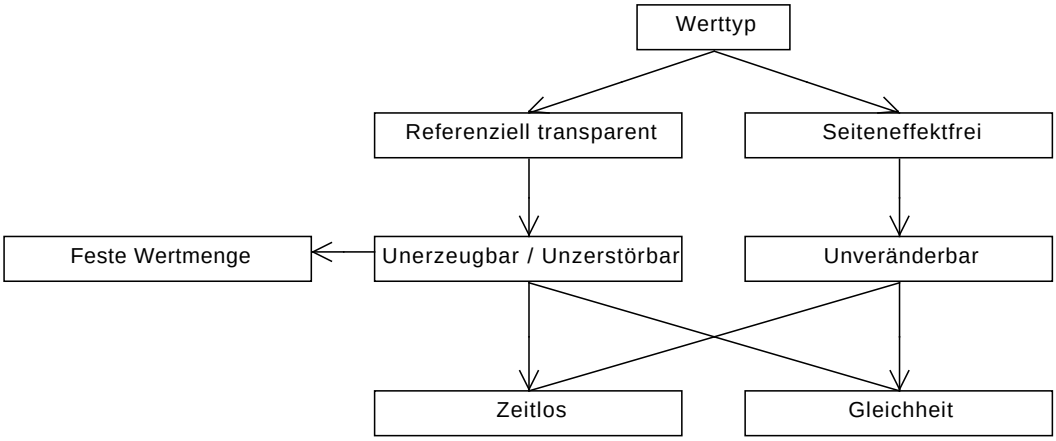

Abbildung 2.1: Abhängigkeiten der Eigenschaften

Gleichheit

Die Gleichheit von Werten ist das Kriterium für ihre Unterscheidung. [Mac81, S. 2], [Rit04, S. 21], [Rit03, S. 2], [RS14, S. 3 ff] Es werden die zwei Vergleiche Identität und Gleichheit unterschieden. Nach [RS14, S. 3] ist nur Gleichheit auf Werte anwendbar. Wert-Gleichheit bedeutet, dass sich zwei Ausdrücke auf denselben (abstrakten) Wert beziehen. Identität dagegen hat für Werte keine Bedeutung, weil die Unterscheidung von konzeptuell gleichen Werten (mit unterschiedlicher Repräsentation) im Widerspruch zur referenziellen Transparenz steht.

Das Ergebnis der Operation $\frac{3}{12} + \frac{3}{12}$ könnte als $\frac{6}{12}$ genauso wie als $\frac{1}{2}$ dargestellt werden. Beide Darstellungen sind korrekt und repräsentieren denselben (abstrakten) Wert. Wären diese Repräsentationen als Werte unterscheidbar, so wäre die Operation nicht referenziell transparent. Auch die Datums-Repräsentationen $20.02.2013$ (Deutsch) und $02/20/2013$ (Amerikanisch) repräsentieren das selbe Datum.

Abbildung 2.1 fasst die behandelten Eigenschaften zusammen und stellt ihre Abhängigkeiten dar. Ein Pfeil von Eigenschaft a zu b bedeutet, dass b aus a hergeleitet werden kann. Mehrere eingehende Pfeile bedeuten, dass die Eigenschaft durch alle eingehenden Eigenschaften zusammen hergeleitet werden kann.

2.3 Eigenschaften von Objekttypen

Ein Objekttyp beschreibt eine Menge von zugehörigen Objekten und ihre Operationen. Dieser Abschnitt beschreibt die Eigenschaften, in denen sich Objekttypen von Werttypen unterscheiden.

Referenzielle Opazität

Objekt-Operationen können im Vergleich zu Wert-Operationen referenziell opak sein. [RS11, S. 119] Referenzielle Opazität ist das Gegenteil von referenzieller Transparenz. Eine referenziell opake Operation kann demnach für die selben Operanden verschiedene Ergebnisse liefern. Ein Beispiel dafür ist die Operation $age()$ am Objekttyp $Person$, die das Alter einer Person zurück gibt. Diese Operation wird, bedingt durch den Alterungsprozess der Person, verschiedene Ergebnisse zurück geben, obwohl der Operand gleich bleibt.

Seiteneffekte

Objekt-Operationen können Seiteneffekte hervorrufen. [Mac81, S. 3], [RS11, S. 119], [BRS+98, S. 9] Seiteneffektbehaftete Operationen ändern den Zustand ihrer Operanden oder anderer Objekte. So ist beispielsweise der einzige Zweck von Operationen ohne Rückgabewert (Operationen mit Rückgabetyp $void$ in Java) Seiteneffekte auszuführen.

Änderbarkeit

Objekte sind änderbar. [Mac81, S. 3], [BRS+98, S. 15] Die Änderung von Objekten geschieht durch seiteneffektbehaftete Objekt-Operationen, die den Zustand des Objekts ändern. Ein Beispiel dafür ist die Operation $setSalary(...)$ am Objekttyp $Person$, die den Zustand eines Personen-Objekts ändert, indem sie sein Gehalt ändert.

Erzeugbarkeit

Objekte werden erzeugt. [Mac81, S. 3], [BRS+98, S. 15] Jeder Objekttyp hat einen Konstruktor, der beim Aufruf ein neues Objekt dieses Typs erzeugt. Entsprechend zum Erzeugen können Objekte auch zerstört werden. Diese Zerstörung kann in einigen Programmiersprachen (C++) vom Entwickler aufgerufen werden. In anderen (Java) werden Objekte automatisch zerstört, wenn sie nicht mehr gebraucht werden.

Objekte existieren also in der Zeit und haben einen Lebenszyklus, der die Erzeugung, mögliche Veränderungen über die Zeit und die Zerstörung des Objektes beinhaltet.

Identität

Die Identität von Objekten ist das Kriterium für ihre Unterscheidung. [Mac81, S. 3], [Rit04, S. 21], [Rit03, S. 2], [RS14, S. 3 ff] Objekt-Identität bedeutet, dass sich zwei Ausdrücke auf dasselbe Objekt beziehen. Diese Definition drückt das gleiche aus wie Wert-Gleichheit für Werte. Deshalb werden Wert-Gleichheit und Objekt-Identität als dasselbe Konzept angesehen. [RS14, S. 5]

Die Gleichheit von Objekten wird hingegen als die Übereinstimmung in „relevanten“ Merkmalen beschrieben. Bei zwei „gleichen“ Objekten handelt es sich also im Allgemeinen nicht um dasselbe Objekt. Die verglichenen Objekte sind sich lediglich ähnlich. Damit hängt die Gleichheit von der willkürlichen Festlegung der Merkmale ab und unterscheidet sich je nach Anwendungskontext. [RS14, S. 3]

2.4 Gemeinsame Eigenschaften

Werttypen und Objekttypen haben auch gemeinsame Eigenschaften. Diese ermöglichen, dass Werte und Objekte in vielen Kontexten gleichartig behandelt werden können.

Typen

Sowohl Objekttypen als auch Werttypen setzen auf dem allgemeinen Typ-Begriff auf. Sie beschreiben ihre Exemplare (Objekte und Werte) und ihre Operationen (Objekt- und Wert-Operationen). Die jeweiligen Operationen werden unabhängig von einer Implementierung beschrieben. Werttypen und Objekttypen können mit Hilfe von abstrakten Datentypen beschrieben werden. [Rit04, S. 11], [Rit03, S. 2]

Benutzung

Objekte und Werte können in vielen Fällen gleich benutzt werden. Sie können als Parameter oder Rückgabewert von Operationen auftreten, an globale und lokale Variablen gebunden werden und als generische Parameter verwendet werden oder Elemente von Sammlungen ($Collections$) sein. [Rit04, S. 11]

Werte	Objekte
Referenziell transparent und unerzeugbar	Referenziell opak und erzeugbar
Seiteneffektfrei und unveränderbar	Seiteneffektbehaftet und veränderbar
Zeitlos	Lebenszyklus
Gleichheit	Identität
Elemente eines Typs	
Gleiche Benutzung	
Bestandteil des Anwendungsbereichs	

Tabelle 2.1: Eigenschaften von Werten und Objekten

Fachlichkeit

Werte sowie Objekte können Bestandteile des modellierten Anwendungsbereichs und damit fachlich motiviert sein. [Rit04, S. 11]

Als Zusammenfassung zeigt Tabelle 2.1 einen Überblick der wichtigsten Eigenschaften von Werten und Objekten. Im ersten Teil der Tabelle sind die unterschiedlichen Eigenschaften einander gegenübergestellt. Die gemeinsamen Eigenschaften sind im unteren Teil der Tabelle dargestellt.

Die Eigenschaften von Werttypen und Objekttypen bilden die Grundlage zur Umsetzung von Werttypen in Scala. Neben den konzeptionellen Eigenschaften ergeben sich bei der Implementierung weitere Anforderungen. Diese werden im folgenden Kapitel diskutiert.

3 Anforderungen an Werttyp-Umsetzungen

Das Ziel eines Verfahrens zur Umsetzung von Werttypen ist die Berücksichtigung ihrer charakteristischen Eigenschaften. Aus softwaretechnischer Sicht ergeben sich weitere Anforderungen an ein solches Verfahren. Ritterbach ([Rit14, Kapitel 5]) untersucht verschiedene Verfahren zur Abbildung von Werttypen in objektorientierten Programmiersprachen und beurteilt deren Eignung anhand der Kriterien Klarheit, Einfachheit, Sicherheit und Effizienz. Auch Bäumer et al. ([BRS+98, S. 17 ff]) untersuchen unter anderem die Klarheit und Effizienz einiger Verfahren. In [RS11, S. 116] werden Klarheit, Einfachheit und Sicherheit als Anforderungen für eine Sprachunterstützung für Werttypen genannt. Auch die Möglichkeit zur Erweiterung der Wert-Operationen eines Werttyps ist ein Kriterium.

In diesem Kapitel werden *Klarheit*, *Einfachheit*, *Sicherheit* und *Erweiterbarkeit* als Kriterien für Werttyp-Umsetzungen betrachtet[1]. Was unter diesen Kriterien verstanden wird und welche konkreten Anforderungen hinter ihnen stehen, soll im Folgenden beschrieben werden. Zur Verdeutlichung der Anforderungen wird das folgende Beispiel der Umsetzung eines Werttyps in Java mit dem *Value Object Pattern* ([Rie06]) betrachtet.

3.1 Beispiel

Dieses Beispiel skizziert die Programmierung eines Typs $Money$ in Java mit Hilfe des Value Object Pattern. Der Typ soll eine simple Abbildung eines Geldbetrags, repräsentiert durch Betrag und Währung, darstellen. Die Operationen sind der Einfachheit halber auf die Addition von Geldbeträgen beschränkt.

Das Value Object Pattern ist ein Verfahren zur Abbildung von Werttypen. Es setzt Werttypen als unveränderbare Klassen um[2]. Die Wertoperationen geben als Berechnungsergebnis eine neue Instanz einer Wert-Klasse zurück, anstatt eine bestehende zu ändern.

[1] Das Kriterium Effizienz wird nicht betrachtet, weil es von der konzeptionellen Sicht auf Werte zu weit wegführt.

[2] Das Value Object Pattern verwendet nicht denselben Werttyp-Begriff wie diese Arbeit. Vor allem die Unerzeugbarkeit ist nicht Teil des zugrundeliegenden Wertbegriffs.

Durch diese Umsetzung können mehrere Instanzen des konzeptionell selben Wertes existieren. Deshalb müssen zusätzlich die Methoden $equals$ und $hashCode$ implementiert werden und Werte dürfen nur mit $equals$ verglichen werden. Listing 3.1 zeigt eine mögliche Implementierung des Geldbetrags.

```
public class Money {
  public final double amount;
  public final Currency currency;

  public Money(double amount, Currency currency) {
    this.amount = amount;
    this.currency = currency;
  }

  public Money plus(Money other){
    assert currency == other.currency;
    return new Money(amount + other.amount, currency);
  }

  public int hashCode() {
    long temp = Double.doubleToLongBits(amount);
    int result = 31 + (int) (temp ^ (temp >>> 32));
    return 31 * result + ((currency == null) ? 0 : currency.hashCode());
  }

  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;
    Money other = (Money) obj;
    if (Double.doubleToLongBits(amount) != Double
      .doubleToLongBits(other.amount))
      return false;
    if (currency != other.currency)
      return false;
    return true;
  }
}
```

```
public enum Currency{
  EUR, USD
}
```

Listing 3.1: Geldbetrag mit Value Object Pattern in Java

Der Geldbetrag wird durch Felder für Betrag ($amount$) und Währung ($currency$) repräsentiert. Zur Repräsentation der Währung wurde ein Aufzählungstyp ($enum$) mit Werten für Euro und Dollar verwendet. Beide Felder sind öffentlich und unveränderbar ($public$, $final$). Mit dem Konstruktor ($Money$) können Instanzen der Klasse $Money$ erzeugt werden. Die Operation $plus$ addiert zwei Geldbeträge und gibt das Ergebnis als neue Instanz zurück. Außerdem wurden $equals$ und $hashCode$ implementiert, um Geldbeträge zu vergleichen.

Im Folgenden wird die Bedeutung der Kriterien Klarheit, Einfachheit, Sicherheit und Erweiterbarkeit erläutert und die Anforderungen anhand dieses Beispiels motiviert.

3.2 Klarheit

Klarheit bedeutet für die Umsetzung eines Werttyps, dass dieser im Programmcode eindeutig als solcher erkennbar ist. Diese Forderung ist beim Benutzen des Werttyps genauso wie beim Ändern des Werttyps essentiell. Erkennt ein Entwickler den Typ nicht als Werttyp, so kann er bei Änderung der Klasse leicht die Eigenschaften von Werttypen verletzen. Bei einem Benutzer des Typs kann es bei Nichterkennen als Werttyp zu einer falschen Verwendung kommen. Listing 3.2 zeigt Beispiele für Fehler bei der Verwendung.

```
1 Money balance1 = new Money(100, Currency.EUR) // 100 EUR
2 balance1.plus(new Money(100, Currency.EUR)) // ohne Wirkung - Fehler
3 balance1 = balance1.plus(new Money(100, Currency.EUR)) // 200 EUR
4 Money balance2 = new Money(200, Currency.EUR) // 200 EUR
5 if (balance1 == balance2) ... // false - Fehler
6 if (balance1.equals(balance2)) ... // true
```

Listing 3.2: Verwendung des Geldbetrags

Ein Fehler bei der Verwendung des Geldbetrags ist in Zeile 2 zu sehen. Der isolierte Aufruf von $plus$ ohne eine Zuweisung des Ergebnisses hat keine Wirkung. Zeile 3 zeigt die korrekte Verwendung von $plus$. Einen weiteren Fehler zeigt Zeile 5. Der Vergleich von Geldbeträgen mit $==$ führt zu einem unerwarteten Ergebnis. Werte, die mit dem Value Object Pattern umgesetzt sind, dürfen wie in Zeile 6 nur mit $equals$ verglichen werden.

In diesem Beispiel ist die Umsetzung des Geldbetrags nicht als Werttyp erkennbar. Dies hat verschiedene Gründe. Zum einen wird bei der Definition des Werttyps mit $class$ das gleiche Schlüsselwort wie bei der Definition von Objekttypen verwendet. Der Entwickler kann die beiden Typen nur anhand der charakteristischen Eigenschaften unterscheiden. Der Unterschied von Werten und Objekte ist allerdings nicht jedem Entwickler bewusst. Zum anderen steht die Erzeugung von Geldbeträgen mit dem Schlüsselwort new im Widerspruch zur Unerzeugbarkeit von Werten. Auch die Verwendung des Geldbetrags mit der Operation $plus$ und dem Vergleich per $equals$ ähnelt kaum der Verwendung der vordefinierten Werttypen wie int (primitive Datentypen). Ein Operator wie "+" und der Vergleich mit $==$ würden Geldbeträge den vorhandenen Werttypen näher bringen.

3.3 Einfachheit

Die Umsetzung eines Werttyps sollte mit einer leichtgewichtigen Syntax möglich sein. Dabei sollte die Komplexität des Programmcodes der Komplexität des abgebildeten Werttyps angemessen sein.

Mit den Methoden $equals$ und $hashCode$ müssen im Beispiel des Geldbetrags zwei Methoden implementiert werden, die leicht vergessen werden können und deren Implementierung fehleranfällig ist. [RS14, S. 1] Diese zusätzliche Komplexität ist für Werttypen aber nicht immer nötig. Nach [RS14] muss der Entwickler die Gleichheit in vielen Fällen nicht implementieren. Die Methoden $equals$ und $hashCode$ können oft generiert werden, sodass sie die Gleichheit von Werttypen korrekt abbilden.

Die übrige Komplexität des Beispiels ist gering. Allerdings berücksichtigt das Beispiel die Unerzeugbarkeit von Geldbeträgen nicht. Diese Eigenschaft kann beispielsweise durch den Einsatz des Entwurfsmusters *Flyweight* von Gamma et al. ([GHJV95, S. 195]) umgesetzt werden. Bei diesem Verfahren muss, anstatt einen Wert zu erzeugen, eine Fabrikmethode aufgerufen werden, die den gewünschten Wert zurückgibt. Die Fabrik verwaltet die Werte an zentraler Stelle und stellt sicher, dass zu jedem Wert nur eine Instanz der Wert-Klasse existiert. Die Verwaltung der Instanzen ist komplex und wird durch Synchronisation in verteilten Systemen noch erschwert. [Rit14, Kapitel 5]

Die Umsetzung von Werttypen soll nicht durch die Komplexität der technischen Umsetzung erschwert werden. Dazu gehört, dass die Eigenschaften von Werttypen ohne großen Aufwand umgesetzt werden können.

3.4 Sicherheit

Das Kriterium Sicherheit bezieht sich auf die Implementierung von Werttypen sowie auf deren Benutzung.

3.4.1 Benutzung

Wie im Abschnitt 3.2 beschrieben, können Fehler bei der Verwendung von Werttypen zur Laufzeit ein unerwartetes Verhalten zur Folge haben. Diese Fehler werden vom Compiler nicht erkannt. Listing 3.3 zeigt zwei semantische Fehler.

```
1 Money balance = new Money(100, Currency.EUR);
2 balance.plus(new Money(100, Currency.EUR)) // keine Wirkung - Fehler
3 if (balance.equals(Currency.EUR)) ... // immer false - Fehler
4 if (balance.currency.equals(Currency.EUR)) ...
```

Listing 3.3: Verwendung des Geldbetrags

Der Fehler in Zeile 2 ist schon aus Abschnitt 3.2 bekannt. Allgemein ist der Aufruf einer Wert-Operation ohne die Verwendung des Ergebnisses wirkungslos. Die 3. Zeile enthält einen weiteren Fehler. Mit $equals$ können zwei beliebige Exemplare verglichen werden. Damit sind auch Vergleiche von Exemplaren mit inkompatiblen Typen zulässig. Inkompatibel sind Typen, deren Exemplare nicht gleich sein können. Insbesondere Werttypen und Objekttypen sind in diesem Sinne inkompatibel. Ihr Vergleich liefert wie auch der Vergleich eines Geldbetrags mit einer Währung im Beispiel immer $false$. Dieser Fehler kann leicht entstehen, wenn der Entwickler statt der korrekten Zeile 4 versehentlich den Ausdruck aus Zeile 3 schreibt.

Um die Sicherheit im Umgang mit Werttypen zu erhöhen, sollte eine Werttyp-Umsetzung semantisch falsche Verwendungen von Wert-Operationen verhindern.

3.4.2 Implementierung

Bei der Implementierung von Werttypen bedeutet Sicherheit, dass die charakteristischen Eigenschaften von Werttypen (referenzielle Transparenz und Seiteneffektfreiheit) *garantiert* werden. Diese Garantie würde den Entwickler davor schützen, unbeabsichtigt die Werttyp-Eigenschaften zu verletzen.

Im Beispiel des Geldbetrags können mit der Operation $plus$ nur Beträge gleicher Währung addiert werden. Um die Operation auf beliebige Additionen zu erweitern, könnte man über den Wechselkurs die Beträge umwandeln und dann addieren. Listing 3.3 zeigt eine entsprechende Implementierung von $plus$.

```
1 class ExchangeRates{
2   public static double getRate(Currency from, Currency to) {...}
3 }
4
5 public Money plus(Money other){
6   double exchangeRate = ExchangeRates.getRate(other.currency, currency);
7   return new Money(amount + (other.amount * exchangeRate), currency);
8 }
```

Listing 3.4: Referenziell opake Implementierung

Auch wenn die Addition von beliebigen Geldbeträgen eine wünschenswerte Funktionalität ist, wird durch diese Implementierung die referenzielle Transparenz der Operation zerstört. Um den Wechselkurs für die Währungen zu erhalten, ruft die Operation $plus$ eine statische Methode am Objekt $ExchangeRates$ auf. Sobald sich der Wechselkurs ändert, wird die Operation $plus$ für dieselben Parameter ein anderes Ergebnis liefern.

Die Anforderung, die Werttyp-Eigenschaften zu garantieren, erfordert ein komplexes Regelwerk für Wert-Operationen, welches festlegt, was in der Implementierung von Wert-Operationen zulässig ist. Beispielsweise müsste geregelt werden, wie und in welchen Fällen Wert-Operationen auf Objekte zugreifen dürfen.

Um dieses Regelwerk zu vereinfachen, wird in [RS11, s. 119f] vorgeschlagen, dass Werttypen keinen Zugriff auf Objekte haben. Weil Werte nicht auf Objekte zugreifen können, Objekte aber Werte benutzen können, besteht ein asymmetrisches Abhängigkeitsverhältnis zwischen Werten und Objekten.

> Im Ergebnis wird dadurch jedes System unterteilt in einen „funktionalen Kern“ und eine „objektorientierte Schale“. [RS11, S. 119]

Der funktionale Kern besteht aus den Werttypen und ist unabhängig von der objektorientierten Schale. Die objektorientierte Schale beinhaltet die Objekttypen. Diese haben freien Zugriff auf die Werttypen. Abbildung 3.1 zeigt dieses asymmetrische Abhängigkeitsverhältnis. [RS11, S. 119]

Eine Werttyp-Umsetzung sollte Sicherheit für die Implementierung und die Benutzung bieten. Dazu gehört die Verhinderung von semantisch falscher Verwendung und die Garantie der Werttyp-Eigenschaften.

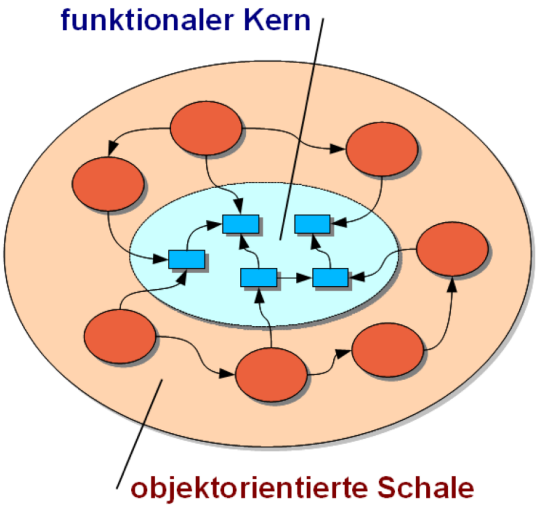

Abbildung 3.1: Funktionaler Kern und objektorientierte Hülle

3.5 Erweiterbarkeit

Unter Erweiterbarkeit wird im Rahmen dieser Arbeit die Erweiterung eines Werttyps um Wert-Operationen verstanden. Je nach Anwendungsbereich werden an die Schnittstelle eines Werttyps andere Anforderungen gestellt.

Bezogen auf das Beispiel können die Addition und Subtraktion von Geldbeträgen für die Umsetzung einer simplen Kontoklasse ausreichend sein. Eine Anwendung, die sich beispielsweise mit Zinsrechnung befasst, würde hingegen die Möglichkeit zur Multiplikation eines Geldbetrags mit einer Zahl (Zinssatz) benötigen.

Wird der gleiche Werttyp in verschiedenen Anwendungsbereichen genutzt, entstehen häufig neue Anforderungen an die Wert-Operationen. Wenn die Möglichkeit besteht, den Programmcode des Werttyps anzupassen und diesen um Wert-Operationen zu erweitern, ist die Erweiterbarkeit gegeben. Vielfach verwendete Werttypen wie Zahl, Zeichenkette und Datum werden vom Anwendungsentwickler häufig nicht selbst implementiert, sondern aus einer Bibliothek in die Anwendung eingebunden. So eingebundene Werttypen können nicht mehr im Programmcode erweitert werden. Um auch diese Werttypen an die Anwendung anzupassen, sollte es Möglichkeiten geben, einen Werttyp um Wert-Operationen zu erweitern, ohne dessen Programmcode zu ändern.

Bevor in Kapitel 5 mithilfe dieser Kriterien Werttyp-Umsetzungen in Scala analysiert werden, gibt das folgende Kapitel eine Einführung in die Programmiersprache Scala.

4 Scala

4.1 Was ist Scala

Scala wurde an der École Polytechnique Fédérale de Lausanne (EPFL) unter der Leitung von Martin Odersky entwickelt und 2003 in der ersten Version veröffentlicht. [OAC+04] Der Name Scala ist ein Akronym für *Scalable Language*. Der Name drückt das Ziel aus, eine Sprache zu entwickeln, die für eine große Bandbreite von Aufgaben, vom Schreiben kleiner Skripte bis hin zur Entwicklung komplexer Softwaresysteme, geeignet ist. [OSV11, Kapitel 1]

Scala ist eine statisch getypte Programmiersprache mit dem Ziel, objektorientierte und funktionale Programmierung zu verbinden. Odersky beschreibt diese Verbindung folgendermaßen:

> Scala is a pure-bred object-oriented language. Conceptually, every value is an object and every operation is a method-call. [...] Scala is also a full-blown functional language. It has everything you would expect, including first-class functions, a library with efficient immutable data structures, and a general preference of immutability over mutation. [Ode]

Scala und Java haben einige Ähnlichkeiten. Scala läuft auf der Java Virtual Maschine (JVM) und übernimmt einen Teil der Syntax von Java. Ein weiteres Ziel von Scala ist die Interoperabilität mit Java. Das heißt, Java-Klassen sowie Java-Bibliotheken sollen in Scala verwendet werden können und umgekehrt. [Ode, OAC+04]

Die folgende Einführung gibt einen groben Überblick über die Programmiersprache und behandelt Sprachmittel, die für dieser Arbeit relevant sind. Sie beginnt mit den Grundlagen von Variablen, Methoden, Klassen und Objekten. Es werden Operatoren, Typparameter und implizite Umwandlungen behandelt, bevor mit impliziten Parametern das Thema der Typklassen vorbereitet wird.

4.2 Einführung

4.2.1 Variablen und Methoden

Variablen

In Scala gibt es zwei Arten von Variablen. Sie werden durch die Schlüsselworte val und var eingeleitet. vals entsprechen Variablen in Java, die mit $final$ gekennzeichnet sind. Sie werden einmal initialisiert und können danach nicht neu zugewiesen werden. vars im Gegensatz dazu entsprechen Variablen in Java, die nicht mit $final$ gekennzeichnet sind. Bei vars sind erneute Zuweisungen möglich. [OSV11, Kapitel 2] Listing 4.1 zeigt die Definition dreier Variablen.

```
1 val name: String = "Peter" // unveraenderbar
2 val gender = "male"
3 var age = 25 // veraenderbar
```

Listing 4.1: Variablendefinition

Zeile 1 zeigt die Definition einer unveränderbaren Variablen (val) mit dem Namen $name$ und dem Typ $String$ und initialisiert diese mit dem Wert $Peter$. Wenn, wie in Zeile 2, der Typ der Variablen nicht explizit angegeben wird, inferiert der Compiler diesen aus dem zugewiesenen Wert. [OSV11, Kapitel 2] Der Typ der Variablen $gender$ ist $String$. Zeile 3 enthält die Definition einer veränderbaren Variablen (var).

Methoden

Methoden werden in Scala mit dem Schlüsselwort *def* eingeleitet. Es folgt der Methodenname und eine kommaseparierte Liste von Parametern. Jeder Parameter besteht aus einem Namen und dem Typ, getrennt durch einen Doppelpunkt. Parametertypen können nicht inferiert werden. Der Rückgabetyp der Methode hingegen kann in vielen Fällen inferiert werden. Eine Ausnahme bilden rekursive Methoden. In diesem Fall muss der Rückgabetyp nach der Parameterliste getrennt durch einen Doppelpunkt angegeben werden. [OSV11, Kapitel 2] Listing 4.2 zeigt die Definition einer rekursiven Methode, die den größten gemeinsamen Teiler der beiden Parameter berechnet.

```
1 def ggt(a: Int, b: Int): Int = {
2   val r = a % b
3   if(r > 0) ggt(b, r) else b
4 }
```

Listing 4.2: Methodendefinition

Der Methodenkopf, bestehend aus dem Namen, einer oder mehrerer Parameterlisten und dem Rückgabetyp, ist durch ein Gleichheitszeichen von Methodenrumpf getrennt. Der Rumpf muss in geschweiften Klammern stehen, wenn er mehr als einen Ausdruck enthält. Die Evaluierung des letzten Ausdrucks des Rumpfs wird als Ergebnis der Methode zurückgegeben. [OAC+04, Kapitel 2] Zum angeführten Beispiel ist zu bemerken, dass die Fallunterscheidung ($if - else$) in Scala im Gegensatz zu Java abhängig von der Bedingung das Ergebnis des ersten oder zweiten Blocks zurückgibt. [OSV11, Kapitel 2]

Der Aufruf einer Methode erfolgt wie in Java. Dabei werden die Argumente in der Reihenfolge der Parameterliste an die Methode übergeben. Scala bietet zusätzlich die Möglichkeit Argumente benannt zu übergeben. Listing 4.3 zeigt beide Aufrufvarianten.

```
1 val res = ggt(31, 7) // ggt(31, 7) = 1
2 val res2 = ggt(b = 7, a = 31) // ggt(31, 7) = 1
```

Listing 4.3: Methodenaufruf

Beim ersten Aufruf wird durch die Reihenfolge der Argumente bestimmt, dass die Methode mit $a = 31$ und $b = 7$ ausgeführt wird. Beim zweiten Aufruf spielt die Reihenfolge der Argumente keine Rolle. Durch die benannten Argumente wird auch hier die Methode mit $a = 31$ und $b = 7$ ausgeführt.

4.2.2 Klassen, Traits und Objekte

Klassen

Klassen in Scala entsprechen von der Funktion her den Klassen in Java. Die Definition unterscheidet sich allerdings von der in Java. Zur Veranschaulichung zeigt Listing 4.4 die Definition einer Klasse $Person$ in Java und Listing 4.5 die gleiche Klasse in Scala.

```
class Person{
  public final String name;
  private final String gender;
  public int age;

  public Person(String name, String gender, int age){
    this.name = name;
    this.gender = gender;
    this.age = age;
  }
}
```

Listing 4.4: Klasse Person in Java

Klassenparameter	Bedeutung
val name	öffentlich, unveränderbar
var name	öffentlich, veränderbar
name	privat, unveränderbar
private var name	privat, veränderbar

Tabelle 4.1: Klassenparameter und ihre Bedeutung

```
class Person(val name: String, gender: String, var age: Int)
```

Listing 4.5: Klasse Person in Scala

In Scala werden alle Felder der Klasse, die durch einen Konstruktorparameter initialisiert werden, in Klammern hinter dem Klassennamen angegeben. Diese Felder werden auch *Klassenparameter* genannt. Der Scala-Compiler generiert Felder für die Klassenparameter und definiert einen Konstruktor, der diese Felder initialisiert.

Tabelle 4.1 zeigt welche Bedeutung die Modifikatoren der Klassenparameter für die generierten Felder haben. Mit *val* gekennzeichnete Parameter (*name*) sind öffentlich und unveränderbar, mit *var* gekennzeichnete (*age*) sind öffentlich und veränderbar und Parameter ohne Kennzeichnung (*gender*) sind privat und unveränderbar. Private veränderbare Parameter werden mit den Schlüsselworten *private var* gekennzeichnet.

Neben den Klassenparametern können Klassen weitere Felder und Methoden enthalten. Diese werden im Rumpf der Klasse definiert. Auch die Definition von zusätzlichen Konstruktoren ist möglich. Wie in Java werden Klassen mit dem Schlüsselwort *new* instanziiert. [OSV11, Kapitel 1]

Traits

Traits sind eine Mischung aus Interfaces und abstrakten Klassen in Java. Ebenso wie abstrakte Klassen können Traits abstrakte und konkrete Methoden[1] enthalten und nicht instanziiert werden. Analog zu Interfaces können Klassen von mehreren Traits erben.

Die Definition eines Trait ähnelt der einer Klasse. Traits können allerdings keine Klassenparameter definieren, weil für sie kein Konstruktor definiert wird. Listing 4.6 zeigt die Definition eine Traits.

[1] Abstrakt sind Methoden, die keinen Rumpf haben.

```
1 trait Ordered[A] extends java.lang.Comparable[A]{
2   def compare(that: A): Int
3   def < (that: A) = (this.compare(that)) < 0
4   ...
5 }
```

Listing 4.6: Trait Definition

Das generische[2] Trait $Ordered$ befindet sich in der Standardbibliothek von Scala und erweitert Javas Interface $Comparable$. $Ordered$ definiert in Zeile 2 eine abstrakte Methode $compare$, die einen Vergleich mit einer Instanz des generischen Typs A vornimmt. Diese Methode entspricht der $compareTo$ Methode des Java-Interfaces. Zeile 3 zeigt eine konkrete Methode, die $compare$ in ihrer Implementierung nutzt.

Objekte

Objekte (Objects) stellen Klassen mit einer einzelnen Instanz dar. Die Definition ähnelt der von Klassen und Traits. Objekte können wie Traits keine Klassenparameter erhalten, weil sie nicht wie Klassen instanziiert werden. Außerdem können Objekte keine abstrakten Felder oder Methoden enthalten oder generisch sein. [OSV11, Kapitel 4] Listing 4.7 zeigt ein Beispiel.

```
1 trait Currency{
2   def name: String
3 }
4 object EUR extends Currency{
5   val name = "Euro"
6 }
7 ...
8 println(EUR.name) // Euro
```

Listing 4.7: Objektdefinition

In Zeile 4 wird das Objekt EUR definiert. Es erbt vom Trait $Currency$ und implementiert die abstrakte Methode $name$. Dieses Beispiel zeigt, dass abstrakte Methoden ohne Parameter durch Variablen implementiert werden können. In Zeile 8 ist zu sehen, wie die Methode $name$ direkt am Objekt aufgerufen wird.

Ein Objekt mit dem gleichen Namen wie eine Klasse wird das *Companion Object* der Klasse genannt. Die Klasse und ihr Companion Object können gegenseitig auf ihre priva-

[2] Typparameter als Grundlage von generischen Typen werden in Unterabschnitt 4.2.4 behandelt.

ten Felder und Methoden zugreifen. Methoden, die in Java als statisch markiert werden, können in Scala in das Companion Object der Klasse eingefügt werden. Ein Beispiel dafür ist eine Factory-Methode, um Instanzen einer Klasse zu erzeugen. Listing 4.8 zeigt dieses Beispiel.

```
1 class Person private (val name: String, var age: Int)
2
3 object Person{
4   def apply(name: String, age: Int) = new Person(name, age)
5 }
6 ...
7 val peter = Person.apply("Peter", "24")
8 val jochen = Person("Jochen", "26")
```

Listing 4.8: Companion Object

In Zeile 1 wird die Klasse $Person$ definiert. Das Schlüsselwort $private$ zwischen dem Klassennamen und den Klassenparametern definiert den Konstruktor der Klasse als privat. Das Companion Object $Person$ kann den Konstruktor aufrufen und in der Methode $apply$ eine Instanz der Klasse $Person$ zurückgeben. Der Methodenname $apply$ hat in Scala eine besondere Semantik. Beim Aufruf einer $apply$-Methode muss der Methodenname nicht mitgeschrieben werden. [OSV11, Kapitel 3] Zeile 7 zeigt den ausführlichen Aufruf, Zeile 8 die Kurzschreibweise.

Case Classes

Eine Case Class wird durch das Schlüsselwort $case$ vor dem $class$-Schlüsselwort eingeleitet. Die Klassenparameter werden implizit mit val gekennzeichnet, falls sie keine andere Kennzeichnung haben. Sie sind also standardmäßig öffentlich und unveränderbar. Der Unterschied zu anderen Klassen liegt darin, dass der Compiler die folgenden Methoden generiert, falls die Case Class diese nicht selbst definiert [Ode13, Kapitel 5.3.2]:

- $equals$ und $hashCode$
- $toString$
- $copy$

Die Methoden $equals$ und $hashCode$ werden so generiert, dass zwei Instanzen einer Case Class genau dann gleich sind, wenn alle ihre Felder gleich sind. Die $toString$-Methode gibt eine Zeichenkette mit dem Klassennamen und dahinter die Klassenparameter in

Klammern zurück. Die *copy*-Methode gibt ein Kopie der Instanz zurück. Änderungen einzelner Klassenparameter können der *copy*-Methode als benannte Argumente übergeben werden. [OSV11, Kapitel 15]

Der Compiler generiert für Case Classes zusätzlich ein Companion Object mit der Methode *apply*. Die *apply*-Methode hat dabei die gleichen Parameter wie die Case Class und gibt eine Instanz dieser zurück. [OSV11, Kapitel 15] In Listing 4.8 ist ein Beispiel für diese Implementierung zu sehen. Listing 4.9 zeigt das Beispiel einer Case Class *Person* mit Namen und Alter.

```
1 case class Person(name: String, age: Int)
2 ...
3 val peter = Person("Peter", 24) // Person(Peter,24)
4 var jochen = Person("Jochen", 24) // Person(Jochen,24)
5 if (peter == jochen) ... // false
6 jochen = jochen.copy(age = 26) // Person(Jochen,26)
```

Listing 4.9: Case Class

Bei der Erzeugung zweier Personen wird implizit die *apply*-Methode aufgerufen (Zeilen 3 und 4). Der ==-Operator verwendet in Scala die *equals*-Methode des linken Operanden. Zeile 6 zeigt den Aufruf von *copy*, um das Alter der Person *Jochen* zu korrigieren. Die Kommentare an den Zeilen enthalten die Ausgabe von *toString* für den jeweiligen Methodenaufruf.

4.2.3 Operatoren

Scala unterstützt Prefix-, Infix- und Postfix-Operatoren. Operatoren sind dabei Methoden. Die Grundlage dafür ist, dass Methoden in Scala sowohl alphanumerisch als auch mit einen Operator-Symbol benannt sein können. Außerdem können beim Methodenaufruf der Punkt und, falls die Methode maximal einen Parameter hat, auch die Klammern weggelassen werden. Damit können sowohl Infix- als auch Postfix-Operatoren umgesetzt werden. [OSV11, Kapitel 5] Listing 4.10 zeigt ein Beispiel.

```
1 class MyInt(i: Int) {
2   def ! : Int = if (i <= 1) i else i * ((i - 1) !)
3   def **(exp: Int): Int = if (exp == 0) 1 else i * (i ** (exp - 1))
4 }
5 ...
6 val res1 = MyInt(5) ! // 120
7 val res2 = MyInt(2) ** 5 // 32
```

Listing 4.10: Postfix- und Infix-Operator

In Zeile 2 wird eine Methode mit dem Namen ! (*Ausrufezeichen*) definiert, welche die Fakultät der Variablen i berechnet. Die Methode hat keine Parameter. Zeile 6 zeigt, wie diese Methode als Postfix-Operator verwendet wird. Zeile 3 zeigt eine Methode mit dem Namen $**$, welche eine Potenz von i berechnet. Der Parameter exp gibt den Exponenten an. Die Verwendung der Methode als Infix-Operator ist in Zeile 7 zu sehen.

Um Methoden als Prefix-Operator verwenden zu können, müssen diese mit $unary\_$ beginnen. Danach sind nur die Symbole $+$, $-$, ! und $\sim$ erlaubt. Es können also nur die genannten Symbole als Prefix-Operatoren verwendet werden, während Postfix- und Infix-Operatoren beliebig benannt sein können. Listing 4.11 zeigt ein Beispiel.

```
1 case class Meter(i: Int) {
2   def unary_- = Meter(-i)
3 }
4 ...
5 if (- Meter(10) == Meter(-10)) ... // true
```

Listing 4.11: Prefix-Operator

Die Klasse $Meter$ definiert eine Methode mit dem Name $unary\_-$, welche die Meterangabe negiert. Zeile 5 zeigt die Verwendung von "$-$" als Prefix-Operator. Der Compiler wandelt den Aufruf $-\ Meter(10)$ zu $Meter(10).unary\_-$ um.

4.2.4 Typparameter

Neben den gewöhnlichen Parametern können in Scala auch Typparameter an Klassen, Traits und Methoden übergeben werden. Dieses Sprachmittel kann mit den *Generics* in Java verglichen werden. Die Typparameter werden in eckigen Klammern hinter dem Klassen-, Trait- oder Methodennamen aufgeführt und können im Rumpf der jeweiligen Struktur benutzt werden. In Listing 4.12 ist das generische Trait $Ordered[A]$ zu sehen, welches den Typparameter A in der Methodensignatur von $compare$ verwendet.

```
trait Ordered[A]{
  def compare(that: A): Int
}
```

Listing 4.12: Definition eines generischen Traits

Das Trait $Ordered[A]$ definiert die Methode $compare$ zum Vergleich von Instanzen vom Typ A. Für den Typparameter A kann ein beliebiger Typ eingesetzt werden. Um den Typparameter zu beschränken, können sogenannte *Bounds* verwendet werden. Ein *Upper Bound* beschränkt den Typparameter von oben. Die Schreibweise $[A\ <:\ B]$ sagt aus,

dass der Typparameter A ein Subtyp von B sein muss[3]. Ein *Lower Bound* beschränkt den Typparameter entsprechend von unten. Demnach bedeutet $[A >: B]$, dass der Typparameter A ein Supertyp von B sein muss. Listing 4.13 zeigt ein Beispiel für den Einsatz eines Upper Bound. [OSV11, Kapitel 19]

```
def sort[A <: Ordered[A]](list: List[A]): List[A] = ...
```

Listing 4.13: Definition einer generischen Methode mit *Upper Bound*

Die generische Methode $sort$ erhält als Parameter eine Liste mit Elementtyp A. Der Typparameter A ist auf Subtypen von $Ordered[A]$ eingeschränkt. Innerhalb des Methodenrumpfes kann deshalb die Methode $compare$ zum Vergleich der Listenelemente verwendet werden.

Lower Bounds werden in Scala beispielsweise in Zusammenhang mit generischen unveränderbaren Datenstrukturen verwendet. Diese sind für die Arbeit allerdings nicht relevant und werden nicht weiter erläutert.

4.2.5 Implicits

Mit dem Schlüsselwort $implicit$ sind zwei unterschiedliche Konzepte verbunden. Das eine sind implizite Umwandlungen (Implicit Conversion) und das andere implizite Parameter (Implicit Parameters).

Implizite Umwandlung

Das Ziel von impliziten Umwandlungen ist es, Exemplare einer Klasse in Exemplare einer anderen Klasse zu überführen, ohne dies im Programmcode explizit zu machen.

Auf technischer Ebene werden implizite Umwandlungen vom Compiler in den Programmcode eingefügt, um Typfehler zu beheben. Das bedeutet, wenn sich ein Wert mit einem Typ A in einer Position im Programmcode befindet, in der er einen Typfehler produziert und eine Umwandlungsvorschrift für Werte dieses Typs A in einen anderen Typ B existiert, der an dieser Stelle keinen Typfehler produziert, dann wird diese Umwandlung eingefügt. Umwandlungsvorschriften von Typ A nach B sind als implizit markierte Methoden, die eine Instanz vom Typ A erhalten und ein Instanz vom Typ B zurückgeben.[OSV11, S. 446] Listing 4.14 zeigt ein Beispiel.

[3] Die Subtyp-Relation und die Supertyp-Relation sind reflexiv. Der Typ A ist also ein Subtyp (bzw. Supertyp) von A selbst.

```
1 class RichInt{ ...
2   def abs: Int = if (self < 0) -self else self
3 }
4 implicit def intWrapper(x: Int) = new RichInt(x)
5 ...
6 val a = -2.abs
7 // val a = intWrappter(-2).abs
```

Listing 4.14: Implizite Umwandlung

Zeile 6 zeigt, wie am Wert -2 vom Typ Int die Methode abs, die den Betrag eines Zahlenwertes zurückgibt, aufgerufen wird. Der Typ Int definiert allerdings keine Methode abs, was zu einem Typfehler führen würde. Weil es aber eine Umwandlungsvorschrift von Werten mit Typ Int zu Werten mit Typ $RichInt$ (Zeile 4) gibt und die Methode abs in $RichInt$ (Zeile 2) definiert ist, fügt der Compiler eine Umwandlung ein. Zeile 7 zeigt den gleichen Ausdruck wie Zeile 6 mit einer expliziten Umwandlung.

Damit implizite Umwandlungen anwendbar sind, müssen die folgenden Bedingungen erfüllt sein.

1. Es muss ein Typfehler vorliegen.
2. Es muss mindestens eine passende Umwandlungsvorschrift verfügbar sein.
3. Die zu wählende Umwandlungsvorschrift muss eindeutig sein.

Eine Umwandlungsvorschrift ist dann verfügbar, wenn sie im *Implicit Scope* liegt. Das bedeutet grob, dass sie ohne Prefix aufrufbar ist[4]. Falls mehr als eine passende Umwandlungsvorschrift verfügbar ist, wird die spezifischste angewendet. Gibt es keine spezifischste Vorschrift, wird ein Fehler angezeigt[5].

Implizite Parameter

Bei impliziten Parametern kann der Compiler eine gesamte fehlende Parameterliste anfügen, um einen Methodenaufruf zu vervollständigen. Dazu muss diese Parameterliste die letzte der Methode und als implizit markiert sein. Außerdem muss für jeden Parameter dieser Liste ein Wert vom Typ des Parameters als implizit markiert sein und im *Implicit Scope* liegen. Implizite Parameter stellen also einen Typ-gesteuerten Auswahlmechanismus zur Verfügung. [OMO10, Kapitel 4] Listing 4.15 zeigt ein Beispiel.

[4] Genaue Regeln zum Implicit Scope sind in [OSV11, Kapitel 21.2] nachzulesen.
[5] Genaueres zur Eindeutigkeit von Umwandlungsvorschriften ist in [OSV11, Kapitel 21.7] nachzulesen.

```
1 def print(msg: String)(implicit o: PrintStream) = o.print(msg)
2 ...
3 implicit val defaultOutput: PrintStream = System.out
4 ...
5 print("Hello World") // defaultOutput wird implizit uebergeben
6 print("Error occurred")(System.err)
```

Listing 4.15: Implizite Parameter

Das Listing zeigt die Methode $print$, die eine Nachricht (msg) auf eine Ausgabe (o) schreibt. Die letzte Parameterliste ist als implizit markiert. In den Zeilen 5 und 6 wird die Methode einmal mit nur einer Parameterliste und einmal voll spezifiziert aufgerufen. Bei der voll spezifizierten Variante hat das $implicit$ keine Wirkung. Bei der Variante in Zeile 5 stellt der Compiler fest, dass der Funktionsaufruf unvollständig und die fehlende Parameterliste als implizit gekennzeichnet ist. In diesem Beispiel findet der Compiler einen eindeutigen Wert vom Typ PrintStream, der als implizit markiert ist (Zeile 3), und vervollständigt damit den Funktionsaufruf.

Mit dem Konzept der impliziten Parameter ist es dem Anwender freigestellt, Parameter explizit zu übergeben oder die Vervollständigung dem Compiler zu überlassen.

4.2.6 Typklassen

Typklassen (Type Classes) wurden um 1988 für die Programmiersprache Haskell entwickelt. [HHPJW96] Sie bestehen aus einer Klassendefinition (auch Konzept genannt) und einer Instanzdeklaration (auch Modell genannt). Das Konzept beschreibt die Anforderungen an die Typen, die zur Typklasse gehören sollen. Das Modell setzt das Konzept für einen bestimmten Typ um. Typen, für die ein Modell existiert, gehören zur Typklasse.

Weder Scala noch Java unterstützen Typklassen als Sprachmittel. Mit dem *Concept Pattern* [OMO10, Kapitel 4] können Typklassen allerdings auf objektorientierte Programmiersprachen übertragen werden. Die Klassendefinition der Typklasse (im weiteren Konzept genannt) wird als generisches Interface umgesetzt. Die Instanzdeklaration (im weiteren Modell genannt) für einen Typ T ist eine Implementierung des Interfaces mit T als Typparameter.

Der aus Java bekannte $Comparator$ ist ein Beispiel für das Concept Pattern. Listing 4.16 zeigt die Implementierung und Verwendung des $Comparator$ in Scalacode.

```
1 trait Comparator[T]{
2   def compare(o1: T, o2: T): Int
3 }
```

```
4 ...
5 object IntComparator extends Comparator[Int]{
6   def compare(o1: Int, o2: Int) = o1 - o2
7 }
```

Listing 4.16: Konzept und Modell

In Zeile 1 bis 3 wird das Konzept definiert. Die Definition eines Modells für den Typ Int ist in Zeile 5 bis 7 zu sehen. Die Anforderung eine Methode $compare$ zu implementieren, wird für Int durch die Subtraktion der zu vergleichenden Werte realisiert.

Auf der Grundlage des Konzepts können generische Methoden definiert werden. Listing 4.17 zeigt die Definition einer Methode $sort$ und ihre Verwendung.

```
1 object Collections{
2   def sort[T](list: List[T])(c: Comparator[T]): List[T] = ...
3 }
4 ...
5 val unsorted = List(2,4,1,3)
6 val sorted = Collections.sort(unsorted)(intComparator) // List(1,2,3,4)
```

Listing 4.17: Verwendung von Comparator

Das Objekt $Collections$ definiert eine Methode $sort$ (Zeile 2), die eine Liste vom Typ T sortieren soll. Um die Werte vom Typ T vergleichen zu können, erwartet die Methode ein Modell des Konzepts $Comparator$ für den Typ T. $sort$ ist also nur für Listen aufrufbar, für deren Elementtyp ein Modell des Konzepts existiert. Zeile 5 und 6 zeigen den Aufruf mit einer Liste vom Typ Int und dem Modell $intComparator$.

In Java wird auf diese Weise die Sortierung von Listen an der Klasse $Collections$ angeboten. Das Modell muss immer explizit übergeben werden, kann dabei aber eine anonyme innere Klasse sein. Kombiniert man das Concept Pattern mit impliziten Parametern, muss das Modell in Scala nicht explizit übergeben werden. Diese Kombination stellt das Muster für Typklassen in Scala dar [OMO10]. Listing 4.18 zeigt eine Umsetzung.

```
1 object Collections{
2   def sort[T](list: List[T])(implicit c: Comparator[T]): List[T] = ...
3 }
4 ...
5 implicit object IntComparator extends Comparator[Int]{
6   def compare(o1: Int, o2: Int) = o1 - o2
7 }
8 ...
```

```
9  val unsorted = List(2,4,1,3)
10 val sorted = Collections.sort(unsorted) // List(1,2,3,4)
```

Listing 4.18: Typklassen in Scala

Wie in Zeile 10 zu sehen, muss das Modell nicht mehr explizit übergeben werden. Um dies zu erreichen, sind zwei Änderungen am Beispiel notwendig. Zum einen muss die zweite Parameterliste der Methode $sort$ als implizit gekennzeichnet werden und zum anderen muss das Modell für den Typ Int als implizit markiert werden und im Implicit Scope liegen.

Die Methode $sort$ ist also für alle Typen aufrufbar, für die ein Modell von $Comparator[T]$ im Implicit Scope liegt. Ein Vorteil der Programmierung mit Typklassen ist, dass ein Modell für eine Klasse implementiert werden kann, ohne den Programmcode der Klasse zu verändern.

5 Untersuchung der Sprachmittel

Da Scala Werttypen nicht als Konzept oder durch ein Sprachmittel unterstützt, müssen Werttypen (wie in Java) durch eine Kombination von Sprachmitteln und Programmierrichtlinien umgesetzt werden. In diesem Kapitel soll untersucht werden, welche Sprachmittel die in Kapitel 3 formulierten Anforderungen für die Umsetzung von Werttypen unterstützen.

5.1 Case Classes

Case Classes wurden in Scala eingeführt, um Pattern-Matching zu unterstützen.[OSV11, S. 269] Auch für die Umsetzung von Werttypen sind einige ihrer Eigenschaften hilfreich.

Kapitel 3 zeigt die Umsetzung eines Geldbetrags mit dem Value Object Pattern in Java. Eine Umsetzung des gleichen Fallbeispiels in Scala mit Hilfe von Case Classes ist in Listing 5.1 zu sehen.

```
case class Money(amount: Double, currency: Currency) {
  def +(other: Money) = {
    require(currency == other.currency)
    copy(amount = amount + other.amount)
  }
}

sealed trait Currency
case object EUR extends Currency
case object USD extends Currency
```

Listing 5.1: Geldbetrag als Case Class in Scala

Wie in Kapitel 3 wird der Geldbetrag durch Felder für Betrag ($amount$) und Währung ($currency$) repräsentiert. Die Währung ist als Aufzählung von Objekten (case object) realisiert. Die Felder der Case Class sind standardmäßig öffentlich und unveränderbar. Die Methode "+" addiert zwei Geldbeträge. Die Methoden $equals$ und $hashCode$ müssen

nicht implementiert werden, weil sie vom Compiler so generiert werden, dass Geldbeträge dann gleich sind, wenn ihr Betrag und ihre Währung gleich sind.

Die Umsetzung in Scala (10 Zeilen) ist viel kürzer und übersichtlicher als die Umsetzung in Java (37 Zeilen). Die Generierung von $equals$ und $hashCode$ verringert nicht nur den Schreibaufwand, sondern entfernt auch eine typische Fehlerquelle [RS14, S. 1]. Eine weitere Fehlerquelle, das Vergessen der Schlüsselworte $public\ final$, wird durch die standardmäßige Definition von Feldern als öffentlich und unveränderbar beseitigt.

Neben der leichtgewichtigen Definition der Klasse $Money$ ergeben sich auch bei der Verwendung einige Vorteile. Listing 5.2 zeigt die Verwendung der Klasse $Money$.

```
1 val balance1 = Money(100, EUR) // 100 EUR
2 val balance2 = Money(50, EUR) + Money(50, EUR) // 50 EUR + 50 EUR
3 if (balance1 == balance2) ... // true
```

Listing 5.2: Verwendung der Klasse $Money$

In Zeile 1 und 2 ist zu sehen, wie Exemplare der Klasse $Money$ ohne das Schlüsselwort new verwendet werden. Dies ist möglich, weil für Case Classes ein Companion Object mit einer $apply$-Methode generiert wird, die den Konstruktor der Klasse $Money$ aufruft. Technisch wird also weiterhin ein Exemplar erzeugt, was in Widerspruch zur Unerzeugbarkeit von Werten steht. Für den Aufrufer ist dieser Widerspruch aber nicht mehr sichtbar. Zeile 3 zeigt, dass der Vergleich von Exemplaren der Klasse $Money$ über den Vergleichsoperator $==$ korrekt funktioniert. Der Operator funktioniert korrekt, weil der Aufruf von $==$ in Scala die $equals$-Methode der zu vergleichenden Werte (oder Objekte) verwendet.

Case Classes scheinen gut geeignet für die Umsetzung von Werttypen. Es kann allerdings weiterhin zu einer semantisch falschen Verwendung der Klasse $Money$ kommen. Listing 5.3 zeigt Beispiele dieser Fehler.

```
1 val balance1 = new Money(100, EUR) // 100 EUR
2 Money(50, EUR) + Money(50, EUR) // ohne Wirkung - Fehler
3 val balance2 = new Money(100, EUR) // 100 EUR
4 if (balance1 eq balance2) ... // false - Fehler
```

Listing 5.3: Falsche Verwendung der Klasse $Money$

Zeile 1 zeigt, dass weiterhin Exemplare der Klasse $Money$ mit dem Schlüsselwort new erzeugt werden können. Dabei handelt es sich nicht um einen Fehler bei der Verwendung, die Schreibweise steht aber im Widerspruch zur Unerzeugbarkeit von Werten. In Zeile 2 ist ein mutmaßlicher Fehler zu sehen. Der Aufruf der Methode “+” zur Addition von zwei Geldbeträgen ist ohne Verwendung des Ergebnisses wirkungslos. Einen weiteren Fehler

zeigt Zeile 4. In Scala stellt die Methode eq den Vergleich der Referenz zweier Objekte dar. Dieser verhält sich wie der Vergleichsoperator $==$ in Java und widerspricht damit den Werttyp-Eigenschaften.

Neben den Problemen bei der Verwendung erfüllt auch die Definition von Werttypen mit Case Classes nicht alle Anforderungen. Obwohl Case Classes typischerweise zur Umsetzung von unveränderbaren Abstraktionen genutzt werden, garantieren sie weder Seiteneffektfreiheit noch referenzielle Transparenz. Entgegen der Eigenschaften von Werten könnten Geldbeträge deshalb auch als veränderbare Objekte umgesetzt werden. Listing 5.4 zeigt die Umsetzung einer veränderbaren Klasse $Money$ und deren Verwendung.

```
1  case class Money(var amount: Double, currency: Currency){
2    def +(other: Money) = {
3      require(currency == other.currency)
4      amount = amount + other.amount
5    }
6  }
7
8  val balance1 = Money(100, EUR) // 100 EUR
9  if (balance1 == Money(200, EUR)) ... // false
10 balance1 + Money(100, EUR) // korrekte Verwendungen
11 if (balance1 == Money(200, EUR)) ... // true
```

Listing 5.4: Veränderbarer Geldbetrag

Bei dieser Umsetzung ist der Betrag ($amount$) als var gekennzeichnet und damit ein öffentliches, *veränderbares* Feld der Klasse. Die Methode "+" ist hier nicht seiteneffektfrei, weil sie das Feld für den Betrag ($amount$) ändert. In Zeile 10 wird der Seiteneffekt durch den Aufruf von "+" ausgelöst und in Zeile 11 durch den Aufruf von $==$ sichtbar gemacht. Der Vergleichsoperator $==$ ist nicht referenziell transparent (Zeile 9 und 11).

Zusammengefasst sind Case Classes ein mögliches Mittel zur leichtgewichtigen Umsetzung von Werttypen. Besonders die Generierung von $equals$ und $hashCode$ sowie der Factory-Methode, durch die das Schlüsselwort new überflüssig wird, sind hervorzuheben. Allerdings lassen Case Classes durch die Methode eq die technische Implementierung von Werten als Objekte durchscheinen. Außerdem garantieren sie weder Seiteneffektfreiheit noch referenzielle Transparenz ihrer Operationen. Zudem werden semantisch falsche Verwendungen nicht erkannt.

5.2 Operatoren

Operatoren wie $+$, $-$, $*$ usw. sind aus der Mathematik bekannt. In Programmiersprachen (wie Java und Scala) bezeichnen diese Operatoren Operationen auf primitiven Datentypen wie Int oder $Double$. Sowohl in der Mathematik wie auch in den Programmiersprachen bezeichnen die Operatoren seiteneffektfreie und referenziell transparente Operationen. Da Operatoren mit diesen Eigenschaften assoziiert werden, können sie die Klarheit bei der Verwendung von Werttypen erhöhen.

Das Beispiel in Listing 5.1 definiert die Addition von Geldbeträgen mit dem Operator "+". In Listing 5.2 Zeile 2 ist die Verwendung des Operators zu sehen. Wie Listing 5.4 zeigt, können auch mit einem Operator benannte Methoden Seiteneffekte haben oder referenziell opak sein.

Auch wenn Operatoren zur Klarheit beitragen können, haben Methodennamen Vorteile, zum Beispiel falls für eine Operation noch kein allgemein gebräuchlicher Operator vorhanden ist. In diesem Fall kann im Methodennamen ausgedrückt werden, was die Operation bewirkt. Der Methodenname $distanceTo$ drückt beispielsweise deutlicher aus, dass die Distanz zu einem Punkt berechnet wird als ein Operator wie "<->" dies tut.

5.3 Value Classes

In den bisher aufgeführten Beispielen sind alle benutzerdefinierten Klassen implizit Subtyp von $AnyRef$. Sie werden in Scala *Reference Classes* genannt. Auf der anderen Seite der Typhierarchie stehen die Subtypen von $AnyVal$, auch *Value Classes* genannt. Diese Typen entsprechen den primitiven Datentypen in Java wie beispielsweise int und $double$. Abbildung 5.1 zeigt einen Ausschnitt der Typhierarchie von Scala. [OSV11, S. 211]

Seit Version 2.10 enthält Scala das neue Sprachmittel *Value Classes*. Dieses erlaubt unter einigen Einschränkungen, Subtypen von $AnyVal$ und somit neue Value Classes zu definieren. Eine Value Class C muss unter anderem die folgenden Bedingungen erfüllen[1]. [OOPS12, S. 2]

1. C hat genau einen öffentlichen, unveränderbaren Klassenparameter. Dessen Typ wird der zugrundeliegende Typ von C genannt.

2. Der zugrundeliegende Typ von C ist keine benutzerdefinierte Value Class. Ansonsten kann der zugrundeliegende Typ ein belibiger Wert- oder Objekttyp sein.

[1] Zur Vereinfachung sind nur die für den Kontext dieser Arbeit relevanten Bedingungen aufgezählt.

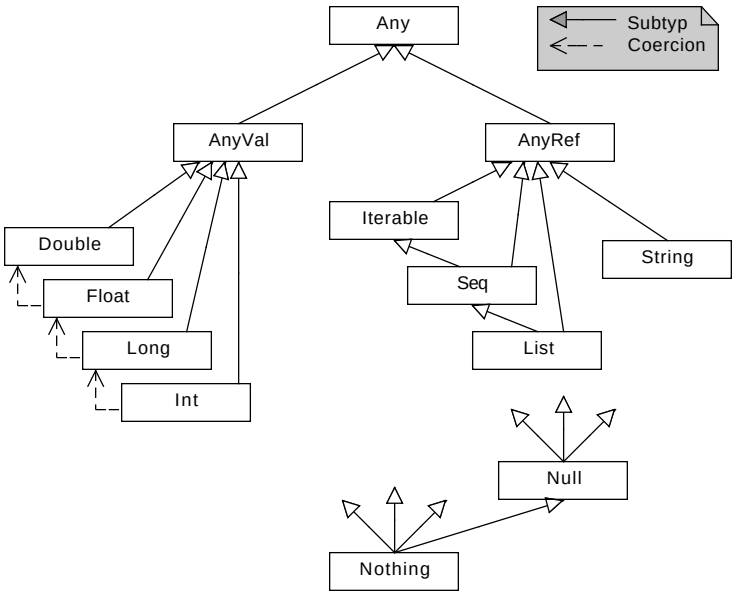

Abbildung 5.1: Klassenhierarchie von Scala

3. C hat keine sekundären Konstruktoren.
4. C definiert weder $equals$ noch $hashCode$ explizit.
5. C enthält keine Felder außer dem Klassenparameter.

Diese Einschränkungen sind deshalb notwendig, weil das Ziel der benutzerdefinierten Value Classes ist, ihre Instanziierung zur Übersetzungszeit wegoptimieren zu können. Das bedeutet, dass zur Übersetzungszeit mit der Klasse C gearbeitet wird und zur Laufzeit mit dem zugrundeliegenden Typ von C.

Um zu untersuchen, inwiefern Value Classes zur Umsetzung von Werttypen geeignet sind, betrachten wir im Folgenden die Umsetzung von drei verschiedenartigen Werttypen: Meter (Längenmaß), Geldbetrag und kartesischer Punkt (zwei-dimensionale Koordinate).

5.3.1 Beispiel Meter

Die Umsetzung des Längenmaßes Meter ist ein typisches Beispiel für Value Classes und als solches auch in der Dokumentation von Scala aufgeführt [Har13]. Listing 5.5 zeigt diese Umsetzung.

```
case class Meter(value: Double) extends AnyVal {
  def +(m: Meter): Meter = Meter(value + m.value)
}
```

Listing 5.5: Meter als Value Class

Die Klasse $Meter$ hat wie gefordert genau einen öffentlichen, unveränderlichen Parameter ($value$). Der Typ des Parameters ($Double$) ist keine *benutzerdefinierte* Value Class. Es gibt keine zusätzlichen Konstruktoren oder Felder in der Klasse. Außerdem werden $equals$ und $hashCode$ nicht explizit definiert, sondern vom Compiler generiert. Das explizite Erben von $AnyVal$ kennzeichnet die Klasse als Value Class. Zur Vereinfachung enthält $Meter$ nur eine Methode "+", die zwei Meterangaben addiert.

```
1 val x = Meter(3.4)
2 val y = Meter(4.3)
3 val z = x + y // z = 7.7 m
4 if (z == Meter(7.7)) ... // true
```

Listing 5.6: Verwendung von Meter

Listing 5.6 zeigt die Verwendung der Value Class. Nach dem Übersetzen des Programmcodes wird durch eine Optimierung anstatt der Klasse $Meter$ ihr zugrundeliegender Typ $Double$ verwendet.

Der Vorteil der Klasse $Meter$ liegt darin, dass der Entwickler zur Übersetzungszeit von der Typsicherheit der Klasse profitiert, zur Laufzeit aber nicht mehr Objekt-Instanziierungen hat, als wäre der Typ $Double$ statt $Meter$ verwendet worden. Mehr Informationen zur Funktionsweise der Optimierung sind in [OOPS12] nachzulesen.

Für diese Arbeit stellt sich die Frage, wie gut Werttypen mit Value Classes umgesetzt werden können. Dazu zeigt Listing 5.7 weitere Details der Verwendung der Klasse $Meter$ und greift die Kritik an der Verwendung von Case Classes aus Abschnitt 5.1 wieder auf.

```
1 val x = new Meter(10)
2 Meter(5) + Meter(5) // ohne Wirkung - Fehler
3 val y = new Meter(10)
4 if (x eq y) ... // nicht uebersetzbar
5 val z: Meter = null // nicht uebersetzbar
```

Listing 5.7: Verwendung von Meter als Werttyp

In Zeile 1 ist zu sehen, dass auch bei Value Classes Exemplare der Klasse $Meter$ mit dem Schlüsselwort new erzeugt werden können, was wiederum im Widerspruch zur Unerzeugbarkeit von Werttypen steht. Auch der Fehler in Zeile 2, die Addition zweier

Meterangaben ohne Verwendung des Ergebnisses, wird bei Value Classes nicht erkannt. Zeile 4 hingegen zeigt einen Vorteil von Value Classes. Der Vergleich der Objektreferenzen (eq) ist nur für Reference Classes definiert, weshalb Zeile 4 nicht übersetzbar ist. Damit haben Value Classes nur eine Vergleichsoperation. Zeile 5 verdeutlicht, dass $Null$ nur Subtyp von allen Reference Classes ist, weshalb $null$ kein gültiger Wert für eine Variable vom Typ einer Value Class ist.

Auch die Definition von Value Classes erfüllt nicht alle geforderten Anforderungen. Zwar ist das Feld (Klassenparameter) von Value Classes immer unveränderbar, der Typ des Parameters kann aber ein beliebiger Objekttyp sein. Deshalb kann das Objekt, das an das Feld gebunden ist, veränderbar sein. Im Allgemeinen garantieren Value Classes nicht, dass ihre Operationen seiteneffektfrei und referenziell transparent sind. Neben diesen nicht erfüllten Anforderungen ergeben sich weitere Probleme bei Werttypen, die durch mehr als ein Feld repräsentiert werden müssen. Die folgenden Beispiele verdeutlichen diese Problematik.

5.3.2 Beispiel Geldbetrag

In Abschnitt 5.1 wurde der Geldbetrag durch einen Betrag und eine Währung repräsentiert. Bei der Umsetzung mit Value Classes muss der Geldbetrag durch genau einen Klassenparameter repräsentiert werden. Listing 5.8 zeigt eine mögliche Umsetzung.

```
1  trait Money[T <: Money[T]] extends Any {
2    def amount: Double
3    def select(amount: Double): T
4    def +(other: T) = select(amount + other.amount)
5    def -(other: T) = select(amount - other.amount)
6  }
7
8  case class EUR(amount: Double) extends AnyVal with Money[EUR] {
9    def select(amount: Double) = EUR(amount)
10 }
11
12 case class USD(amount: Double) extends AnyVal with Money[USD] {
13   def select(amount: Double) = USD(amount)
14 }
```

Listing 5.8: Money als Value Class

Bei dieser Umsetzung sind alle Geldbeträge Subtypen des Traits $Money$. Außerdem sind alle Geldbeträge in Euro Exemplare der Klasse EUR, entsprechend für USD und Dollar und beliebige weitere Währungen. Das Trait $Money$ erwartet einen Typparameter T, welcher Subtyp von $Money[T]$ ist. T enthält, wie in Zeile 8 und 12 zu sehen, die Währung, die das Trait $Money$ erweitert. Durch diesen Mechanismus sind die Währungen Subtypen von $Money$. T steht also für Geldbeträge einer bestimmten Währung.

Das Trait $Money$ definiert in Zeile 2, dass jeder Geldbetrag einen Betrag ($amount$) haben muss. Die Klassen EUR und USD definieren das Feld für den Betrag durch den Klassenparameter der Case Class (Zeile 8 und 12). Als nächstes muss jede Währung eine Methode $select$ haben, die aus einem $Double$-Wert einen Geldbetrag der jeweiligen Währung macht. Diese Methode wird für die folgenden Operationen auf Geldbeträgen gebraucht, die im Trait $Money$ definiert sind. Die Operation "+" addiert zwei Geldbeträge einer Währung und "−" subtrahiert sie. Die Klassen für Geldbeträge in Euro und Dollar implementieren die $select$-Methode mit ihrer Factory-Methode.

Die Klassen EUR und USD haben genau einen öffentlichen, unveränderlichen Parameter und können deshalb als Value Classes definiert werden (`extends AnyVal`). Damit die Klassen das Trait $Money$ erweitern können, muss dies explizit von Any erben und damit zu einem *Universal Trait* werden. Durch die Typhierarchie in Abbildung 5.1 wird deutlich, dass $Money$ kein Subtyp von $AnyRef$ sein darf, weil es sonst auf der Seite der Reference Classes stehen würde.[2].

Listing 5.9 zeigt die Verwendung der Geldbeträge. Es ist bemerkenswert, dass die Addition zweier Geldbeträge verschiedener Währung (Zeile 5) nicht übersetzbar ist, anstatt wie in Abschnitt 5.1 einen Fehler zur Laufzeit zu produzieren. Abgesehen davon ergeben sich die gleichen Probleme bei der Verwendung wie beim Beispiel Meter in Unterabschnitt 5.3.1.

```
1 val m1 = EUR(100)
2 val m2 = EUR(200)
3 val sum = m1 + m2
4 val m3 = USD(300)
5 m3 + m1 // nicht uebersetzbar
```

Listing 5.9: Verwendung von Money

Die Umsetzung des Geldbetrags als Value Class ist um einiges komplexer als die Umsetzung als Case Class. Der Grund dafür ist unter anderem die Bedingung, dass Value Classes genau einen Parameter haben müssen. Viele Werttypen lassen sich nicht sinnvoll durch

[2]Für Universal Traits gibt es weitere Bedingungen, nachzulesen in [OOPS12, S. 2]

nur einen Parameter repräsentieren. Ein Beispiel dafür ist der angesprochene kartesische Punkt, der im Folgenden behandelt wird.

5.3.3 Beispiel Punkt

Einen Punkt durch eine x- und y-Koordinate zu repräsentieren ist eine naheliegende Modellierung. In Value Classes ist jedoch nur genau ein Feld zulässig. Um diese Problematik zu umgehen, kann die Klasse $TupleN$ (N steht für die Anzahl der Elemente des Tupel) verwendet werden. Listing 5.10 zeigt eine Umsetzung des Punktes mit Hilfe von $Tuple2$.

```
1 case class Point(tuple: Tuple2[Int, Int]) extends AnyVal {
2   def x = tuple._1
3   def y = tuple._2
4   def +(other: Point) = Point(new Tuple2(x + other.x, y + other.y))
5 }
```

Listing 5.10: Punkt als Value Class

Die Value Class $Point$ hat genau einen Parameter, nämlich $tuple$. In diesem stehen die x- und y-Koordinate. Durch die Methoden x und y (Zeile 2 und 3) werden die Koordinaten von außen zugänglich[3]. Die Operation "+" addiert zwei Punkte koordinatenweise.

Mehrere Parameter in einer Value Class durch Tupel zu simulieren hat einen Nachteil. Der wesentliche Vorteil von Value Classes geht verloren. Denn an den Stellen, wo ein $Point$ instanziiert wird, muss zunächst ein Tupel erzeugt werden (zu sehen in Zeile 4). Nachdem die Instanziierung von $Point$ wegoptimiert wurde, bleibt die Instanziierung des Tupel bestehen. Dadurch hat die Value Class keinen effizienztechnischen Vorteil gegenüber der Umsetzung eines Punktes mit Case Classes.

Zusammenfassung

Value Classes sind eine sinnvolle Ergänzung zu Case Classes. Neben der Effizienz ergeben sich weitere Vorteile for Werttypen. Die so definierten Werttypen stehen in der Typhierarchie auf der Seite der primitiven Datentypen und sind somit klar von den Reference Classes unterscheidbar. Außerdem ist der Referenzvergleich mit eq nicht definiert und $null$ ist kein zulässiger Wert.

Allerdings werden Value Classes auch zur effizienten Erweiterung von Objekttypen verwendet[4]. Nicht jede Value Class ist also die Umsetzung eines Werttyps. Auch Seiten-

[3] Um die Bedingungen der Value Class einzuhalten, müssen x und y Methoden und keine Felder sein.
[4] Dieser Anwendungsfall von Value Classes wird in Abschnitt 6.2 beschrieben.

effektfreiheit und referenzielle Transparenz werden von Value Classes nicht garantiert. Ein weiterer Nachteil ist, dass die Bedingungen, die an Value Classes gestellt werden, die Umsetzung von beliebigen Werttypen erschweren. Nur Werttypen, die durch genau einen Parameter repräsentiert werden könnnen, profitieren von allen Vorteilen der Value Classes und sind weiterhin einfach umsetzbar.

5.4 Implizite Umwandlung

Bisher standen die Klarheit, die Einfachheit und die Sicherheit im Vordergrund. Zur Erweiterbarkeit von Werttypen trägt das Sprachmittel *implizite Umwandlung* (Implicit Conversion) bei. Dabei fügt der Compiler auf Grundlage von Umwandlungsvorschriften Umwandlungen in den Programmcode ein.

Um Klassen um Operationen zu erweitern, kann der Entwickler eigene Umwandlungsvorschriften erstellen. Um dies an einem Beispiel zu illustrieren, zeigt Listing 5.11 dieselbe Klasse $Money$ wie das Beispiel für Case Classes in Listing 5.1.

```
case class Money(amount: Double, currency: Currency) {
  def +(other: Money) = {
    require(currency == other.currency)
    copy(amount = amount + other.amount)
  }
}
```

Listing 5.11: Case Class $Money$

Angenommen ein Anwendungsfall benötigt eine Operation zum Subtrahieren von Geldbeträgen. Allerdings kann der Programmcode der Klasse $Money$ nicht geändert werden, zum Beispiel weil die Klasse aus einer Bibliothek eingebunden wurde. In diesem Fall kann eine implizite Umwandlung die Klasse $Money$ um eine Operation "$-$" erweitern. Listing 5.12 zeigt den Programmcode einer solchen Erweiterung.

```
1 class RichMoney(money: Money) {
2   def -(other: Money) = {
3     require(money.currency == other.currency)
4     money.copy(amount = money.amount - other.amount)
5   }
6 }
7 ...
8 implicit def moneyToRichMoney(money: Money) = new RichMoney(money)
```

```
9  ...
10 val m1 = Money(100, EUR)
11 val m2 = Money(200, EUR)
12 val res = m2 - m1 // res = 100 EUR
```

Listing 5.12: Erweiterung mit impliziter Umwandlung

Zunächst wird (von Zeile 1 bis 6) eine Klasse $RichMoney$ definiert, die ein Exemplar ($money$) der Klasse $Money$ erhält und die Operation "$-$" definiert. In Zeile 8 ist die Umwandlungsvorschrift zu sehen. Es handelt sich hierbei um eine einfache Methode, die ein Exemplar vom Typ $Money$ als Parameter erhält und ein Exemplar von Typ $RichMoney$ zurückgibt. Um die Methode zu einer Umwandlungsvorschrift zu machen, wird sie mit dem Schlüsselwort $implicit$ gekennzeichnet.

Zeile 10 bis 12 zeigen die Verwendung der Operation "$-$". Für den Benutzer wirkt die Operation als wäre sie in der Klasse $Money$ definiert. Die implizite Umwandlung ist für den Nutzer transparent. Weil das Muster einer Klasse wie $RichMoney$ in Kombination mit einer Umwandlungsvorschrift oft auftritt, kann auch die Klasse als implizit markiert werden. Die Umwandlungsvorschrift wird dann vom Compiler generiert. Listing 5.13 zeigt die sogenannte *implizite Klasse* (Implicit Class).

```
implicit class RichMoney(money: Money) { ... }
```

Listing 5.13: Implizite Klasse $RichMoney$

Implizite Umwandlungen wurden nicht zur Unterstützung von Werttypen eingeführt, sondern bieten breitere Anwendungsmöglichkeiten. Sie bieten beispielsweise die Möglichkeit, *beliebige* Bibliotheken auf die Anforderung der eigenen Anwendung anzupassen. [Ode06]

Zusammengefasst eignen sich implizite Klassen dennoch sehr gut zum Erweitern von Werttypen um Operationen. Ihre Definition ist leichtgewichtig und die Verwendung der Operationen ist transparent. Dennoch handelt es sich bei impliziter Umwandlung um ein komplexes Sprachkonstrukt. Die exakten Regeln, wann und welche Umwandlungen der Compiler einfügt, füllen mehrere Seiten.

6 Pragmatik

6.1 Programmierrichtlinien

Mit den im letzten Kapitel untersuchten Sprachmitteln von Scala können nicht alle Anforderungen an die Umsetzung von Werttypen erfüllt werden. Diese Schwächen sollen durch Programmierrichtlinien ausgeglichen werden. Für die Schwächen im Bereich der Sicherheit werden Programmierrichtlinien zur Garantie der Seiteneffektfreiheit und referenziellen Transparenz von Wert-Operationen und zur Verhinderung von semantisch falscher Verwendung benötigt. Außerdem sind Richtlinien zur Implementierung und Verwendung aufzustellen, welche die Klarheit der Umsetzungen erhöhen.

6.1.1 Sicherheit

Um die Werttyp-Eigenschaften zu garantieren, müssen Wert-Operationen referenziell transparent und seiteneffektfrei implementiert werden. Dazu fordert diese Arbeit, dass Werttypen als unveränderbare Klassen implementiert werden. Außerdem dürfen keine globalen Variablen gelesen oder verändert werden. Wie am Beispiel von Wechselkursen für Geldbeträge in Abschnitt 3.4 veranschaulicht, ist nicht immer offensichtlich, ob eine Operation referenziell transparent ist. Gleiches gilt für die Seiteneffektfreiheit von Operationen. Wenn aus einer Wert-Operation auf Objekttypen zugegriffen wird, kann die Überprüfung der Seiteneffektfreiheit und referenziellen Transparenz durch den Entwickler sehr komplex werden. [RS11, Kapitel 4.4]

Um diesen Komplexitätszuwachs bei der Umsetzung von Werttypen zu verhindern, schließt sich diese Arbeit der Forderung aus [RS11] an, dass Werttypen keinen Zugriff auf Objekttypen haben dürfen. Dadurch wird das in Unterabschnitt 3.4.2 beschriebene asymmetrische Abhängigkeitsverhältnis zwischen Wert- und Objekttypen und damit die Trennung des Systems in einen funktionalen Kern und eine objektorientierte Schale (Abbildung 3.1) erreicht.

6.1.2 Klarheit

Eine klare Trennung von Wert- und Objekttypen kann in Scala auch durch Programmierrichtlinien nicht erreicht werden. Einige Richtlinien können dennoch bei der Unterscheidung helfen.

Um die Unerzeugbarkeit von Werten deutlich zu machen, sollte auf die direkte Instanziierung mit dem Schlüsselworts new verzichtet und stattdessen eine Factory-Methode verwendet werden. Um die mit Operatoren assoziierten Eigenschaften zu erfüllen, sollten Operatoren den Werttypen vorbehalten sein. Außerdem dürfen Werttypen nicht mittels eq (Referenzvergleich) verglichen werden.

Zusammengefasst sollten Werttypen ...

1. als unveränderbare Klassen implementiert sein.
2. keinen Zugriff auf Objekttypen haben.
3. keinen lesenden oder schreibenden Zugriff auf globale Variablen haben.
4. weder mit new instanziiert noch mit eq verglichen werden.
5. als einzige Klassen Operatoren definieren dürfen.

6.2 Hilfsmittel

Für die Implementierung von Werttypen gibt es über die formulierten Programmierrichtlinien hinaus einige Techniken, die in der Praxis von Bedeutung sein können. Im Folgenden wird das Thema *Typsicherer Vergleich* und die Kombination von impliziten Umwandlungen und Value Classes dargestellt.

6.2.1 Typsicherer Vergleich

In Unterabschnitt 3.4.1 wurde kritisiert, dass der $==$-Operator Vergleiche von Ausdrücken mit inkompatiblen Typen zulässt. Mit Hilfe von Typklassen kann ein Operator $===$ definiert werden, der einen typsicheren Vergleich durchführt. Der Operator wird für alle Typen definiert, zu denen eine Instanz der Typklasse $Equal[T]$ existiert. Listing 6.1 zeigt das Konzept der Typklasse.

```
1 trait Equal[T] {
2   def isEqual(o1: T, o2: T): Boolean
3 }
```

Listing 6.1: Typklasse $Equal$

In Zeile 2 deklariert das Konzept $Equal[T]$ eine Methode $isEqual$, die zwei Instanzen vom Typ T vergleichen soll. Mit Hilfe von $Equal[T]$ kann der Operator $===$ definiert werden, der ein Modell des Konzepts verwendet, um Ausdrücke zu vergleichen.

Die Verwendung des Operators soll der Form $a === b$ entsprechen. Der Aufruf $a === b$ bedeutet in Scala, dass die Methode $===$ an a mit dem Parameter b aufgerufen wird. Um diesen Aufruf zu ermöglichen, muss die Klasse von a um die Methode $===$ erweitert werden. Listing 6.2 zeigt die Definition des Operators $===$.

```
1 object Equal {
2   implicit class EqualOperators[T](o1: T) {
3     def ===(o2: T)(implicit eq: Equal[T]): Boolean = eq.isEqual(o1, o2)
4   }
5 }
```

Listing 6.2: Definition des Operators

Der Operator wird im *Companion Object* der Typklasse definiert. Die impliziten Klassen (Zeile 2) erweitertern T um die Methode $===$. Diese Methode erwartet als ersten Parameter die zu vergleichende Instanz $o2$, ebenfalls von Typ T, und als zweiten Parameter ein Modell für den Typ T, mit dessen Hilfe der Vergleich von $o1$ und $o2$ durchgeführt wird. Damit ist der $===$-Operator nur für die Typen aufrufbar, für die ein Modell existiert. Listing 6.3 zeigt die Verwendung des Operators. $Money$ und $Currency$ entsprechen den Werttyp-Umsetzungen mit Case Classes (Abschnitt 5.1).

```
1  implicit object EqualMoney extends Equal[Money] {
2    def isEqual(o1: Money, o2: Money) = o1 == o2
3  }
4  implicit object EqualCur extends Equal[Currency] {
5    def isEqual(o1: Currency, o2: Currency) = o1 == o2
6  }
7  ...
8  val balance = Money(100, EUR)
9  if (balance === 100) ... // Fehler - nicht uebersetzbar
10 if (balance === Money(100, EUR)) ... //true
11 if (balance.currency === EUR) ... //true
```

Listing 6.3: Verwendung des Operators

In Zeile 1 bis 6 werden die Modelle für die Typen $Money$ und $Currency$ als implizite Objekte definiert. Für die Implementierung der Methode $isEqual$ wird der $==$-Operator verwendet. Der erste Vergleich mit dem $===$-Operator (Zeile 9) zeigt einen Fehler. Der

Vergleich der Variable $balance$ von Typ $Money$ mit der Zahl 100 von Typ Int ist nicht möglich, weil die Methode $===$ einen Ausdruck von Typ $Money$ als Parameter erwartet. Die Vergleiche in Zeile 10 und 11 funktionieren korrekt, weil beide Operanden vom gleichen Typ sind und jeweils ein Modell des Typs im Implicit Scope liegt.

Generalisierung des Modells

Die beiden Modelle (Zeile 1 bis 6) verwenden jeweils den $==$-Operator, um die Methode $isEqual$ zu implementieren. Unter der Voraussetzung, dass der $==$-Operator für Werttypen korrekt implementiert ist, kann die Definition des Modells für alle Werttypen generalisiert werden. Die Generalisierung ist in Listing 6.4 zu sehen.

```
1 trait ValueClass[T <: ValueClass[T]] {
2   implicit def valueClassEqual: Equal[T] = new Equal[T] {
3     def isEqual(o1: T, o2: T) = o1 == o2
4   }
5 }
6
7 case class Money(amount: Double, cur: Currency) extends ValueClass[Money]
8 object Money extends ValueClass[Money]
```

Listing 6.4: Generalisierung des Modells

Das generische Trait $ValueClass$ enthält eine implizite Methode (Zeile 2 bis 4), die ein Modell für den Typ T zurückgibt und dabei den $==$-Operator zum Vergleich verwendet. Der Typ T ist Subtyp von $ValueClass[T]$. Zeile 7 und 8 zeigen die Verwendung des Traits. Eine Werttyp-Umsetzung erbt von $ValueClass$ und übergibt sich selbst als generischen Parameter. Dadurch gibt die Methode $valueClassEqual$ im Fall der Klasse $Money$ ein Modell für den Typ $Money$ zurück. Weil der Inhalt des Companion Objects einer Klasse Teil des Implicit Scope ist, erbt auch das Companion Object von $Money$ von $ValueClass$ (Zeile 8). Wenn die Typklasse, die Definition des Operators und das Trait $ValueClass[T]$ gegeben ist, dann kann durch Vererbung (Zeile 7 und 8) mit wenig Aufwand der $===$-Operator für Werttypen definiert werden.

Listing 6.5 veranschaulicht die Schritte, in denen der Compiler den Aufruf des $===$-Operators auflöst.

```
1 import Equal._
2 val m1 = Money(100, EUR)
3 val m2 = Money(100, EUR)
4
```

```
5 if (m1 === m2) ... // true
6 if (new EqualOperators[Money](m1).===(m2)) ... // true
7 if (new EqualOperators[Money](m1).===(m2)(Money.valueClassEqual)) ... // true
```

Listing 6.5: Funktionsweise des Operators

In der ersten Zeile wird die Definition des Operators (aus Listing 6.2) importiert und in den Zeilen 2 und 3 werden zwei Variablen mit dem gleichen Geldbetrag initialisiert. Zeile 5 zeigt die Verwendung des $===$-Operators. Die beiden folgenden Zeilen veranschaulichen wie der Compiler diese Verwendung auflöst.

Weil in der Klasse $Money$ die Methode $===$ nicht definiert ist, fügt der Compiler eine Umwandlung des ersten Operanden ($m1$) in den Typ $EqualOperators[Money]$ ein. Die eingefügte Umwandlung ist in Zeile 6 zu sehen. Die in Listing 6.2 definierte Methode $===$ erwartet einen zweiten, als implizit markierten Parameter eq vom Typ $Equal[Money]$. Der Compiler fügt an dieser Stelle den Funktionsaufruf der Methode $valueClassEqual$ am Objekt $Money$ ein (Zeile 7). Diese Methode ist in Listing 6.4 definiert und wurde an das Objekt $Money$ vererbt.

Die Typklasse $Equal[T]$ und der $===$-Operator müssen in Scala nicht selbst implementiert werden. Die Bibliothek $scalaz$ beinhaltet diese Implementierungen sowie einige Modelle für Typen der Standardbibliothek wie Int, $String$ oder $List$. Mit diesem typsicheren Vergleichsoperator können fehlerhafte Vergleiche zur Kompilierzeit erkannt werden. Ein Nachteil ist, dass mit $===$ neben den Vergleichen mit $==$ und eq noch ein dritter Operator hinzukommt. Der Entwickler muss zwischen diesen Vergleichen wählen.

6.2.2 Implizite Umwandlung und Value Classes

Abschnitt 5.4 hat implizite Klassen zur Erweiterung von Werttypen um Operationen eingeführt. Durch die Definition von Umwandlungsvorschriften können Wertypen um Operationen erweitert werden. Für einen Benutzer ist diese Umwandlung transparent. Effizienztechnisch haben diese Operationen jedoch einen Nachteil. Listing 6.6 veranschaulicht diesen Nachteil.

```
1 implicit class RichMoney(money: Money) {
2   def -(other: Money) = {
3     require(money.currency == other.currency)
4     money.copy(amount = money.amount - other.amount)
5   }
6 }
```

```
7  ...
8  val balance = Money(100, EUR)
9  val sum = balance + balance
10 val dif = balance - balance // (new RichMoney(balance)) - balance
```

Listing 6.6: Implizite Klasse $RichMoney$

$Money$ sei ein Werttyp, der nur die Operation "+" enthält. In Zeile 1 bis 6 ist die implizite Klasse $RichMoney$ definiert, die eine Operation "−" zum Werttyp $Money$ hinzufügt. In Zeile 9 ist der Aufruf der Operation "+" zu sehen. Diese wird direkt auf der Instanz der Klasse $Money$ aufgerufen. Der Aufruf der Operation "−" (Zeile 10) kann nicht direkt auf einer Instanz der Klasse $Money$ ausgeführt werden. Deshalb wird, wie im Kommentar angedeutet, eine Umwandlung zu einer Instanz der Klasse $RichMoney$ eingefügt.

Der Aufruf von Operationen, die durch implizite Umwandlungen zu einer Klasse hinzugefügt werden, führt also zu einer zusätzlichen Instanziierung der impliziten Klasse. Dieser effizienztechnische Nachteil kann durch die Verwendung von Value Classes ausgeglichen werden. Dazu genügt es, die implizite Klasse als Value Class zu implementieren. Listing 6.7 zeigt die Implementierung.

```
implicit class RichMoney(val money: Money) extends AnyVal {...}
```

Listing 6.7: $RichMoney$ als Value Class

In dieser Definition erbt $RichMoney$ von $AnyVal$ und ist damit eine Value Class. Weil Value Classes genau einen öffentlichen, unveränderbaren Parameter haben müssen, ist der Parameter $money$ als val gekennzeichnet.

Wie in Abschnitt 5.3 beschrieben, werden die Operationen von Value Classes (wie $RichMoney$) auf dem zugrundeliegenden Typ ($Money$) ausgeführt und die Instanziierung der Value Class wegoptimiert. Mit dieser Kombination von impliziten Klassen und Value Classes können Klassen ohne effizienztechnische Nachteile um Operationen erweitert werden. Diese Technik wird auch *inlined implicit wrappers* genannt. [OOPS12]

7 Evaluierung

Dieses Kapitel fasst die Stärken und Schwächen von Scala im Hinblick auf die Anforderungen an die Werttyp-Umsetzungen zusammen. Tabelle 7.1 zeigt dazu die Eigenschaften der untersuchten Sprachmittel.

7.1 Stärken

Die Stärken von Scala bezüglich der Umsetzung von Werttypen liegen in der Einfachheit und der Erweiterbarkeit. Auch zur Klarheit finden sich positive Ansätze.

Zur Einfachheit tragen bei der Umsetzung mit Case Classes die leichtgewichtige Definition mit beliebigen unveränderbaren Klassenparametern, die Generierung einer Factory-Methode als Alternative zum Schlüsselwort new, aber vor allem die generierten Methoden $equals$ und $hashCode$ bei. Allgemein können leicht Operatoren für einen Werttyp definiert werden und durch die für Werttypen semantisch korrekte Definition des $==$-Operators wird eine weitere Fehlerquelle entfernt. Für Werttypen, die mit einem Klassenparameter repräsentiert werden können, ist auch die Umsetzung mit Value Classes einfach. Bei anderen Werttypen kann die Umsetzung mit Value Classes komplex werden oder gar nicht sinnvoll möglich sein.

Einige Punkte tragen auch zur Unterscheidbarkeit von Wert- und Objekttypen und damit zur Klarheit von Werttypen-Umsetzungen bei. Die Verwendung ohne das Schlüsselwort new, der Vergleich mit dem $==$-Operator und die Möglichkeit, Operatoren für Werttypen zu definieren, können die Klarheit bei der Verwendung erhöhen. Eine Definition als Value Class ordnet Werttypen sogar in der Typhierarchie auf der Seite der primitiven Datentypen ein und verhindert einen Referenz-Vergleich mittels eq.

Die Erweiterbarkeit ist eine große Stärke von Scala. Mit impliziten Umwandlungen können Operationen zu Werttypen (wie zu Objekttypen) hinzugefügt werden, ohne den Programmcode der Klasse zu ändern. Die Definition der Umwandlungsvorschriften ist leichtgewichtig und die Umwandlung ist für den Benutzer transparent. Das macht gerade die Verwendung von Werttypen aus Bibliotheken flexibler, weil diese um fehlende Operationen erweitert werden können.

Sprachmittel	Einfachheit	Klarheit	Sicherheit	Erweiterbarkeit
Case Classes	++	−	−−	o
Operatoren	+	+	−	o
Value Classes	+	+	−	o
Implizite Umwandlung	−	o	o	++

Tabelle 7.1: Stärken und Schwächen der untersuchten Sprachmittel

7.2 Schwächen

Die größte Schwäche offenbart Scala bei der Sicherheit. Es gibt keine Möglichkeit, die charakteristischen Eigenschaften von Werttypen zu garantieren oder zu prüfen. Methoden können in Scala im Allgemeinen Seiteneffekte haben und referenziell opak sein. Es gibt in der Sprache keine Möglichkeit, Seiteneffektfreiheit oder referenzielle Transparenz zu erzwingen oder durch den Compiler prüfen zu lassen. Bei der Verwendung werden außerdem semantische Fehler, wie der Aufruf einer Werttyp-Operation ohne Verwendung des Ergebnisses nicht erkannt. Auch lässt der $==$-Operator den Vergleich von Ausdrücken inkompatibler Typen zu.

Auch die Klarheit der Werttyp-Umsetzungen ist eine Schwäche von Scala, weil keine klare Trennung von Wert- und Objekttypen erreicht werden kann. Werte werden weiterhin mit Objekten umgesetzt, können mit new instanziiert und im Allgemeinen mit eq verglichen werden. Die Operatoren werden zwar mit Werteigenschaften assoziiert, sie können diese aber nicht garantieren. Auch die Trennung in der Typhierarchie ist nicht strikt. So müssen weder alle Reference Classes Objekttypen sein (bspw. $String$), noch sind alle Value Classes Werttypen (siehe *inlined implicit wrappers* [OOPS12]).

7.3 Schlussfolgerung

Die fundamentalen Unterschiede zwischen Werten und Objekten verschwimmen bei der Programmierung in Scala. Eine Garantie von referenzieller Transparenz und Seiteneffektfreiheit ist nicht möglich. Es gibt keine klare Trennung zwischen funktionalen und imperativen Sprachmitteln. Dadurch fällt auch die Trennung von Wert- und Objekttypen schwer. Deshalb kommt diese Arbeit zu dem Ergebnis: Die Umsetzung von Werttypen unter Erfüllung aller formulierten Anforderungen ist in Scala nicht möglich.

8 Fazit

In der Einleitung wurde die Hoffnung zum Ausdruck gebracht, dass Scala den funktionalen und den objektorientierten Programmierstil so verbindet, dass Werte und Objekte gleichermaßen unterstützt werden und wohlunterscheidbare Teile eines Anwendungssystems bilden.

Diese Verbindung der Programmierstile ist in Scala nicht umgesetzt. Das Problem ist die fehlende Trennung von funktionalen und objektorientierten Konzepten. Dadurch können funktional implementierte und damit zustandslose Programmteile nicht von zustandsbehafteten Programmteilen unterschieden werden. Seiteneffekte und Zustandsänderungen sind prinzipiell überall möglich. Der Entwickler ist für die Trennung von Zustand und Zustandslosigkeit im Anwendungssystem verantwortlich und wird bei der Sicherstellung von Seiteneffektfreiheit und referenzieller Transparenz nicht unterstützt.

Deshalb können in Scala bei der Umsetzung von Werttypen die charakteristischen Eigenschaften nicht garantiert und Wert- und Objekttypen nicht klar voneinander getrennt werden. Mit diesem Ergebnis schließt sich diese Arbeit der Forderung nach einer expliziten Sprachunterstützung für Werttypen an.

Literaturverzeichnis

[BRS+98] D. Bäumer, D. Riehle, W. Siberski, C. Lilienthal, D. Megert, K.-H. Sylla, and H. Züllighoven. Values in Object Systems. Technical report, Ubilab, UBS AG, Oktober 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1 edition, 1995. 37. Reprint (2009).

[Har13] Mark Harrah. Value Classes and Universal Traits. Scala Documentation, 2013. `http://docs.scala-lang.org/overviews/core/value-classes.html`.

[HHPJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, March 1996.

[Mac81] Bruce J. MacLennan. Values and Objects in Programming Languages. Technical report, Naval Postgraduate School (U.S.), April 1981.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. 2004.

[Ode] Martin Odersky. What is Scala? `http://www.scala-lang.org/what-is-scala.html`. [Online; accessed 10-März-2014].

[Ode06] Martin Odersky. Pimp my Library. `http://www.artima.com/weblogs/viewpost.jsp?thread=179766`, 2006. [Online; accessed 19-September-2013].

[Ode13] Martin Odersky. The Scala Language Specification. Technical report, EPFL, September 2013.

[OMO10] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type Classes As Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM.

[OOPS12] Martin Odersky, Jeff Olson, Paul Phillips, and Joshua Suereth. Sip: Value Classes. Part of the Scala Improvement Process, Februar 2012. `http://docs.scala-lang.org/sips/pending/value-classes.html`.

[OSV11] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2 edition, Januar 2011.

[Rat06] Jörg Rathlev. Ein Werttyp-Konstruktor für Java, August 2006.

[Rie06] Dirk Riehle. Value Object. In *Proceedings of the 2006 conference on Pattern languages of programs*, PLoP '06, pages 30:1–30:6, New York, NY, USA, 2006. ACM.

[Rit03] Beate Ritterbach. Eigene werttypen in java. *JavaSpektrum 4/2003*, 2003.

[Rit04] Beate Ritterbach. Support for Value Types in an Object-Oriented Programming Language. In Mathias Weske and Peter Liggesmeyer, editors, *Net.ObjectDays*, volume 3263 of *Lecture Notes in Computer Science*, pages 9–23. Springer, 2004.

[Rit14] Beate Ritterbach. *Werttypen in objektorientierten Programmiersprachen*. PhD thesis, Universität Hamburg, 2014. Noch nicht erschienen.

[RS11] Beate Ritterbach and Axel Schmolitzky. Werttypen in objektorientierten Programmiersprachen: Anforderungen an eine Sprachunterstützung. In Ralf Reussner, Matthias Grund, Andreas Oberweis, and Walter F. Tichy, editors, *Software Engineering*, volume 183 of *LNI*, pages 111–122. GI, 2011.

[RS14] Beate Ritterbach and Axel Schmolitzky. Warum Programmiersprachen genau einen Vergleich unterstutzen sollten. Technical report, Fachbereich Informatik, Universität Hamburg, 2014.

[Win10] Fredrik Winkler. Benutzerdefinierte Werttypen in C++, September 2010.

Danksagung

Als Erstes möchte ich mich bei Frau Beate Ritterbach bedanken, die diese Arbeit in ihrer Rolle als Erstgutachterin und mit ihrer Forschung zum Thema Werttypen erst möglich gemacht hat. Besonders bedanken möchte ich mich auch für die erstklassige Betreuung während der Bearbeitung. Die regelmäßige Rückmeldung und die Diskussionen zum Thema Werttypen haben sehr zu dieser Arbeit beigetragen.

Des Weiteren möchte ich Herrn Dr. Axel Schmolitzky danken, dass er mich auf das Thema dieser Arbeit aufmerksam gemacht und sich bereit erklärt hat, diese Arbeit als Zweitgutachter zu betreuen.

Mein besonderer Dank geht an Fred Winkler für die intensive Betreuung während der Bearbeitung. Seine Hinweise zum Stil, Aufbau und zur Versändlichkeit waren sehr hilfreich.

Ich möchte mich auch bei Sandra Kunz und Sabine Witt für die regelmäßige korrektur der Rechtschreibung und die Hinweise zum Ausdruck bedanken. Für abschließende Korrekturen und Rückmeldungen bedanke ich mich außerdem bei Dennis Michielse, Mirko Köster und Oliver Bestmann.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Erik Witt

Hamburg, den 12. Juni 2014