Ein Werttyp-Konstruktor für Java

Erweiterung einer objektorientierten Programmiersprache um Werttypen

Diplomarbeit

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Jörg Rathlev
Försterweg 52
22525 Hamburg

11. August 2006

Erstbetreuer: Dr. Axel Schmolitzky, Universität Hamburg
Zweitbetreuer: Dr. Michael Kölling, University of Kent

Inhaltsverzeichnis

Kapitel 1

Einführung

Werte und Objekte sind zwei grundlegend unterschiedliche Konzepte in der Softwaretechnik. Bei der Modellierung von Anwendungssystemen werden beide Konzepte benötigt. Objekte werden unter anderem verwendet, um Gegenstände und fachliche Dienste des Anwendungsbereichs zu modellieren. Mit Werten dagegen können Werte im mathematischen Sinne und abstrakte Größen modelliert werden.

Bei der Implementierung eines Softwaresystems stellt sich dann das Problem, Werte und Objekte so zu implementieren, dass die Implementierung ihren konzeptionellen Eigenschaften entspricht. Die meisten objektorientierten Programmiersprachen stellen an Werttypen lediglich einige einfache Typen wie ganze Zahlen, Gleitkommazahlen und Boolesche Werte zur Verfügung (primitive Typen), bieten Entwicklern jedoch keine Möglichkeit, eigene Werttypen zu definieren.

Die Nutzung von Werten hat eine Reihe von Vorteilen. Werte verhalten sich referentiell transparent, was zum Beispiel die Weitergabe von Werten einfacher gestaltet, da keine Probleme durch unbeabsichtigte gemeinsame Nutzung zu befürchten sind. Referentielle Transparenz kann die Verständlichkeit von Software verbessern (vgl. [Mey97]). Sie bietet auch aus technischer Sicht Vorteile in verteilten und nebenläufigen Systemen und kann die Implementierung von Persistenzmechanismen und Datenbankanbindungen vereinfachen (vgl. [BRS+98]).

Möchten Entwickler von diesen Vorteilen profitieren, also Werttypen verwenden, so müssen sie, wenn die Programmiersprache keine Möglichkeit zur Definition eigener Werttypen anbietet, entweder die von der Programmiersprache angebotenen primitiven Typen verwenden oder Objektklassen verwenden, um Werte zu modellieren.

Beide Lösungen sind unbefriedigend. Bei der Nutzung der primitiven Typen stehen nur die für diese von der Sprache definierten Operationen zur Verfügung, die jedoch allgemein gehalten sind und sich nicht an den fachlichen Anforderungen des jeweiligen Einsatzkontextes orientieren.

Werden zur Modellierung von Werten dagegen Objektklassen verwendet, muss der Entwickler selbst dafür Verantwortung tragen, bei der Implementierung die konzeptionellen Eigenschaften von Werten zu berücksichtigen. Die Programmiersprache bietet dem Entwickler hierfür keine Hilfestellung und kann nicht sicherstellen, dass Exemplare des Typs sich tatsächlich wie Werte verhalten. Die Verwendung von Objektklassen verringert außerdem die Ausdrucksstärke des Quelltextes, da einer Objektklasse nicht immer auf den ersten Blick angesehen werden kann, ob diese einen Werttyp oder einen Objekttyp definieren soll.

Daher scheint es ratsam, objektorientierte Programmiersprachen um ein zusätzliches Sprachmittel zu erweitern, dass Entwicklern die Definition eigener Werttypen erlaubt. Dies wird auch an verschiedenen Stellen in der Literatur gefordert, z. B. in [BRS+98] und [Wer98].

1.1 Bisherige Arbeiten

Am Arbeitsbereich Softwaretechnik der Universität Hamburg wurden bereits mehrere studentische Projekte durchgeführt, die sich mit der Unterstützung von Werttypen in der objektorientierten Programmierung beschäftigt haben.

Müller stellt in [Mül99] ein Rahmenwerk für die Sprache Java vor, dass die Implementierung von Werten vereinfachen soll. Basierend auf diesem Rahmenwerk hat Sauer in [Sau01] Möglichkeiten untersucht, XML-Schema zur Definition von Werttypen zu verwenden.

Fürter und Spreckelmeyer haben in [FS02] ein grafisches Entwicklungswerkzeug zur Bearbeitung von Fachwertbeschreibungen entwickelt. Aus der mit dem Werkzeug entwickelten Fachwertbeschreibung generiert das Werkzeug einen Quelltext, der den beschriebenen Fachwert implementiert. Dafür wird auf das Rahmenwerk aus [Mül99] zurückgegriffen.

Heiden hat in [Hei05] eine Sprache (abstraktes Fachwertmodell, AFM) zur Deklaration von Werttypen in Java-ähnlicher Syntax entworfen. Aus dem AFM-Quelltext wird Java-Quelltext generiert, der die deklarierten Werttypen implementiert. Allerdings ist das AFM keine integrierte Programmiersprache, sodass Entwickler das AFM und Java parallel nutzen müssen. Die Nutzung von Werten unterscheidet sich dabei zwischen den beiden Sprachen, weil im AFM spezielle syntaktische Konstrukte zum Umgang mit Werten

bereitgestellt werden, die in Java nicht zur Verfügung stehen. Entwickler müssen wissen, in welcher Weise die im AFM deklarierten Klassen in Java umgewandelt werden, um die Werte in Java nutzen zu können.

1.2 Ziele dieser Arbeit

Ziel dieser Arbeit ist es, ein Konzept für integrierte, objektorientierte Programmiersprachen zu entwickeln, die sowohl die Definition benutzerdefinierter Objektklassen als auch benutzerdefinierter Werttypen ermöglichen.

Basierend auf diesem Konzept soll eine Programmiersprache entworfen werden, die auf Java basiert und Java entsprechend dem Konzept um eine Möglichkeit zur Deklaration von Werttypen erweitert. Für diese Sprache soll ein einfacher Compiler entwickelt werden, der es erlaubt, die Möglichkeiten der Sprache auch praktisch zu erkunden.

1.3 Aufbau dieser Arbeit

In Kapitel 2 werden die konzeptionellen Unterschiede zwischen Werten und Objekten genauer untersucht und von anderen Konzepten abgegrenzt. Außerdem werden die von verschiedenen objektorientierten Programmiersprachen angebotenen Sprachmittel genauer betrachtet.

In Kapitel 3 wird dann ein Konzept vorgestellt, dass als Grundlage für Programmiersprachen dienen kann, die sowohl die Erstellung von Werttypen als auch von Objekttypen erlauben.

Kapitel 4 stellt die Programmiersprache VJ vor, eine auf der Basis des in Kapitel 3 erläuterten Konzepts entworfene, objektorientierte Programmiersprache.

Die Implementierung des VJ-Compilers wird in Kapitel 5 erläutert.

Kapitel 6 fasst die Arbeit zusammen und gibt einen Ausblick auf mögliche weitere Entwicklungen.

Kapitel 2

Werte und Objekte in objektorientierten Programmiersprachen

In diesem Kapitel werden die Begriffe *Wert* und *Objekt* definiert und erläutert und von anderen, häufig mit ähnlichen Begriffen bezeichneten Konzepten, abgegrenzt. Es wird untersucht, inwieweit die vorgestellten Konzepte in verschiedenen Programmiersprachen unterstützt werden.

2.1 Werte und Objekte

In Anlehnung an die Arbeit von MacLennan (vgl. [Mac82]) lassen sich die wesentlichen Unterschiede zwischen Werten und Objekten wie folgt zusammenfassen:

- Werte sind zeitlos.
- Werte haben keinen veränderbaren Zustand.
- Werte verhalten sich referentiell transparent.
- Werte haben keine Identität.

Demgegenüber gilt für Objekte:

- Objekte werden erzeugt und zerstört.
- Objekte haben einen veränderbaren Zustand.

- Wird ein Objekt verändert, so ändert sich dessen Identität nicht.
- Objekte können gemeinsam verwendet werden.

2.1.1 Werte

Werte sind zeitlos, sie existieren unabhängig von Raum und Zeit. Dagegen durchlaufen Objekte einen Lebenszyklus, sie werden erzeugt und können zerstört werden.

Werte haben keinen veränderbaren Zustand. Ein Wert kann niemals so verändert werden, dass er „einen anderen Wert hat“. Wird mit Werten gerechnet, so verändern die Rechenoperationen nicht die Werte, sondern das Ergebnis ist ein anderer Wert. Zum Beispiel wird durch die Auswertung von $2 + 3$ weder einer der Werte „2“ oder „3“ verändert (oder zerstört), noch wird der Wert „5“ erst durch Auswertung von $2 + 3$ erschaffen.

Werte verhalten sich referentiell transparent. Das bedeutet, dass das Ergebnis eines Ausdrucks ausschließlich durch seinen Wert bestimmt wird. Ein Teilausdruck kann durch einen beliebigen anderen Teilausdruck, der den gleichen Wert ergibt, ersetzt werden, ohne dass dadurch das Ergebnis des Gesamtausdrucks beeinflusst würde (vgl. [Fol97]). Bei der Auswertung von Werten treten keine Seiteneffekte auf.

Referentielle Transparenz erleichtert das Verständnis der Software und kann somit die Wartbarkeit verbesern (vgl. [Mey97]).

Dass ein Wert keine Identität hat, bedeutet, dass gleiche Werte nicht durch eine künstliche Identität unterscheidbar sind. Es ist nicht sinnvoll, eine 42 von einer anderen 42 unterscheiden zu wollen, beides ist derselbe Wert.

Eine weitere wichtige Eigenschaft von Werten ist, dass der Zugriff auf Werte immer nur über eine Repräsentation möglich ist. Der eigentliche, abstrakte Wert kann nicht direkt genutzt werden. Dabei ist es egal, welche Repräsentation man verwendet, die unterschiedlichen Repräsentationen 42, `0x2A` und „Zweiundvierzig“ bezeichnen jeweils genau denselben Wert.

2.1.2 Objekte

Objekte existieren nicht zeitunabhängig, sie müssen erzeugt und können zerstört werden. Sie haben einen Zustand, der in der Regel während der Lebensdauer des Objektes verändert werden kann.

Außerdem besitzen Objekte eine zustandsunabhängige Identität. Wird der Zustand eines Objektes geändert, so wird seine Identität dadurch nicht verändert, es handelt sich vor und nach der Änderung um dasselbe Objekt.

Diese Eigenschaft ermöglicht es, Objekte eindeutig und unabhängig von ihrem Zustand zu referenzieren (vgl. [Mey97]).

Objekte können daher durch mehrere Klienten gleichzeitig gemeinsam verwendet werden, wobei Zustandsänderungen jeweils für alle Klienten sichtbar werden. Diese Eigenschaft von Objekten kann als erwünschte Eigenschaft zum Austausch von Daten zwischen Objekten verwendet werden. Sie kann aber auch zu unerwünschten Effekten führen, wenn dasselbe Objekt an mehreren Stellen verwendet wird, ohne dass dies den einzelnen Klienten bekannt ist, sodass diese auf durch andere Klienten durchgeführte Änderungen nicht vorbereitet sind. Dieses Problem wird in der Regel als *Aliasing-Problem* bezeichnet.

Bei der Verwendung von Werten kann dieses Problem aufgrund ihrer referentiellen Transparenz nicht auftreten. Alle Repräsentationen desselben Wertes sind unabhängig voneinander.

Es ist zu beachten, dass Werte nicht lediglich Objekte sind, deren Zustand nicht geändert werden kann. Ein unveränderliches Objekt ist kein Wert, denn es existiert nicht zeitunabhängig, sondern es muss erzeugt und kann zerstört werden. Auch haben unveränderliche Objekte einen von ihrem Zustand unabhängige Identität, zwei unveränderliche Objekte mit identischem Zustand sind dennoch zwei verschiedene, unterscheidbare Objekte.

Ein Beispiel für ein unveränderliches Objekt, bei dem es sich nicht um einen Wert handelt, ist ein Log-Eintrag, durch den das Auftreten eines Ereignisses dokumentiert wird. Der Eintrag bleibt nach der Aufzeichnung unverändert. Bevor das Ereignis aufgetreten ist, existiert er jedoch weder technisch noch konzeptionell.

2.2 Speichermodelle

Die Unterscheidung zwischen Werten und Objekten muss abgegrenzt werden von der Unterscheidung zwischen verschiedenen Möglichkeiten, Werte oder Objekte in einem System zu speichern und zu referenzieren. Die im Folgenden beschriebenen Konzepte *Wertsemantik* und *Referenzsemantik* scheinen mit den Konzepten Wert bzw. Objekt verwandt zu sein. Tatsächlich handelt es sich jedoch um zueinander orthogonale Konzeptpaare.

Der Begriff *Variable* wird im folgenden Text im Sinne des Variablenbegriffs der imperativen Programmierung verwendet. Nach Sebesta kann eine Variable charakterisiert werden als ein Sechstupel bestehend aus Name, Adresse, Wert, Typ, Lebensdauer und Sichtbarkeit (vgl. [Seb02]). Dabei ist die *Adresse* die mit der Variablen assoziierte Speicheradresse, der *Wert* ist

der Inhalt des dort für die Variable reservierten Speicherbereichs. Um Verwechslungen mit Werten im Sinne dieser Arbeit zu vermeiden, wird der Wert einer Variablen im Folgenden als ihre *Belegung* bezeichnet. Die Belegung einer Variablen kann zur Laufzeit durch Zuweisungen verändert werden.

Eine Variable kann sich auf ein Objekt oder einen Wert beziehen, dieser wird im Folgenden als die *Bezugsgröße* der Variablen bezeichnet. Die Bezugsgröße einer Variablen kann zur Laufzeit durch Zuweisung verändert werden. Der *Typ* einer Variablen legt in statisch getypten Sprachen fest, auf welche Bezugsgrößen sie sich zur Laufzeit beziehen kann.

Von Wertsemantik wird gesprochen, wenn eine Variable direkt mit ihrer Bezugsgröße belegt ist, die Belegung der Variablen also der Wert bzw. das Objekt ist, auf das die Variable sich bezieht. Wertsemantik wird in dieser Arbeit als *Speichersemantik* bezeichnet.

Speichersemantik erfordert bei Zuweisungen von einer Variablen an eine andere, dass die Bezugsgröße kopiert wird.

In vielen Programmiersprachen kommt Speichersemantik bei einfachen Datentypen wie Zahlen, Zeichen und Booleschen Werten zum Einsatz. Aber auch Objekttypen können mit Speichersemantik verwendet werden. In diesem Fall sind alle zum Zustand des jeweiligen Objektes gehörenden Daten direkt in der Variablen abgelegt, die sich auf das Objekt bezieht.

Bei Referenzsemantik wird in einer Variablen lediglich eine Referenz auf ihre Bezugsgröße abgelegt. Bei einer Zuweisung wird diese Referenz kopiert, nicht jedoch die referenzierte Bezugsgröße. Die Variablen sind also nach der Zuweisung nicht unabhängig voneinander. Wird nach der Zuweisung über eine der Variablen der Zustand der Bezugsgröße verändert, so ist der neue Zustand anschließend über beide Variablen sichtbar.

Referenzsemantik wird in der Regel für Objekttypen verwendet.

2.3 Parameterübergabe

Beim Aufruf von Operationen gibt es verschiedene Modelle zur Übergabe von Parametern. Eine Übersicht findet sich in [Seb02]. Die beiden in objektorientierten Programmiersprachen am häufigsten unterstützten Modelle sind die *Übergabe als Wert* (*pass-by-value*) und die *Übergabe per Referenz* (*pass-by-reference*).

Bei der Übergabe als Wert werden die aktuellen Parameter als Ausdrücke ausgewertet und die formalen Parameter mit den Ergebnissen initialisiert. Die formalen Parameter dienen dann innerhalb der Methode als lokale Variablen (vgl. [Seb02]). Diese Form der Übergabe entspricht in ihrer Semantik

einer Zuweisung vom aktuellen an den formalen Parameter und wird daher nachfolgend als *Übergabe durch Zuweisung* bezeichnet.

Werden Parameter durch Zuweisung übergeben, können als aktuelle Parameter sowohl Variablen als auch Ausdrücke angegeben werden. Wird eine Variable angegeben, so ist nach der Übergabe deren Belegung unabhängig von der Belegung des formalen Parameters. Eine Zuweisung an den formalen Parameter innerhalb der aufgerufenen Methode ändert die Belegung der aktuellen Parametervariablen nicht.

Bei der Übergabe per Referenz sind die formalen Parametervariablen Referenzen auf die beim Aufruf angegebenen Variablen. Bei dieser Form der Übergabe müssen als aktuelle Parameter Variablen angegeben werden, die Verwendung von Ausdrücken ist nicht möglich. Die aufgerufene Methode kann durch Zuweisungen an den formalen Paremeter die Belegung der als aktueller Parameter angegebenen Variablen ändern.

Die Art der Parameterübergabe ist unabhängig von Typ und Speichermodell der Variablen. Sowohl Variablen mit Speichersemantik als auch Variablen mit Referenzsemantik können sowohl durch Zuweisung als auch per Referenz übergeben werden.

Wird eine Referenz durch Zuweisung übergeben, so wird der formale Parameter mit einer weiteren, unabhängigen Referenz auf dieselbe Bezugsgröße belegt. Ist die Bezugsgröße ein Objekt, so kann die aufgerufene Methode über diese Referenz den Zustand des Objektes verändern[1] und diese Änderungen werden für jeden Klienten, der dieses Objekt referenziert, sichtbar, auch für den Aufrufer der Methode. Wird jedoch innerhalb der Methode dem formalen Parameter eine andere Referenz zugewiesen, so wirkt diese Änderung sich nicht auf den Aufrufer aus, da dessen Variable unabhängig davon weiterhin mit derselben Referenz auf dasselbe, zuvor referenzierte Objekt belegt ist.

Die Übergabe per Referenz dagegen ermöglicht es der aufgerufenen Methode, den Inhalt der Variablen des Aufrufers zu verändern. Die Methode kann dieser also eine andere Referenz zuweisen.

2.4 Unterstützung von Werten in existierenden Programmiersprachen

In diesem Abschnitt werden die objektorientierten Programmiersprachen C#, Eiffel, Java und Smalltalk daraufhin untersucht, wie diese mit Werten und Objekten umgehen.

[1] Wenn das Objekt entsprechende modifizierende Operationen bereitstellt.

Die Informationen über C# wurden [Mös03] und [Ecm05] entnommen, die Informationen über Eiffel [Mey97]. Für Java wurde [GJSB05] verwendet und für Smalltalk [GR83].

2.4.1 Smalltalk

Smalltalk basiert auf dem Grundprinzip „alles ist ein Objekt“. In Smalltalk sind alle Typen als Objektklassen modelliert, auch einfache Typen wie Ganzzahlen. Für Variablen wird in Smalltalk ausschließlich Referenzsemantik verwendet. Speichersemantik kann nicht genutzt werden.

Für den Zugriff auf Exemplare einfacher Typen stellt Smalltalk ebenso wie andere Programmiersprachen Literale zur Verfügung, diese werden als *literal Constants* bezeichnet. Sie sind als konstante Referenzen auf Objekte definiert, d. h. die Sprache sichert zu, dass sich ein Literal immer auf dasselbe Objekt bezieht.

Diese Objekte existieren per Definition, Mechanismus und Zeitpunkt der Erzeugung werden nicht definiert (vgl. [KR99]).

Davon abgesehen unterscheiden sich die Objekte jedoch nicht von anderen Objekten in Smalltalk. Sie können zum Beispiel geklont werden, was dazu führt, dass Identität und Gleichheit für diese Objekte unterschiedlich sein können (vgl. [KR99]).

Smalltalk bietet zwei Gleichheitsoperatoren, `==` und `=`. Letzterer ist für die Klassen der einfachen Typen standardmäßig redefiniert als oberflächlicher Gleichheitsvergleich (shallow equality). Dies entspricht für diese Typen einer Prüfung auf Wertgleichheit. Der Operator `==` kann zum Vergleich von Objekten einfacher Typen aufgrund der erwähnten Möglichkeit, diese zu klonen, nicht sicher verwendet werden.

Benutzerdefinierte Typen können in Smalltalk nur als Objektklassen definiert werden. Eine spezielle Unterstützung für die Definition von Werten bietet Smalltalk nicht an. Ein wertähnliches Verhalten kann daher in Smalltalk nur über die Erstellung unveränderlicher Objekte angenähert werden.

Zusammenfassend lässt sich sagen, dass Smalltalk die Modellierung von Werttypen praktisch nicht unterstützt.

2.4.2 Eiffel

In Eiffel sind, ähnlich wie in Smalltalk, alle Typen Objektklassen. Eiffel unterscheidet allerdings zwischen *Referenztypen* und *expanded Types*. Letztere werden mit Speichersemantik anstelle von Referenzsemantik verwendet.

Beispiel 2.1 Deklaration einer Variablen x vom expanded Type C in Eiffel (Quelle: [Mey97, S. 254])

x: **expanded** C

Beispiel 2.2 Deklaration einer Klasse als expanded Type in Eiffel (Quelle: [Mey97, S. 255])

expanded class E **feature**
    ... wie in jeder anderen Klasse ...
end

Einfache Typen wie Ganzzahlen – die in Eiffel Objektklassen sind – sind expanded Types. Aber auch benutzerdefinierte Typen können als expanded Types verwendet werden. Die Auswahl kann an der Variablen erfolgen (siehe Beispiel 2.1), dann werden Objekte in dieser Variablen mit Speichersemantik gespeichert, oder in der Deklaration des Typs, wodurch Speichersemantik für alle Variablen des Typs vorgegeben wird (siehe Beispiel 2.2). Ein Typ, der in der Typdeklaration als expanded Type deklariert ist, kann nicht mit Referenzsemantik verwendet werden.

Variablen eines expanded Types enthalten immer ein Objekt, sie können keine Nullreferenz (*Void* in der Eiffel-Terminologie) enthalten.

Eine Zuweisung $x := y$ weist x die in y enthaltene Referenz zu, wenn beide Variablen Referenzen enthalten. Ist der Typ von x ein expanded Type, dann kann x keine Referenz zugewiesen werden kann. In diesem Fall wird der Zustand des Objektes, auf das y sich bezieht, in das in x enthaltene Objekt kopiert. Die verwendete Kopieroperation ist eine oberflächliche Kopie (shallow Copy), d. h. sind in dem kopierten Objekt Referenzen enthalten, dann wird nicht das referenzierte Objekt kopiert, sondern die Referenz, sodass die Kopie anschließend dasselbe Objekt referenziert wie das Original (vgl. [GS00]).

Hat bei der Zuweisung x einen Referenztyp, y aber einen expanded Type, so wird das in y enthaltene Objekt geklont und x eine Referenz auf den Klon zugewiesen. Das stellt sicher, dass niemals Referenzen auf Objekte eines expanded Types existieren können. Bei der Klonoperation handelt es sich wie auch beim Kopieren um ein oberflächliches Klonen.

Für den Vergleich von Objekten auf Gleichheit stellt Eiffel den =-Operator bereit. Sind beide Operanden Referenzen, so werden diese auf Referenzgleichheit geprüft, also darauf, ob beide dasselbe Objekt referenzieren. Ist dagegen mindestens einer der Operanden ein expanded Type, dann werden die Objekte auf oberflächliche Gleichheit geprüft.

Expanded Types sind keine Werttypen im Sinne dieser Arbeit. Die Sprache sichert nicht zu, dass sich Objekte eines expanded Types referentiell transparent verhalten und keine Seiteneffekte auslösen. Aus einem expanded Type heraus können außerdem Objekte – auch veränderliche – referenziert werden, sodass durch Kombination mit dem nur oberflächlichen Kopieren bzw. Klonen auch im Umgang mit expanded Types das Aliasing-Problem auftreten kann, wenn ein aus einem Objekt eines expanded Types heraus referenziertes Objekt verändert wird.

2.4.3 Java

Das Typsystem von Java unterteilt sich in *primitive Typen* und *Referenztypen*. Für die primitiven Typen wird in Java Speichersemantik verwendet, für die Referenztypen Referenzsemantik.

Benutzerdefinierte Typen sind in Java grundsätzlich Referenztypen. Entwickler können keine eigenen Typen deklarieren, die mit Speichersemantik verwendet werden.

Damit primitive Werte dort verwendet werden können, wo Referenztypen erwartet werden, stellt Java für die primitiven Typen Wrapper-Klassen zur Verfügung. Exemplare dieser Typen speichern den eingepackten primitiven Wert, stellen aber keine modifiziereden Operationen zur Verfügung. Insofern verhalten sie sich ähnlich wie Werte, jedoch nicht bei Vergleichsoperationen (s. u.). Technisch sind die Wrapper-Klassen gewöhnliche Objektklassen, die mit Referenzsemantik verwendet werden.

Seit Java 5 unterstützt Java Autoboxing, d. h. primitive Werte werden automatisch in Wrapper-Objekte eingepackt und aus diesen ausgepackt, wenn entsprechende Zuweisungen von primitiven Typen auf Referenztypen oder umgekehrt durchgeführt werden.

Zur Gleichheitsprüfung stellt Java den Operator `==` und eine von allen Objekten implementierte Method `equals` bereit. Der `==`-Operator prüft bei primitiven Typen die Operanden auf Wertgleichheit, bei Referenztypen werden die Referenzen verglichen. Letzteres gilt auch für die Wrapper-Typen.

Dies kann insbesondere in Verbindung mit Autoboxing zu verwirrendem Verhalten führen. Das Programm im Beispiel 2.3 gibt „`true false`“ aus, weil numerischen Vergleiche `<=` und `>=` die automatisch ausgepackten primitiven Werte vergleichen. Der `==`-Operator dagegen führt einen Referenzvergleich durch, weil beide Operanden Referenztypen haben. Da explizit zwei Wrapper-Objekte erstellt wurden, referenzieren `i1` und `i2` aber unterschiedliche Objekte. Hier hätte für den Vergleich die `equals`-Methode herangezogen werden müssen.

Java versucht, dieses Problem dadurch abzumildern, dass für Boolesche Werte, Zahlenwerte im Bereich −128 bis 127 und die ersten 128 `char`-Werte von der Sprache zugesichert wird, dass beim automatischen Einpacken dieser Werte in Wrapper-Objekte immer dasselbe Objekt zurückgeliefert wird. Diese Zusicherung gilt aber nur für Werte innerhalb der angegebenen Wertebereiche und, wie im Beispiel gezeigt, nicht, wenn die Wrapper-Objekte explizit erzeugt werden. In der Praxis kann ein Entwickler sich auf diese Zusicherung also kaum verlassen.

Beispiel 2.3 Auswirkungen von Auto-Unboxing in Java 5

```
Integer i1 = new Integer(42);
Integer i2 = new Integer(42);

System.out.print( (i1 <= i2 && i1 >= i2) // => true
                + " "
                + (i1 == i2));           // => false
```

Auch `String` ist in Java eine Objektklasse. String-Literale sind in Java ähnlich wie in Smalltalk als Referenzen auf Objekte der Klasse `String` definiert. Java sichert zu, dass alle String-Literale dasselbe Objekt referenzieren. Trotzdem sollten Strings nicht mit dem `==`-Operator verglichen werden, weil diese Zusicherung nur für Literale und andere konstante Ausdrücke gilt. Im Allgemeinen muss die `equals`-Methode verwendet werden, wenn String-Objekte auf Gleichheit der Zeichenkette verglichen werden sollen.

Seit der Version 5 besitzt Java mit `enum` einen zusätzlichen Typkonstruktor zur Deklaration von Aufzählungstypen. Bei Aufzählungstypen handelt es sich um Objektklassen, die mit Referenzsemantik arbeiten. Die Sprache sichert jedoch zu, dass für jede der deklarierten Aufzählungskonstanten zur Laufzeit immer genau Objekt existiert, dass von einer konstanten Klassenvariable mit dem Namen der Aufzählungskonstante referenziert wird. Diese Implementierung entspricht dem Entwurfsmuster *typesafe enum* aus [Blo01].

Weil zu jeder Aufzählungskonstanten genau ein Objekt existiert, dessen Erzeugung der Entwickler nicht bemerkt und das bis zum Ende der Laufzeit des Programms existiert, sind Aufzählungskonstanten Werten im Sinne dieser Arbeit relativ ähnlich. Allerdings sichert Java für Aufzählungskonstanten nicht zu, dass diese sich referentiell transparent verhalten. Entwickler können Aufzählungskonstanten so deklarieren, dass diese einen veränderbaren Zustand besitzen oder durch den Aufruf modifizierender Operationen an anderen Objekten Seiteneffekte auslösen.

2.4.4 C#

Das Typsystem von C# teilt sich, ähnlich dem Typsystem von Eiffel, auf in *Referenztypen* und *Werttypen*. Referenztypen werden mit Wertsemantik verwendet, Werttypen mit Speichersemantik. Im Folgenden werden Werttypen als *Speichertypen* bezeichnet, um sie von Werttypen im Sinne dieser Arbeit abzugrenzen.

Ebenso wie in Eiffel sind in C# einfache Typen als Speichertypen definiert. Auch die Erstellung von benutzerdefinierten Speichertypen ist in C# möglich. Anders als in Eiffel ist in C# die Unterscheidung zwischen Speicher- und Referenztyp aber grundsätzlich eine durch die Deklaration des Typs vorgegebene Entscheidung. Für als Referenztypen deklarierte Typen können keine Variablen mit Speichersemantik deklariert werden.

Benutzerdefinierte Speichertypen werden in C# mit Hilfe des Schlüsselwortes `struct` definiert, benutzerdefinierte Referenztypen mit dem Schlüsselwort `class`.

C# ist die einzige der hier untersuchten Programmiersprachen, die dem Entwickler die Möglichkeit gibt, Parameter per Referenz zu übergeben. Dafür wird das Schlüsselwort `ref` verwendet. Die Definition eines Referenzparameters in C# ist, dass dieser sich die Speicherstelle mit der als aktueller Parameter angegebenen Variablen teilt, d. h. beide Variablen haben immer dieselbe Belegung.

Für die Definition benutzerdefinierter Werttypen im Sinne dieser Arbeit bietet C# keine Möglichkeit.

2.4.5 Fazit

Keine der untersuchten Programmiersprachen erlaubt Entwicklern die Definition benutzerdefinierter Werttypen im Sinne dieser Arbeit. Möchte der Entwickler mit Typen arbeiten, deren Exemplare sich entsprechend den konzeptionellen Eigenschaften von Werten verhalten, so muss er Objektklassen verwenden und durch Einhaltung von Konventionen selbst für ein wertähnliches Verhalten sorgen. Die untersuchten Programmiersprachen bieten ihm dabei keine Hilfestellung und können eine Missachtung der Konventionen nicht anmahnen.

Kapitel 3

Ein neues Konzept für Werte in objektorientierten Programmiersprachen

In diesem Kapitel soll ein neues Konzept vorgestellt werden, dass als Grundlage für die Entwicklung objektorientierter Programmiersprachen dienen kann, in denen Werte und Objekte gleichermaßen unterstützt werden. Das Konzept ermöglicht sowohl die Definition benutzerdefinierter Werttypen als auch benutzerdefinierter Objekttypen.

Das Konzept basiert auf den Arbeiten von Breitling et al. [BSK+04] und Ritterbach [Rit03].

3.1 Werttypen

Ein *Werttyp* ist ein Typ, dessen Exemplare sich referentiell transparent und seiteneffektfrei verhalten. Die Menge der Werte des Typs wird durch die Definition des Typs festgelegt. Werte können zur Laufzeit weder erzeugt noch zerstört werden.

3.2 Ein Typkonstruktor für Werttypen

Programmiersprachen sollten für Werttypen einen eigenen Typkonstruktor bereitstellen, zusätzlich zu dem in objektorientierten Sprachen ohnehin vorhandenen Typkonstruktor für Objekttypen. Für mit diesem Werttypkonstruktor definierte Typen kann die Programmiersprache dann zusichern,

dass der Umgang mit Exemplaren dieser Typen den konzeptionellen Eigenschaften von Werten entspricht. Die Programmiersprache sichert also zu, dass alle Operationen von Werttypen seitenffektfrei und referentiell transparent arbeiten und dass für die Benutzung von Werten keine vorherige Erzeugung der Werte notwendig ist.

Werttypen können auf zwei Weisen definiert werden, entweder als *Aufzählungstypen* oder als *zusammengesetzte Werttypen*. Für beide Möglichkeiten sollte ein Typkostruktor bereitgestellt werden (dabei kann es sich auch um den gleichen handeln).

3.3 Aufzählungstypen

Aufzählungstypen sind Typen, deren Exemplare in der Definition des Werttyps explizit definiert werden. In der Typdefinition wird für jedes Exemplar des Werttyps ein Bezeichner definiert, über den der Zugriff auf das Exemplar erfolgt.

Ein Beispiel für einen Aufzählungstyp ist der Boolesche Typ mit seinen beiden Exemplaren „wahr“ und „falsch“.

3.4 Zusammengesetzte Werttypen

Die meisten Werttypen lassen sich nicht durch explizite Aufzählung ihrer Werte definieren, weil ihre Wertmenge unendlich oder zumindest so groß ist, dass eine Aufzählung aller Werte nicht praktikabel ist. Diese Werttypen lassen sich als zusammengesetzte Werttypen modellieren.

Ein zusammengesetzter Werttyp ist ein Werttyp, dessen Werte durch eine Kombination von Werten definiert werden. Die Werte, die ein Exemplar aus der Wertmenge des zusammengesetzten Werttyps definieren, werden als die *konstituierenden Eigenschaften* des Wertes bezeichnet.

Ein Beispiel für einen zusammengesetzten Werttyp ist der Typ „Kalendertag“, der als Kombination aus Tag, Monat und Jahr modelliert werden kann.

3.4.1 Wertbereich der konstituierenden Eigenschaften

Werden für einen zusammengesetzten Typ keine Einschränkungen definiert, so ist jede Kombination von Werten der konstituierenden Eigenschaften gültig, d. h. die Wertmenge des zusammengesetzten Typs umfasst dann das

gesamte kartesische Produkt der Wertmengen der konstituierenden Eigenschaften.

Die Definition eines zusammengesetzten Werttyps kann Einschränkungen der Wertmenge beinhalten. Die Einschränkungen werden definiert durch Gültigkeitsbedingungen für die konstituierenden Eigenschaften des Werttyps. Nur die Kombinationen von Werten, die den Gültigkeitsbedingungen entsprechen, sind *gültige* konstituierende Eigenschaften. Die Wertmenge des zusammengesetzten Werttyps umfasst nur die Werte, die durch gültige konstituierende Eigenschaften beschrieben werden.

Die Gültigkeitsbedingung für die konstituierenden Eigenschaften des Typs „Kalendertag“ ist, dass der jeweilige Tag tatsächlich ein gültiger Kalendertag ist (also z. B. nicht der 30. Februar).

3.4.2 Zugriff auf die konstituierenden Eigenschaften

Die Exemplaroperationen eines zusammengesetzten Werttyps können auf die konstituierenden Eigenschfaten des Wertes zugreifen. Sie können diese jedoch nicht verändern. Das stellt die Unveränderbarkeit von Werten sicher. Für Klienten, die einen Wert benutzen, müssen dessen konstituierende Eigenschaften nicht notwendigerweise zugreifbar gemacht werden. Ist ein Zugriff möglich, dann auch hier nur ein lesender.

3.4.3 Undefinierter Wert

In einigen Fällen kann es sinnvoll sein, die Wertmenge eines zusammengesetzten Werttyps neben den durch konstituierende Eigenschaften beschriebenen Werten zusätzlich um einen „undefinierten Wert“ zu erweitern, also um einen Wert, der die Information repräsentiert, dass in dem Kontext, in dem Werte des Typs verwendet werden, der Wert nicht definiert ist (zum Beispiel weil der Wert noch nicht bekannt ist).

Eine Alternative ist, einen Wert aus der gewöhnlichen Wertmenge eines Typs als Standardwert zu bestimmen. Dieser Wert kann von Klienten dann verwendet werden, wenn diese keinen spezifischen Wert aus der Wertmenge auswählen können.

Ob es sinnvoll ist, dass ein Werttyp einen Standardwert festlegt oder seine Wertmenge einen Sonderwert „undefiniert“ enthält, hängt von dem Werttyp und den Kontexten, in denen Werte des Typs verwendet werden, ab.

Mit dieser Frage verwandt ist, wenn Werte per Referenzsemantik verwendet werden, die Frage, ob eine Variable anstelle einer Referenz auf einen

Wert auch eine Nullreferenz enthalten kann. Auch dies wird in der Regel vom jeweiligen Kontext abhängen, in dem die Variable genutzt wird.

3.4.4 Wertwähler

Da Werte nicht erzeugt werden, stellen Werttypen auch keine Konstruktoren bereit. Der Zugriff auf Werte zusammengesetzter Typen[1] erfolgt über spezielle Operationen, die *Wertwähler*. Diese wählen den Wert aus der Wertmenge aus und geben ihn zurück.

Wird ein Wertwähler mehrfach mit den gleichen Parametern aufgerufen, so liefert jeder dieser Aufrufe denselben Wert zurück. Dies stellt einen wesentlichen Unterschied zu Konstruktoren dar, die bei jedem Aufruf stets ein neues Objekt erzeugen.

Der Aufruf eines Wertwählers zur Auswahl eines zusammengesetzten Wertes ähnelt der Angabe eines Literals zur Auswahl eines in die Sprache eingebauten Wertes. Auch bei mehrfacher Verwendung desselben Literals werden nicht verschiedene Werte erzeugt, das Literal steht immer für denselben Wert. Ein Literal kann insofern als ein spezieller Wertwähler für in die Sprache eingebaute Werttypen angesehen werden.

Jeder zusammengesetzte Werttyp besitzt mindestens eine Möglichkeit, Werte durch Angabe ihrer konstituierenden Eigenschaften auszuwählen. Diese Möglichkeit muss Klienten des Werttyps nicht unbedingt zur Verfügung gestellt werden, sie muss aber mindestens demjenigen gegeben sein, der den Werttyp und dessen Wertwähler implementiert. In der Implementierung eines Wertwählers muss letztendlich der zu wählende Wert spezifiziert werden, dies erfolgt bei zusammengesetzten Werttypen durch Angabe seiner konstituierenden Eigenschaften.

Die eigentliche Auswahl des Wertes kann vom Entwickler nicht selbst durchgeführt werden. Es handelt sich dabei um eine Sprachinterne Operation, die den Wert anhand der angegebenen konstituierenden Eigenschaften auswählt.

3.5 Gleichheit und Identität

Bei Werten sollte nicht zwischen Gleichheit und Identität unterschieden werden. In dieser Arbeit wird zur Verdeutlichung von *Wertgleichheit* gesprochen, wenn die Gleichheit bzw. Identität von Werten gemeint ist, d. h.

[1] Zur Erinnerung: der Zugriff auf die Werte von Aufzählungstypen erfolgt über die in der Typdefinition für die einzelnen Werte definierten Bezeichner (siehe Abschnitt 3.3).

Wertgleichheit liegt vor, wenn sich derselbe Wert ergibt. Zum Beispiel sind die beiden Ausdrücke $2 + 3$ und $1 + 4$ wertgleich, da beide den Wert 5 ergeben. Zwei Variablen sind dann wertgleich, wenn der Inhalt beider Variablen denselben Wert repräsentiert, unabhängig davon, wie diese Repräsentation technisch umgesetzt ist.

Weil jeder Wert konzeptionell nur einmal existiert, sollte es dem Entwickler verborgen bleiben, falls zur Laufzeit aus technischen Gründen mehrere Repräsentationen eines Wertes existieren. Eine Programmiersprache sollte daher für Werttypen keine Möglichkeit anbieten, die in Variablen dieser Typen enthaltenen Wertrepräsentationen auf Identität zu überprüfen. Stattdessen sollte ein Identitätsvergleich auf Identität des repräsentierten Wertes – also auf Wertgleichheit – prüfen.

Wird im Umgang mit Werten Speichersemantik (siehe Abschnitt 2.2) verwendet, enthalten die Variablen also direkt die Repräsentation eines Wertes, dann reicht zum Test auf Wertgleichheit ein Vergleich des Inhalts der Variablen aus. Zu beachten ist allerdings, dass dies nur gilt, wenn tatsächlich überall Speichersemantik verwendet wird – nicht nur bei den zu vergleichenden Variablen, sondern auch bei den Eigenschaften der in den Variablen enthaltenen Werte. Verwendet ein (direkt oder indirekt) in der Variablen enthaltener Wert Referenzsemantik, um den Wert einer konstituierenden Eigenschaft zu referenzieren, so reicht ein einfacher Vergleich des Inhalts der Variablen nicht mehr aus.

Dieser Fall ist in Abbildung 3.1 dargestellt. A und B beziehen sich dort auf einen Wert, der sich aus einer Zahl und einer Zeichenkette zusammensetzt, wobei die Zeichenkette per Referenzsemantik verwaltet wird. Dagegen arbeiten A und B mit Speichersemantik. A und B repräsentieren denselben Wert, die Belegung der Variablen unterscheidet sich jedoch, da sich die enthaltenen Referenzen unterscheiden. Die Gleichheitsprüfung muss hier die Referenzen verfolgen und die referenzierten Werte auf Gleichheit überprüfen.

Dasselbe gilt natürlich auch dann, wenn die Variablen selbst mit Referenzsemantik arbeiten.

In der Terminologie von Grogono und Sakkinen (vgl. [GS00]) ist die Prüfung auf Wertgleichheit eine Prüfung auf tiefe Gleichheit[2], d. h. zwei Variablen A und B repräsentieren denselben Wert, wenn $A =^{\infty} B$. Strukturelle Gleichheit ist dabei nicht erforderlich (siehe Abbildung 3.2, dort gilt $A =^{\infty} B$ aber nicht $A \cong B$). Dies gilt sowohl für Variablen mit Wert- als auch für solche mit Referenzsemantik. Diese Definition von Wertgleichheit

[2] Probleme durch zyklische Strukturen können dabei nicht auftreten, da Werte keine zyklische Struktur haben können.

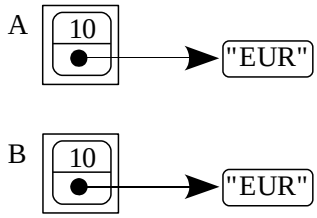

Abbildung 3.1: Nutzung von Speicher- und Referenzsemantik

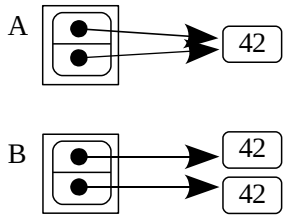

Abbildung 3.2: A und B repräsentieren trotz unterschiedlicher Struktur denselben Wert

ist also eine einheitliche Definition.

Verhält sich der von einer Programmiersprache zur Prüfung auf Identität bereitgestellte Operator beim Vergleich von Werten entsprechend der obigen Definition, so verhält er sich für eingebaute und für benutzerdefinierte Werttypen einheitlich, unabhängig davon, mit welchem Speichermodell diese verwendet werden.

3.6 Werte und verändernde Operationen

Es kann aus pragmatischen Gründen nützlich scheinen, zu erlauben, dass Werte um zusätzliche, veränderbare Attribute erweitert werden. In diesen könnte beispielsweise eine zwecks Lokalisierung veränderbare, natürlichsprachliche Repräsentation eines Wertes abgelegt werden.

Eine solche Erweiterung kann jedoch aus mehreren Gründen nicht zugelassen werden. Zum einen würde sie zu Problemen bei der Gleichheitsprüfung von Werten führen. Die zusätzlichen Attribute müssten von der Prüfung ausgeklammert werden, damit unterschiedliche aber wertgleiche Repräsen-

tationen weiterhin als wertgleich erkannt werden. Sind gleiche Objekte nicht unterscheidbar, aber veränderbar, dann müssen aber Gleichheit und Objektidentität für diese Objekte übereinstimmen (vgl. [GS00]). Genau das ist bei den Repräsentationen von Werten jedoch nicht der Fall, ansonsten würde zur Prüfung auf Gleichheit ein einfacher Referenzvergleich ausreichen.

Zum anderen würde eine solche Erweiterung das Aliasing-Problem wieder auf Werte ausweiten. Es wäre dann nicht mehr egal, ob Werte mit Wert- oder mit Referenzsemantik genutzt werden. Damit ginge ein Vorteil des in diesem Kapitel dargestellten Konzeptes, nämlich die Unabhängigkeit von der konkreten Implementierung, verloren.

3.7 Vererbung

Bei der Vererbung muss unterschieden werden zwischen der reinen Implementierungsvererbung, dem Subtyping und der Kombination aus beidem. Da die reine Implementierungsvererbung nur der Wiederverwendung von Code dient und keinen Einfluss auf die Typhierarchie hat, ist sie für diese Arbeit nicht von Interesse und soll nicht weiter betrachtet werden.

In Bezug auf Werttypen von Interesse ist die Frage, inwieweit es zulässig ist, Subtypen von einem Werttyp abzuleiten und inwieweit Werttypen Subtypen anderer Typen sein können.

3.7.1 Werttypen als Basistypen

Zunächst ist festzustellen, dass Werttypen, wenn sie überhaupt als Basistypen für andere Typen verwendet werden können, dann höchstens als Basistypen für andere *Wert*typen. Das folgt aus der Möglichkeit der polymorphen Zuweisung. Könnte ein Objekttyp Subtyp eines Werttyps sein, so könnten Exemplare des Objekttyps dort verwendet werden, wo der Werttyp erwartet wird. Die Sprache könnte für Werttypen dann keine Zusicherungen wie etwa die Sicherstellung referentieller Transparenz mehr machen.

Es bleibt die Frage, ob ein Werttyp als Basistyp eines anderen Werttyps dienen kann bzw. umgekehrt ob ein Werttyp von einem anderen Werttyp abgeleitet werden kann. Auch hier muss die Möglichkeit der polymorphen Zuweisung beachtet werden. Diese steht dem Subtyping nicht grundsätzlich im Weg. Allerdings ist bei Werttypen per Definition die Wertmenge des Typs festgelegt (siehe Abschnitt 3.1). Der Subtyp kann daher die Wertmenge nicht erweitern, dies würde der Definition des Basistyps widersprechen.

Eine Erweiterung der Wertmenge wäre auch deshalb problematisch, weil diese in der Regel eine Redefinition der Wertgleichheitsprüfung erforderlich

macht, um die neuen Werte zu berücksichtigen. Die Prüfung auf Gleichheit über Klassengrenzen hinweg ist jedoch problematisch (vgl. [GS00], [Blo01]).

Ein von einem Werttyp abgeleiteter Subtyp kann daher höchstens die vorhandenen Werte dieses Typs um zusätzliche Operationen erweitern, der Wertmenge jedoch keine weiteren Werte hinzufügen. Die neuen Operationen sind dabei natürlich nur von den Klienten nutzbar, die die Werte über den Subtyp referenzieren.

3.7.2 Werttypen als Subtypen

Werttypen können keine Subtypen von Objekttypen sein, weil Objekte erzeugt und zerstört werden können. Die Frage, inwieweit Werttypen Subtypen anderer Werttypen sein können, wurde oben bereits erläutert. Damit bleibt die Frage, inwieweit Werttypen Subtypen von reinen Schnittstellentypen (Interfaces) sein können.

Aus pragmatischer Sicht ist wünschenswert, dass Werttypen Interfaces implementieren können. Es gibt Konzepte, die von unterschiedlichen Werttypen unterstützt werden können, beispielsweise dass die Exemplare eines Typs geordnet sind. Es spricht nichts dagegen, dass ein Werttyp die zur Unterstützung solcher Konzepte notwendigen Operationen anbietet.

Grundsätzlich kann ein Werttyp also Interfaces implementieren, allerdings nur solche Interfaces, die keine Operationen definieren, die in der Schnittstelle eines Werttyps unzulässig sind. Interfaces, die von Werttypen implementiert werden können, werden im Folgenden als *Wertinterfaces* bezeichnet.

Kapitel 4

VJ: Ein Sprachentwurf basierend auf Java

In diesem Kapitel wird die im Rahmen dieser Diplomarbeit entworfene Programmiersprache *VJ* vorgestellt. VJ ist eine auf Java [GJSB05] basierende, objektorientierte Programmiersprache, die neben Objekttypen auch die Definition von Werttypen unterstützt.

Die Sprachspezifikation von VJ befindet sich im Anhang C.

4.1 Anforderungen

Dem Entwurf von VJ liegen die folgenden Anforderungen zugrunde:

- Der Umgang mit Werten soll den konzeptionellen Eigenschaften von Werten entsprechen. Das bedeutet:
  - Werte müssen sich referentiell transparent verhalten.
  - Werte sollen nicht erzeugt oder vernichtet werden können.
  - Der Identitätsprüfungs-Operator (`==`) der Sprache muss Werte auf Wertgleichheit prüfen.
- Einfache Werttypen sollen sich einfach entwickeln lassen. Ein entscheidender praktischer Nachteil der Emulation von Werttypen über normale Objektklassen ist, dass bereits die Entwicklung einfacher Werttypen mit einem überproportional hohen Aufwand verbunden ist, weil viele Konventionen beachtet werden müssen bzw. – wenn ein Rahmenwerk

verwendet wird – alle Vorgaben des Rahmenwerks erfüllt werden müssen. Diesen Nachteil sollte eine Programmiersprache, die Werttypen als explizites Sprackkonstrukt unterstützt, nicht aufweisen.

- Die neue Sprache soll quelltextkompatibel zu Java sein, d. h. jeder gültige Java-Quelltext soll möglichst auch gültiger VJ-Quelltext sein.
- Außerdem soll die Sprache kompatibel zu bestehendem Bytecode bleiben, damit existierende, in Java programmierte Bibliotheken ohne Neuübersetzung aus VJ-Programmen heraus genutzt werden können. Insbesondere muss es möglich sein, die Java-Standardbibliothek zu nutzen.
- Neue Sprachkonstrukte sollten sich in ihrer Syntax an der Java-Syntax orientieren; die neue Sprache sollte sich „wie Java anfühlen“.

Die Anforderungen lassen sich nicht vollständig erfüllen. Durch die Einführung zusätzlicher Schlüsselwörter sinkt beispielsweise die Quelltextkompatibilität. Während der Entwicklung von VJ dienten sie daher nicht als starre Vorgaben, sondern als Entwurfsrichtlinien und Zielvorgabe.

4.2 Objektklassen

Die Deklaration und Benutzung von Objektklassen unterscheidet sich in VJ nicht von Java.

4.3 Aufzählungstypen

Wie im vorangegangenen Kapitel erläutert, können Werttypen entweder als Aufzählungstypen oder als zusammengesetzte Werttypen definiert werden. Zur Definition von Aufzählungstypen bietet Java seit der Version 5 den Typkonstruktor `enum` an (vgl. [GJSB05, § 8.9]).

In mit dem `enum`-Typkonstruktor erstellten Aufzählungstypen können allerdings veränderbare Felder und Methoden, die den Zustand eines Exemplars ändern, deklariert werden. Grundsätzlich sind Zustandsänderungen bei Werten höchstens dann zulässig, wenn für die Wertrepräsentationen Gleichheit und Identität übereinstimmen, d. h. für jeden Wert höchstens eine Repräsentation existiert (siehe Abschnitt 3.6). Genau dies garantiert Java für Aufzählungstypen. Jeder deklarierten Aufzählungskonstanten ist zur Laufzeit genau ein Exemplar des jeweiligen Typs zugeordnet; zwei Variablen, die

sich auf dieselbe Aufzählungskonstante beziehen, enthalten also immer die gleiche Referenz.

Allerdings wird für Aufzählungstypen keine Seiteneffektfreiheit garantiert. Aufzählungstypen können Referenzen auf Objekte verwenden und ihre Methoden können verändernde Operationen an Objekten aufrufen. Daher ist der `enum`-Typkonstruktor kein Werttypkonstruktor im eigentlichen Sinne.

Aus praktischen Gründen ist es aber dennoch nicht sinnvoll, die Sprache um einen zusätzlichen Typkonstruktor für Wert-Aufzählungstypen zu erweitern, der sich von dem bereits vorhandenen lediglich dadurch unterscheidet, dass er keine Benutzung von Objektreferenzen zulässt. Es wäre zu befürchten, dass Entwickler den zusätzlichen Typkonstruktor lediglich als eingeschränkte, weniger mächtige Variante des `enum`-Konstruktors wahrnehmen würden und der zusätzliche Typkonstruktor in der Praxis nicht verwendet würde. Außerdem kann davon ausgegangen werden, dass die meisten Aufzählungstypen tatsächlich nur zur Aufzählung von Konstanten erstellt werden und keine Methoden mit Seiteneffekten enthalten. Die Einführung eines zweiten Typkonstruktors für Aufzählungstypen hätte also nur einen geringen Zusatznutzen, der die gestiegene Komplexität der Sprache nicht rechtfertigt.

VJ führt daher keinen weiteren Typkonstruktor für Aufzählungstypen ein, sondern greift auf den `enum`-Typkonstruktor zurück und behandelt mit diesem erstellte Typen als Werttypen.

4.4 Zusammengesetzte Werttypen

Die Erstellung zusammengesetzter Werttypen wird von Java nicht unterstützt. VJ führt daher einen neuen Typkonstruktor für diese Typen ein: `valueclass`.

Die mit diesem Typkonstruktor erstellten Typen sind Referenztypen, d. h. sie werden per Referenzsemantik verwaltet. Variablen vom Typ einer Wertklasse kann `null` zugewiesen werden.

Es wurde entschiedenen, für diesen Typkonstruktor ein neues Schlüsselwort zu reservieren, um auch syntaktisch deutlich zu machen, dass es sich um ein von Objektklassen getrenntes Konzept handelt. Die Alternative, lediglich einen Modifier für den vorhandenen Objektklassen-Konstruktor zu definieren („`const class`“ oder Ähnliches) könnte den Eindruck erwecken, es handele sich bei Wertklassen lediglich um Varianten von Objektklassen. Das ist jedoch nicht der Fall.

Wünschenswert wäre es, auch den vorhandenen Typkonstruktor `class`

umzubenennen in `objectclass`, um die konzeptionelle Unterscheidung noch deutlicher hervorzuheben. Durch diese Änderungen würden jedoch fast alle existierenden Java-Quelltexte zu VJ inkompatibel, daher kommt dieser Schritt nicht in Frage.

Die Quelltextkompatiblität wird durch das Reservieren des Wortes „`valueclass`“ kaum negativ beeinflusst. Gemäß den Konventionen für Bezeichner in Java (vgl. [GJSB05, § 6.8]) sollte es in dieser Form ohnehin nicht als Bezeichner verwendet werden, stattdessen müsste entweder „`valueClass`“ (bei Methoden bzw. Variablen) oder „`ValueClass`“ (bei Typen) verwendet werden. Da Java und VJ Groß- und Kleinschreibung unterscheiden, bleiben diese Bezeichner in VJ weiterhin gültig. Lediglich für Paketnamen wäre „`valueclass`“ die empfohlene Schreibweise, dass dieser Bezeichner für ein Paket verwendet wird, ist jedoch unwahrscheinlich.

Syntaktisch sind Wertklassendeklarationen ähnlich aufgebaut wie Objektklassendeklarationen. Die einzelnen Bestandteile einer Wertklassendeklaration werden im Folgenden besprochen.

4.5 Konstituierende Eigenschaften

Die konstituierenden Eigenschaften eines zusammengesetzten Werttyps werden syntaktisch ähnlich wie die Felder einer Objektklasse deklariert. Allerdings sind die meisten der Modifier, die bei Feldern von Objektklassen zulässig sind (vgl. [GJSB05, § 8.3.1]), bei der Deklaration von konstituierenden Eigenschaften nicht zulässig. Im Einzelnen:

- `protected` ist nicht zulässig, weil von Wertklassen ohnehin nicht geerbt werden kann.
- `static` wird zur Deklaration von Klassenvariablen verwendet.
- `final` ist nicht zulässig, weil die konsituierenden Eigenschaften eines Werts ohnehin unveränderlich sind. Um irreführende Quelltexte durch eine inkonsistente Benutzung zu vermeiden, ist auch die optionale Verwendung von `final` nicht zulässig:

```
public valueclass Example {
    final int a;  // nicht zulässig!
    int b;
}
```

Dieses Quelltextbeispiel erweckt den Eindruck, es handele sich bei `b` im Gegensatz zu `a` nicht um ein unveränderliches Feld. Tatsächlich sind jedoch beides konstituierende Eigenschaften.

- `transient` dient bei Objektklassen zur Kennzeichnung von Feldern, die nicht Teil des persistenten Zustands eines Objektes sind. Bei zusammengesetzten Werttypen wird jeder Wert durch seine konstituierenden Eigenschaften definiert, zur Auswahl eines Wertes müssen alle seine konstituierenden Eigenschaften angegeben werden. Daher müssen, wenn ein Wert persistent gespeichert werden soll, auch alle konstituierenden Eigenschaften des Wertes gespeichert werden.

- `volatile` hat nur bei veränderbaren Feldern eine Bedeutung und ist daher auf konstituierende Eigenschaften nicht anwendbar.

Die Definition eines zusätzlichen Schlüsselwortes, welches kennzeichnet, dass es sich bei der Deklaration um die Deklaration einer konstituierenden Eigenschaft handelt, ist nicht erforderlich, da in einer Wertklasse keine sonstigen Felder deklariert werden können (mit Ausnahme der Deklaration von Klassenvariablen, die durch das Schlüsselwort `static` gekennzeichnet sind).

Der Zugriff auf die konstituierenden Eigenschaften wird, abgesehen davon, dass der Modifier `protected` nicht zur Verfügung steht, ebenso kontrolliert wie der Zugriff auf Felder in Objektklassen, d. h. der Zugriff ist für alle Klassen des gleichen Pakets möglich und kann mit den Schlüsselwörtern `public` bzw. `private` auf allen Klassen ermöglicht bzw. auf die eigene Klasse beschränkt werden.

Dabei ist zu beachten, dass konstituierende Eigenschaften über den ihnen zugeordneten Parameter des automatisch erstellten Wertwählers immer indirekt die Schnittstelle des Typs beeinflussen, auch dann, wenn sie `private` sind. Dennoch kann es sinnvoll sein, das Lesen einer konstiuierenden Eigenschaft durch andere Klassen durch die Deklaration als `private` zu verbieten, wenn diese konstituierende Eigenschaft nicht losgelöst von den anderen konstituierenden Eigenschaften verwendet werden soll. Zum Beispiel kann es bei Messwerten oder Mengenangaben unerwünscht sein, den Zahlenwert ohne die zugehörige Maßeinheit zu verwenden.

Die Deklaration von konstituierenden Eigenschaften darf – ebenfalls im Gegensatz zur Deklaration von Feldern in Objektklassen – keinen Initialisierer enthalten. Da konstituierende Eigenschaften nicht verändert werden können, würde der Initialisierer[1] die konstituierende Eigenschaft für alle Werte

[1] Strenggenommen ist der Begriff „Initialisierer“ in diesem Kontext falsch. Weil Werte konzeptionell nicht erzeugt werden, werden sie auch nicht im eigentlichen Sinne initialisiert.

des Werttyps auf denselben Wert festlegen. Dann würde die Eigenschaft die einzelnen Werte aber nicht definieren, weil sie kein Unterscheidungsmerkmal wäre; sie wäre dann auch keine konstituierende Eigenschaft im Sinne des Konzeptes.

Werden in einem Werttyp Konstanten benötigt, so können dafür Klassenkonstanten verwendet werden.

4.6 Wertwähler

In VJ wird besitzt jeder zusammengesetzte Werttyp implizit einen Wertwähler, der auf der Basis der deklarierten konstituierenden Eigenschaften definiert ist. Der Wertwähler besitzt für jede konstituierende Eigenschaft genau einen formalen Parameter gleichen Typs.

Beispiel: Die in Beispiel 4.1 deklarierte Wertklasse besitzt einen Wertwähler mit zwei Parametern, dabei hat der erste Parameter den Typ `int` und der zweite den Typ `Currency`[2]. Die Sichtbarkeit des Wertwählers ist `public`.

Beispiel 4.1 Eine einfache Wertklasse mit zwei konstituierenden Eigenschaften

```
public valueclass Money {
    private int amount;
    public Currency currency;
}
```

4.6.1 Explizit deklarierte Wertwähler

Wie in Abschnitt 3.4.4 erläutert, muss für jeden zusammengesetzten Werttyp eine Möglichkeit zur Verfügung stehen, Werte durch Angabe ihrer konstituierenden Eigenschaften auszuwählen. Diese Möglichkeit besteht in VJ durch den implizit vorhandenen Wertwähler. Dieser steht dabei Klienten der Wertklasse immer zur Verfügung. Es gibt keine Möglichkeit, den Zugriff auf den Wertwähler zu kontrollieren. Um den Zugriff kontrollieren zu können, müsste der Wertwähler explizit deklariert werden, in der Deklaration könnten dann die Schlüsselwörter zur Zugriffskontrolle (`public` und `private`) verwendet werden, um die Zugriffsmöglichkeiten festzulegen. Syntaktisch könnte sich

[2]Bei `Currency` muss es sich natürlich ebenfalls um einen Werttyp handeln.

eine solche Deklaration an der Syntax zur Deklaration von Konstruktoren in Objektklassen orientieren.

Würden Wertwähler explizit deklariert, müsste dem Entwickler ein anderer Mechanismus bereitgestellt werden, über den er die konstituierenden Eigenschaften des zu wählenden Wertes angeben, also die eigentlich Wertauswahl durchführen, kann.

Auch hier könnte man sich an der bei Konstruktoren verwendeten Syntax orientieren und die Syntax von Zuweisungen übernehmen, wie etwa im Beispiel 4.2, das eine Wertwähler-Deklaration für die Wertklasse `Money` aus Beispiel 4.1 zeigt.

Beispiel 4.2 Eine Konstruktor-ähnliche Syntax für Wertwähler (kein gültiger VJ-Code)

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

Diese Syntax ist jedoch äußerst irreführend, da es sich tatsächlich nicht um Zuweisungen, sondern um die Auswahl eines Wertes handelt.

Eine andere Möglichkeit wäre die Verwendung des Schlüsselworts `this`, ähnlich wie in einem expliziten Konstruktoraufruf innerhalb eines Konstruktors (siehe Beispiel 4.3).

Beispiel 4.3 Verwendung des Schlüsselwortes `this` in Wertwählern (kein gültiger VJ-Code)

```
public Money(int amount, Currency currency) {
    this(amount, currency);
}
```

Auch diese Syntax ist aus mehreren Gründen irreführend. Erstens kann sie den Eindruck erwecken, es würde eine Endlosrekursion vorliegen, weil der Wertwähler sich selbst aufruft. Bei einem Konstruktor in einer Objektklasse wäre dies der Fall (und ein Übersetzungszeit-Fehler). Zweitens muss der explizite Konstruktoraufruf in Konstruktoren die erste Anweisung im Rumpf des Konstruktors sein, bei einem Wertwähler dagegen muss die eigentliche Wertauswahl als letzter Schritt erfolgen. Die durch die Syntax angedeutete, tatsächlich aber nicht vorhandene Ähnlichkeit, könnte hier leicht zu Programmierfehlern führen.

4.6.2 Alternative Wertwähler

Da Wertwähler in VJ nicht explizit deklariert werden, besteht keine Möglichkeit, alternative Wertwähler zu definieren, die einen Wert basierend auf einer anderen Repräsentation des Wertes als durch seine konstituierenden Eigenschaften auswählen.

Solche alternativen Wertwähler müssten in ihrer Implementierung zunächst die konstituierenden Eigenschaften des Wertes aus der alternativen Repräsentation bestimmen und dann durch Angabe der ermittelten konstituierenden Eigenschaften die eigentliche Auswahl des Wertes durchführen (zum Problem der dafür zu verwendenden Syntax siehe oben).

Anstelle von alternativen Wertwählern können Wertklassen Fabrikmethoden bereitstellen, die die Funktionalität von alternativen Wertwählern bereitstellen. Beispiel 4.4 zeigt eine mögliche Fabrikmethode für den Werttyp `Money` aus Beispiel 4.1.

Beispiel 4.4 Eine mögliche Fabrikmethode für den Werttyp `Money`

```
public valueclass Money {
    ...
    public static Money euros(int amount) {
        return const Money(amount, Currency.EUR);
    }
}
```

4.7 Methoden

Methoden werden in Wertklassen ebenso deklariert wie in Objektklassen. Allerdings gelten für die Methoden von Wertklassen einige Einschränkungen:

- Der Rückgabetyp der Methode muss ein Werttyp oder ein Wertinterface sein.
- Als Parametertypen sind nur Werttypen zulässig.
- In der Implementierung der Methode dürfen keine Objekte erzeugt und keine Klassenmethoden von Objektklassen aufgerufen werden.
- Die Methode darf keine geprüften Exceptions werfen.

Diese Einschränkungen gelten sowohl für Klassen- als auch für Exemplarmethoden.

Der Rückgabetyp der Methode darf nicht `void` sein. Diese Einschränkung folgt daraus, dass Werte sich referentiell transparent und seiteneffektfrei verhalten. Eine Operation, die nichts zurückgibt, muss aber, wenn sie nicht wirkungslos und damit überflüssig ist, irgendeinen Seiteneffekt auslösen.

Da Werte keine Objekte referenzieren und Methoden in ihrer Implementierung keine Objekte erzeugen können, kann eine Methode nur Werte, aber keine Objekte zurückgeben. Daher können als Rückgabetyp auch nur Werttypen oder Wertinterfaces verwendet werden.

Als Typen der formalen Parameter dürfen nur Werttypen verwendet werden. Auch diese Einschränkung dient der Sicherstellung der referentiellen Transparenz und Seiteneffektfreiheit von Werten. Würde eine Methode eines Werttyps den Zustand eines Objektes lesen, dann wäre ihr Ergebnis vom Zustand des Objektes abhängig, damit wäre die Methode jedoch nicht referentiell transparent. Würde eine Methode den Zustand eines Objektes verändern, dann wäre sie nicht frei von Seiteneffekten.

Wertinterfaces sind als Parametertypen nicht zulässig, weil diese Interfaces nicht nur von Werttypen, sondern auch von Objekttypen implementiert werden können. Einer Methode, die einen Parameter vom Typ eines Wertinterfaces hat, können also auch Objekte übergeben werden. Dies soll für Werte aber unzulässig sein. Als Rückgabetyp dagegen können Wertinterfaces verwendet werden, weil Werte nur andere Werte zurückgeben können.

Aus dem gleichen Grund dürfen auch in der Implementierung keine Objekte erzeugt werden. Dies wäre ein Seiteneffekt, zudem würde bei jeder Ausführung der Methode ein weiteres, neues Objekt erzeugt, sodass die referentielle Transparenz nicht sichergestellt werden könnte. Der Aufruf von Klassenmethoden einer Objektklasse ist ebenfalls nicht zulässig, da nicht sichergestellt ist, dass diese sich seiteneffektfrei und referentiell transparent verhalten würde.

Weil die genannten Einschränkungen auch für die Klassenmethoden von Wertklassen gelten, ist jedoch der Aufruf dieser Methoden zulässig.

Die Methoden von Wertklassen dürfen keine geprüften Exceptions werfen. Geprüfte Exceptions werden in der Regel verwendet, um Fehler zu signalisieren, deren Auftreten der Entwickler nicht ausschließen kann, da die Fehlerursache außerhalb des Programms liegt, zum Beispiel Ein-/Ausgabefehler. Solche Fehler können im Umgang mit Werten jedoch nicht auftreten. Der Erfolg des Aufrufs einer bestimmten Methode mit gegebenen Werten hängt nicht von äußeren Umständen ab. Tritt dennoch ein Fehler auf, so handelt es sich um einen Programmierfehler, der durch eine ungeprüfte Exception

signalisiert werden sollte.

Ungeprüfte Exceptions können auch im Umgang mit primitiven Werten auftreten (z. B. eine `ArithmeticException` bei einer Division durch Null).

4.8 Definition des Wertbereichs der konstituierenden Eigenschaften

Gültigkeitsbedingungen für die konstituierenden Eigenschaften können in VJ durch die Deklaration einer Klassenmethode `isValid` definiert werden. Die Methode muss in Quelltextreihenfolge der konstituierenden Eigenschaften für jede konstituierende Eigenschaft genau einen Parameter vom Typ der Eigenschaft besitzen, d. h. die Parameter der Methode entsprechen den Parametern des automatisch vorhandenen Wertwählers.

Wenn eine solche Methode deklariert ist, dann wird sie beim Versuch, einen Wert auszuwählen, automatisch aufgerufen. Ergibt dieser Aufruf, dass die übergebenen Werte keine gültigen konstituierenden Eigenschaften sind, der zu wählende Wert also nicht existiert, dann schlägt die Auswahl des Wertes mit einer entsprechenden Exception fehl.

Dadurch, dass die Gültigkeitsprüfung in einer Klassenmethode durchgeführt wird, kann sie von Nutzern der Klasse aufgerufen werden, bevor ein Wert ausgewählt wird, wenn nicht sicher ist, ob die konstituierenden Eigenschaften gültig sind, zum Beispiel, weil diese aus einer Benutzereingabe stammen. Die Gültigkeitsprüfung in einer Exemplarmethode anzusiedeln wäre aber auch logisch falsch, denn die Gültigkeit der konstituierenden Eigenschaften bestimmt ja erst, ob ein zugehöriger Wert überhaupt existiert. An einem existierenden Wert abfragen zu können, ob dieser Wert existiert, ergäbe keinen Sinn.

Als Alternative zu Methoden, welche die Gültigkeit prüfen, könnte eine Programmiersprache auch Mittel bereitstellen, mit denen die Gültigkeitsbedingungen deklarativ spezifiziert werden können. Vor allem bei einfachen Bedingungen wie Minima, Maxima oder der Einschränkung, dass der Wert einer konstituierenden Eigenschaft nicht `null` sein darf, könnte dies die Entwicklung vereinfachen und die Lesbarkeit des Quelltextes verbessern.

In der Praxis werden allerdings nicht nur solche einfachen Bedingungen benötigt, sondern es kann auch vorkommen, dass die Gültigkeit einer konstituierenden Eigenschaft vom Wert einer anderen abhängt oder dass die Gültigkeit des Wertes einer konstituierenden Eigenschaft zum Beispiel durch die Kontrolle einer im Wert enthaltenen Prüfziffer geprüft werden muss. Sollen sich auch solche Bedingungen deklarativ ausdrücken lassen, so müsste

zunächst eine entsprechend komplexe Sprache zur Formulierung solcher Bedingungen entwickelt und diese dann in die eigentliche Programmiersprache integriert werden.

Für die Sprache VJ wurde die Möglichkeit, Gültigkeitsbedingungen deklarativ zu spezifizieren, daher nicht weiter untersucht. Der gewählte Ansatz über eine Methode zur Prüfung stellt Entwicklern praktisch den gesamten Sprachumfang von Java zur Verfügung – nur Objektklassen dürfen nicht verwendet werden, von der Existenz oder dem Zustand von Objekten kann die Gültigkeit aber ohnehin nicht abhängen – und ermöglicht es ihnen so, auch sehr komplexe Prüfungen zu implementieren, ohne dass sie eine komplett neue Sprache erlernen müssen.

4.9 Vererbung

VJ lässt keine Vererbung von Wertklassen zu. Weder können Wertklassen als Basisklassen anderer Wert- oder Objektklassen verwendet werden, noch können umgekehrt Wertklassen andere Klassen erweitern. Diese Einschränkung wurde aufgrund der in Abschnitt 3.7 diskutierten Probleme im Zusammenhang mit Vererbung gewählt.

Erlaubt würden könnten demnach höchstens solche Subklassen, die den Basistyp um zusätzliche Operationen erweitern, jedoch ansonsten nicht verändern. Eine derart eingeschränkte Vererbung hat jedoch auch nur noch einen geringen praktischen Nutzen. Die gleiche Funktionalität lässt sich in der Regel auch über in anderen Klassen deklarierte Klassenmethoden erreichen. Dies entspricht auch dem in Java verwendeten Ansatz, nützliche Zusatzoperationen zum Umgang mit Objekten als Klassenmethoden bereitzustellen[3].

Erlaubt wird in VJ jedoch die Implementierung von Interfaces, wenn es sich dabei um Wertinterfaces (siehe Abschnitt C.4) handelt. Die Implementierung von Interfaces zu erlauben ist schon aus praktischen Gründen sinnvoll, um beispielsweise das Interface `java.lang.Comparable<T>` implementieren zu können. Aber auch konzeptionell spricht nichts gegen die Implementierung von geeigneten Interfaces (siehe Abschnitt 3.7.2).

Die Einschränkungen, die ein Interface einhalten muss, um als Wertinterface verwendet werden zu können, entsprechen den Einschränkungen für die Parameter- und Rückgabetypen von Methoden in Wertklassen, d. h. ein Wertinterface darf keine Methoden deklarieren, die eine Wertklasse nicht

[3]Zum Beispiel stellt die Klasse `java.util.Collections` zusätzliche Funktionen zum Umgang mit Sammlungen zur Verfügung.

implementieren kann. Ausnahmnsweise wird jedoch die Deklaration der in `Object` deklarierten Methoden zugelassen, weil diese Methoden von jedem Interface mindestens implizit deklariert werden (vgl. [GJSB05, § 9.2]).

4.10 Polymorphe Verwendung von Werten

Weil Wertklassen Referenztypen sind und weil sie Wertinterfaces implementieren können, kann nicht verhindert werden, dass ein Wert einer Variablen vom Typ `Object` zugewiesen wird. Diese Zuweisung ist mindestens indirekt möglich, indem der Wert zunächst einer Variablen vom Typ eines von dem Wert implementierten Wertinterfaces zugewiesen wird und anschließend diese Variable einer Variablen vom Typ `Object`. In VJ wird auch die direkte Zuweisung eines Werts zu einer Variablen vom Typ `Object` nicht verhindert.

In einer Wertklasse können allerdings nicht die in `Object` deklarierten Methoden überschrieben werden, mit Ausnahme von `toString`. Das verhindert, dass Entwickler die Werteigenschaften durch eine fehlerhafte Implementierung der Methoden zerstören können.

Die Methode `equals` ist die einzige Methode einer Wertklasse, die einen Parameter besitzt, dessen Typ kein Werttyp ist. Allerdings wird für diese Methode automatisch eine Implementierung erstellt, die sicherstellt, dass die Werte sich dennoch entsprechend den konzeptionellen Eigenschaften von Werten verhalten. Wird der Methode `null` oder ein Objekt einer Objektklasse übergeben, so ist das Ergebnis des Vergleichs immer `false`. Nur wenn ein Wert übergeben wird, wird dieser auf Wertgleichheit verglichen. Die Methode `hashCode` wird automatisch passend zu `equals` implementiert.

Die Methode `getClass` ist die einzige Methode einer Wertklasse, deren Rückgabetyp (`Class<T>`) ein Objekttyp ist. Objekte vom Typ `Class<T>` haben jedoch keinen veränderbaren Zustand, insbesondere lässt sich über diese Objekte nicht zur Laufzeit die Wertklasse modifizieren.

Das Klonen eines Wertes durch Aufruf von `clone` ist nicht möglich, weil Werte konzeptionell nicht geklont werden können, denn jeder Wert existiert nur einmal.

Die Methode `finalize` sollte von Entwicklern ohnehin nicht manuell aufgerufen werden, sondern sie wird vom Garbage Collector automatisch aufgerufen, wenn ein Objekt zerstört wird. Da Werte konzeptionell niemals zerstört werden, können bei der Zerstörung von Werten keine Aktionen ausgelöst werden. Daher kann diese Methode vom Implementierer einer Wertklasse nicht überschrieben werden.

Die einzige in `Object` deklarierte Methode, die von Entwicklern in der

Implementierung von Wertklassen überschrieben werden kann, ist `toString`. Diese Methode gibt einen Werttyp zurück und bekommt keine Parameter, sie entspricht also den Einschränkungen für die Methoden von Wertklassen. Sie kann überschrieben werden, um eine textuelle Repräsentation von Werten zu definieren.

Jeder Wert verfügt außerdem über die in `Object` deklarierten Methoden `wait`, `notify` und `notifyAll`, die zur Steuerung nebenläufiger Programme verwendet werden. Diese Methoden sollten an Werten nicht verwendet werden, da sie nicht frei von Seiteneffekten arbeiten.

4.11 Wertwähler-Aufruf

Durch den Aufruf des Wertwählers einer Wertklasse wird ein Wert aus der Wertmenge der Wertklasse ausgewählt. Ein Wertwähler-Aufruf ist eine `<PrimaryExpression>` (siehe [GJSB05, § 15.8]), d. h. ein Wertwähler-Aufruf kann überall dort verwendet werden, wo auch Literale angegeben werden können. Das Ergebnis des Ausdrucks ist der ausgewählte Wert.

Syntaktisch ähnelt der Aufruf eines Wertwählers dem Aufruf eines Konstruktors einer Objektklasse (siehe Beispiel 4.5), als Schlüsselwort wird jedoch nicht `new` sondern `const` verwendet. Dadurch soll deutlich gemacht werden, dass bei der Auswahl eines Wertes dieser Wert konzeptionell nicht erzeugt wird.

Beispiel 4.5 Aufruf des Wertwählers der Wertklasse `Money`

```
Money m = const Money(10, Currency.EUR);
```

Das Schlüsselwort `const` ist in Java bereits ein reserviertes Wort, wird dort jedoch nicht verwendet (vgl. [GJSB05, § 3.9]). Durch die Verwendung dieses Schlüsselwortes in VJ kann es also zu keinen Inkompatibilitäten mit bestehenden Java-Quelltexten kommen.

Als Alternative Schlüsselwörter wurden `get`, `value` und `pick` in Betracht gezogen. `get` und `value` werden jedoch in der Java-Standardbibliothek als Bezeichner für Methoden und Felder verwendet (z. B. die Methode `get(int)` in `java.util.List` und das Feld `value` in `java.sql.DriverPropertyInfo`, vgl. [Sun04, API Specification]). Diese als Schlüsselwort zu reservieren, würde dazu führen, dass VJ inkompatibel mit der Java-Standardbibliothek würde. `pick` wird in der Java-Bibiliothek nicht verwendet, es handelt sich dabei jedoch ebenfalls um ein relativ kurzes Wort, sodass ein entsprechend hohes Risiko besteht, dass dieses in existierendem Code bereits verwendet wird.

Daher wurde entschieden, dass bereits existierende Schlüsselwort `const` zu verwenden.

Anstelle der konstruktorähnlichen Syntax wäre es wünschenswert, eine Syntax zu verwenden, die der Syntax für Literale ähnelt. Dies für benutzerdefinierte Typen zu ermöglichen, würde jedoch auch eine benutzerdefinierte Syntax erfordern. Die Syntax durch den Entwickler beeinflussbar zu gestalten, hätte sehr weitreichende Folgen, da die Anpassungen der Syntax bereits in der lexikalischen und syntaktischen Analyse des Quelltextes berücksichtigt werden müssten. Im Entwurf von VJ wurde diese Idee daher nicht weiter verfolgt.

Selbstverständlich haben Entwickler aber die Möglichkeit, eine Fabrikmethode zu erstellen, die eine Zeichenkette analysiert und darauf basierend einen Wert zurückgibt.

4.12 Gleichheitsoperatoren

Die Gleichheitsoperatoren `==` und `!=` prüfen in VJ ihre Operanden auf Wertgleichheit, wenn es sich um Werte handelt.

Dazu werden die Gleichheitsoperatoren in entsprechende Aufrufe der `equals`-Methode übersetzt. In Verbindung mit der automatisch erstellten Implementierung der `equals`-Methode sorgt dies dafür, dass zur Laufzeit ein Vergleich der Repräsentationen auf tiefe Gleichheit durchgeführt wird. Dies entspricht der konzeptionellen Definition von Wertgleichheit (siehe Abschnitt 3.5).

Sind die Operanden Objekte, dann wird wie in Java ein Vergleich auf Objektidentität durchgeführt, also geprüft, ob beide Operanden dasselbe Objekt referenzieren (vgl. [GJSB05, § 15.21.3]).

Weil Werte teilweise polymorph verwendet werden können, kann zur Übersetzungszeit anhand der statischen Typen der Operanden nicht immer festgestellt werden, ob die Operanden zur Laufzeit Werte oder Objekte referenzieren werden. Daher gilt:

- Ist zur Übersetzungszeit aufgrund der statischen Typen der Operanden sicher, dass die Operanden nur Objekte referenzieren können, wird der Gleichheitsoperator als Operation zur Prüfung auf Objektidentität entsprechend dem `==`-Operator (bzw. `!=`-Operator) in Java übersetzt.

- Ist zur Übersetzungszeit aufgrund der statischen Typen der Operanden sicher, dass die Operanden nur Werte referenzieren können, wird

der Gleichheitsoperator als Operation zur Prüfung auf Wertgleichheit übersetzt.

- Ansonsten wird der Operator so übersetzt, dass zur Laufzeit zunächst geprüft wird, ob die Operanden Objekte oder Werte referenzieren, und dann die dem Typ entsprechende Prüfung durchgeführt wird.

Die Regel aus der Java-Spezifikation, dass der Typ eines Operanden durch eine Cast-Umwandlung in den Typ des anderen konvertierbar sein muss (vgl. [GJSB05, § 15.21.3]), gilt auch in VJ, d. h. der Vergleich ist unzulässig, wenn die beiden Operanden nicht gleich sein können.

4.13 Integration mit der Java-Standardbibliothek

Neben Aufzählungstypen – die in VJ als Werttypen betrachtet werden – existieren in der Java-Standardbibliothek noch einige weitere Typen, bei denen es wünschenswert ist, sie als Werttypen nutzen zu können.

4.13.1 `java.lang.String`

Dies gilt vor allem für die Klasse `String`. Objekte vom Typ `String` sind in Java bereits als unveränderliche Objekte implementiert, sodass ihr Verhalten dem von Werten ähnelt. Daher wird `String` in VJ als Werttyp behandelt.

Das ermöglicht es Werttypen, Zeichenketten vom Typ `String` als Rückgabe- und Parametertypen zu verwenden. Werte können somit in textuelle Repräsentationen umgewandelt oder (in entsprechenden Fabrikmethoden) aus textuellen Repräsentationen berechnet werden. Diese textuellen Repräsentationen können problemlos mit Objekten ausgetauscht werden, die Zeichenketten vom Typ `String` erwarten oder liefern.

Auch die Tatsache, dass Exemplare vom Typ `String` durch die Angabe von Literalen erhalten werden, die Nutzung von `String` in diesem Punkt also der Nutzung von primitiven Typen ähnelt, spricht dafür, in `String` einen Werttyp zu sehen.

Die Behandlung von `String` als Werttyp erstreckt sich auf die Interpretation der Gleichheitsoperatoren. Wenn `String`-Objekte verglichen werden, wird hier in VJ deren Wertgleichheit geprüft, d.h. das Ergebnis des Vergleichs ist `true`, wenn die Operanden unterschiedliche `String`-Objekte referenzieren, diese aber dieselbe Zeichenkette enthalten[4].

[4] Natürlich ist das Ergebnis auch `true`, wenn beide Operanden dasselbe Objekt referenzieren.

Außerdem ist es in VJ möglich, die „Konstruktoren“ von `String` (die aus konzeptioneller Sicht keine Konstruktoren, sondern Wertwähler sind) mit dem Schlüsselwort `const` aufzurufen. Aus Kompatibilitätsgründen bleibt aber auch die Verwendung von `new` weiterhin zulässig, ebenso natürlich die Verwendung eines Literals.

4.14 Zusammenfassung

Der vorgestellte Sprachentwurf erfüllt die am Anfang aufgestellten Anforderungen weitgehend. An einigen Stellen mussten jedoch Kompromisse eingegangen werden, um eine Integration mit bestehenden Sprachelementen und Klassen von Java zu ermöglichen. So wird zum Beispiel die Verwendung von Aufzählungstypen in Wertklassen erlaubt, obwohl kein seiteneffektfreies Verhalten der Aufzählungskonstanten garantiert ist.

Die Sprache vereinfacht besonders die Deklaration einfacher Werttypen deutlich. Dies zeigt zum Beispiel die Wertklasse im Beispiel 4.1. Würde der Entwickler diese Klasse mit wertähnlichem Verhalten in Java implementieren wollen, so müsste er zusätzlich mindestens einen Konstruktor und die Methoden `equals` und `hashCode` implementieren. Diese Arbeit nimmt VJ dem Entwickler ab und kann dadurch gleichzeitig zusichern, dass die Werte sich tatsächlich referentiell transparent verhalten.

Kapitel 5

Implementierung eines Übersetzers für VJ

Im Rahmen dieser Arbeit wurde mit der Implementierung eines Übersetzers für die im vorangegangenen Kapitel vorgestellte Programmiersprache VJ begonnen.

Der Übersetzer ist als Prä-Compiler implementiert, der den VJ-Quelltext in Java-Quelltext übersetzt. Dieser Java-Quelltext kann anschließend mit einem Java-Übersetzer in Bytecode übersetzt werden. Entwickler schreiben ihre Programme ausschließlich in VJ, eine genaue Kenntnis der Übersetzung in Java-Quelltext ist nicht erforderlich. Die vom VJ-Übersetzer generierten Java-Quelltexte sollten von Entwicklern nicht bearbeitet werden.

Als Programmiersprache für die Implementierung des Compilers wurde Java gewählt.

Zur allgemeinen Information über die Entwicklung von Übersetzern wurden während der Entwicklung die folgenden Quellen herangezogen: [ASU86], [WM92] und [App98].

5.1 Übersetzung in Java-Quelltexte

In diesem Abschnitt wird erläutert, wie die Sprachmittel der Sprache VJ in Java-Quelltexte übersetzt werden.

Weil Java zur Definition benutzerdefinierter Typen nur Objektklassen zur Verfügung stellt, müssen Wertklassen bei der Übersetzung in Objektklassen umgewandelt werden. Die Implementierung der erzeugten Objektklassen wird so gestaltet, dass sich das Verhalten ihrer Exemplare zur Laufzeit den konzeptionellen Eigenschaften von Werten entspricht. Das bedeutet vor

allem, dass die Exemplare unveränderlich sind und dass bei Gleichheitsprüfungen nicht die Objektidentität sondern die Wertgleichheit geprüft wird.

Die Objektklassen, die als Übersetzung von Wertklassen erzeugt werden, werden im Folgenden als *Wertobjektklassen* bezeichnet.

5.1.1 Unveränderliche Objekte in Java

Damit Objekte einer Klasse unveränderlich sind, muss diese folgende Bedingungen erfüllen:

- Die Klasse darf nicht als Basisklasse für andere Klassen verwendet werden können. Ansonsten könnten in Subklassen verändernde Operationen hinzugefügt werden.

- Alle Exemplarvariablen der Klasse müssen als `final` deklariert sein. Dann ist sichergestellt, dass diese bei der Erzeugung des Exemplars initialisiert werden und anschließend nicht mehr verändert werden können.

- Keine der Exemplarvariablen darf ein veränderliches Objekt referenzieren, es sei denn, das Objekt wird defensiv kopiert und die Klasse selbst ruft an ihm keine verändernde Operation auf.

Die ersten beiden Punkte stellt der VJ-Übersetzer bei der Übersetzung einer Wertklasse in eine Objektklasse sicher, indem entsprechende `final`-Deklarationen eingefügt werden. Die dritte Forderung, keine veränderlichen Objekte zu referenzieren, ist bei Wertklassen ohnehin erfüllt, weil diese nur andere Werte referenzieren können.

5.1.2 Übersetzung einer Wertklasse

Eine Wertobjektklasse wird als `final` deklariert und erweitert die Basisklasse `vj.lang.Value` (siehe Beispiel 5.1).

Das Wertobjektklassen eine Basisklasse erweitern, überrascht zunächst, weil Vererbung für Wertklassen ausgeschlossen wurde (siehe Abschnitt 4.9). Die Ableitung aller Wertobjektklassen von einer gemeinsamen Basisklasse ist jedoch notwendig, um die Gleichheitsprüfung übersetzen zu können (siehe Abschnitt 5.1.6).

Beispiel 5.1 Übersetzung einer Wertklasse in eine Objektklasse

Die Wertklasse:

```
public valueclass Example {
}
```

Die vom Übersetzer erzeugte Objektklasse:

```
public final class Example extends vj.lang.Value {
}
```

5.1.3 Die Basisklasse `vj.lang.Value`

Die Basisklasse `vj.lang.Value` wird von allen Wertobjektklassen erweitert. Neben ihrer Funktion, Wertobjektklassen als solche zu kennzeichnen, überschreibt sie auch die Methoden aus der Klasse `Object`, die von Wertklassen nicht überschrieben werden können, und deklariert diese als `final`.

Der Quelltext der Klasse befindet sich im Anhang A.

5.1.4 Übersetzung der konstituierenden Eigenschaften

Die konstituierenden Eigenschaften einer Wertklasse werden in als `final` deklarierte Exemplarvariablen der Wertobjektklasse übersetzt. Außerdem wird ein Konstruktor erstellt, der für jede konstituierende Eigenschaft einen formalen Parameter entsprechenden Typs hat, die Reihenfolge der Parameter entspricht dabei der Quelltextreihenfolge der konstituierenden Eigenschaften.

Die konstituierenden Eigenschaften dienen auch als Basis für die automatisch erzeugte Implementierung der Methoden `equals` und `hashCode`. Die Implementierung von `hashCode` folgt dabei der in [Blo01] empfohlenen Implementierung.

Beispiel 5.2 zeigt die Übersetzung einer einfachen Wertklasse mit einer konstituierenden Eigenschaft in die zugehörige Wertobjektklasse.

5.1.5 Die Übersetzung von Wertwähler-Aufrufen

Weil bei der Übersetzung der konstituierenden Eigenschaften ein Konstruktor erzeugt wird, dessen Parameter den Parametern des Wertwählers der Wertklasse entsprechen, kann ein Wertwähler-Aufruf einfach in einen Aufruf dieses Konstruktors übersetzt werden. Syntaktisch wird im Prinzip nur

Beispiel 5.2 Übersetzung einer Wertklasse mit einer konstituierenden Eigenschaft

Die Wertklasse:

```
public valueclass Example {
    public int x;
}
```

Die vom Übersetzer erzeugte Objektklasse:

```
public final class Example extends vj.lang.Value {
    final public int x;

    public Example(int x) {
        this.x = x;
    }

    public boolean equals(Object o) {
        if (o instanceof Example) {
            return this.x == ((Example) o).x;
        } else {
            return false;
        }
    }

    public int hashCode() {
        int result = 17;
        result = 37*result + x;
        return result.
    }
}
```

das Schlüsselwort `const` durch das Schlüsselwort `new` ersetzt (siehe Beispiel 5.3).

Beispiel 5.3 Übersetzung eines Wertwähler-Aufrufs

Der Aufruf des Wertwählers in VJ:

```
Example e = const Example(42);
```

Das Ergebnis der Übersetzung in Java:

```
Example e = new Example(42);
```

5.1.6 Die Übersetzung des Gleichheitsoperators

Die Operatoren `==` und `!=` sind in VJ so definiert, dass sie ihre Operanden auf Wertgleichheit prüfen, wenn es sich diese Werttypen haben. Die Prüfung auf Wertgleichheit wird dabei in der Methode `equals` durchgeführt.

Wenn zur Übersetzungszeit anhand der statischen Typen der Operanden nicht festgestellt werden kann, ob es sich bei den Operanden zur Laufzeit um Objekte einer Objektklasse oder um Objekte einer Wertobjektklasse handeln wird, muss daher mit Hilfe des `instanceof`-Operators der Typ der Operanden zur Laufzeit geprüft werden. Dadurch, dass alle Wertobjektklassen Subtypen von `vj.lang.Value` sind, kann diese Klasse für die Prüfung verwendet werden. Beispiel 5.4 zeigt ein Beispiel für die Übersetzung einer Gleichheitsprüfung.

Sind die Operanden vom Typ `String`, so muss ebenfalls eine Wertgleichheitsprüfung durchgeführt werden, daher wird zur Laufzeit auch auf diesen Typ geprüft.

Die `instanceof`-Prüfung wird vom Übersetzer auch dann eingefügt, wenn bereits zur Übersetzungszeit aufgrund des statischen Typs sicher ist, dass es sich bei den Operanden um Werttypen handeln wird. Auch in diesem Fall müsste nämlich zur Laufzeit zunächst geprüft werden, ob nicht einer der beiden Operanden `null` ist. Ansonsten könnte es beim Aufruf der `equals`-Methode zu einer `NullPointerException` kommen. Diese Prüfung wird vom `instanceof`-Operator mit erledigt, dieser gibt `false` zurück, wenn der rechte Operand `null` ist.

Beispiel 5.4 Übersetzung des Gleichheitsoperators

Verwendung des Gleichheitsoperators in VJ:

```
public void test(Object o1, Object o2) {
    if (o1 == o2) {
        System.out.println("o1 == o2");
    }
}
```

Das Ergebnis der Übersetzung in Java: Zur besseren Lesbarkeit ist die Prüfung auf `java.lang.String` hier nicht enthalten.

```
public void test(Object o1, Object o2) {
    if (  (   (o1 instanceof vj.lang.Value)
           && (o2 instanceof vj.lang.Value))
        ? o1.equals(o2)
        : o1 == o2) {
        System.out.println("o1 == o2");
    }
}
```

5.2 Ablauf der Übersetzung

Die Übersetzung wird in den folgenden Schritten durchgeführt:

1. Lexikalische Analyse des Quelltextes.
2. Syntaktische Analyse und Erzeugung eines abstrakten Syntaxbaums.
3. Aufbereitung des abstrakten Syntaxbaums.
4. Aufbau einer Symboltabelle.
5. Typprüfung.
6. Umwandlung von VJ-Sprachkonstrukten in die entsprechende abstrakte Java-Syntax.
7. Generierung von Java-Quelltext.

5.2.1 Lexikalische und syntaktische Analyse

Die zwei ersten Schritte sind die lexikalische und die syntaktische Analyse des Quelltextes. In der lexikalischen Analyse wird der Quelltext in *Token* unterteilt, in der syntaktischen Analyse wird die Syntax mit Hilfe einer Grammatik der Programmiersprache analysiert.

Für die lexikalische und syntaktische Analyse wurde der Parsergenerator ANTLR[1] verwendet. Als Alternativen wurden JavaCC[2] und SableCC[3] in Betracht gezogen.

Für die Auswahl von ANTLR sprachen vor allem die Tatsache, dass für ANTLR eine Java-5-Grammatik [Jav04a] frei verfügbar ist, die bereits die Generierung eines AST beinhaltet, sowie die Unterstützung von Baumparsern und die umfangreiche, im Web frei verfügbare Dokumentation [Par05]. Ein Vergleich der drei Parsergeneratoren ist in Tabelle 5.1 zu finden.

Die Grammatik [Jav04a] wurde als Grundlage für die Entwicklung der VJ-Grammatik verwendet.

5.2.2 Baumgrammatiken

Das Ergebnis der syntaktischen Analyse ist ein abstrakter Syntaxbaum (AST). Zur Verarbeitung von abstrakten Syntaxbäumen stellt ANTLR sogenannte *Baumparser* zur Verfügung. Dabei handelt es sich um Parser, die eine Baumstruktur durchlaufen. In dem Parser können Aktionen angegeben werden, die während des Durchlaufens der Baumstruktur ausgeführt werden.

Alle auf die syntaktische Analyse folgenden Verarbeitungsschritte sind im VJ-Übersetzer mit Hilfe von Baumparsern implementiert.

5.2.3 Aufbereitung des abstrakten Syntaxbaums

Im dritten Schritt[4] wird der in der syntaktischen Analyse erstellte AST aufbereitet. Die ursprüngliche Struktur des erzeugten ASTs lässt sich teilweise nur umständlich weiter verarbeiten, weil für einige Teilbäume mit unterschiedlicher Bedeutung der gleiche Wurzelknotentyp verwendet wird. Die eigentliche Bedeutung ergibt sich dann erst, wenn auch die Kinder des Knotens betrachtet werden. Baumparser lassen sich jedoch einfacher formulieren, wenn Entscheidungen anhand des Wurzelknotens getroffen werden können.

[1] `http://www.antlr.org/`
[2] `https://javacc.dev.java.net/`
[3] `http://sablecc.org/`
[4] `src/java/de/uni_hamburg/informatik/vjcompiler/parser/ASTValidator.g`

	Vorteile	Nachteile
ANTLR	+ Java-5-Grammatik inklusive AST-Erzeugung verfügbar + Baumgrammatiken + ausführliche Dokumentation	− wenig Unterstützung für heterogene ASTs (ASTs mit unterschiedlichen Knoten-Typen) − unzureichende Unicode-Unterstützung
JavaCC	+ Klassen für heterogene ASTs können automatisch erstellt werden, inklusive automatischer Erzeugung eines Visitors + vollständige Unicode-Unterstützung	− keine Möglichkeit zur Prüfung der Struktur des erzeugten ASTs − veraltete Dokumentation
SableCC	+ sehr gut lesbare Syntax	− keine Java-5-Grammatik verfügbar − keine Unterstützung von Prädikaten oder Aktionen in der Grammatik − veraltete Dokumentation

Tabelle 5.1: Vergleich der Parsergeneratoren ANTLR, JavaCC und SableCC

Bei der Aufbereitung des Syntaxbaums werden daher die ohne Betrachtung der Kindknoten mehrdeutigen Knotentypen durch eigene, für die jeweiligen Bedeutungen definierte Typen ersetzt. Die in Beispiel 5.5 gezeigte Regel ersetzt zum Beispiel für Klassenliterale den Wurzelknotentyp `DOT` durch den Typ `CLASS_LITERAL`. Die Baumparser für die nachfolgenden Verarbeitungsschritte können dann entsprechend einfacher formuliert werden.

Beispiel 5.5 Aufbereitung des ASTs

```
classLiteral!
  : #(DOT
      type:typeSpecArray
      "class"
      endOfTree[#DOT]
    )
    {  ## = #([CLASS_LITERAL, "ClassLiteral"], type); }
  ;
```

5.2.4 Aufbau einer Symboltabelle

Der aufbereitete AST wird nun vom `DeclarationParser`[5] durchlaufen und es wird eine Symboltabelle aufgebaut. Dazu werden für die gefundenen Deklarationen Objekte erstellt, die die deklarierten Elemente repräsentieren, diese Objekte werden in die Symboltabelle eingefügt. Sie dienen auch gleichzeitig selbst als Symboltabelle für innerhalb der durch sie repräsentierten Deklaration deklarierte Elemente. Die Symboltabelle erhält dadurch eine hierarchische Struktur.

Typdeklarationen werden allerdings nicht direkt in die Pakete eingefügt, in denen sie deklariert sind, sondern in ein Objekt, dass ihre Übersetzungseinheit (compilation unit) repräsentiert. Die Übersetzungseinheit ist notwendig, um bei der Auflösung von Namen die Import-Anweisungen mit berücksichtigen zu können.

Um die Sichtbarkeitsbereiche des Programms repräsentieren zu können, werden nicht nur für Deklarationen, sondern auch für Blöcke, `for`-Anweisungen und Exception-Handler Objekte erstellt, die diese Repräsentieren und die als Symboltabelle für in diesen deklarierte lokale Variablen dienen.

Für im Quelltext verwendete Typen und Namen werden in diesem Schritt ebenfalls Objekte erstellt, die diese repräsentieren. Aufgelöst werden sie al-

[5] `src/java/de/uni_hamburg/informatik/vjcompiler/parser/DeclarationParser.g`

lerdings noch nicht. Die Auflösung ist in Java in diesem Schritt auch noch nicht möglich, weil Elemente in Java schon vor ihrer Deklaration benutzt werden können.

Die erstellten Objekte werden als Attribute an die Knoten des AST angehängt. Die Attribute können in späteren Schritten wieder ausgelesen werden.

Die Objekte zur Repräsentation der Deklarationen, Namen und Typen befinden sich in dem Paket `de.uni_hamburg.informatik.vjcompiler.model` und seinen Unterpaketen `declaration` und `type`. Die Aufteilung von Deklarationen und Typen ist notwendig, weil Java 5 – und damit auch VJ – generische Typen unterstützt. Bei einem generischen Typ existiert keine 1:1-Beziehung zwischen der Deklaration und den Typen, sondern die Deklaration definiert eine Menge von möglichen Typen. Daher kann zur Repräsentation von Deklaration und Typ nicht dasselbe Objekt verwendet werden.

Die Struktur des `model`-Pakets ähnelt der Struktur des Pakets `javax.lang.model` aus dem Java-API von Java 6 (vgl. [Sun06]). Auch dieses Paket dient der Repräsentation von Java-Programmen. Es deklariert allerdings nur Interfaces und bietet im Gegensatz zu den Klassen des VJ-Übersetzers auch keine Typen, die Sichtbarkeitsbereiche oder Übersetzungseinheiten repräsentieren. Diese sind innerhalb eines Übersetzers aber notwendig. Die Typen aus dem `javax`-Paket konnten daher im VJ-Übersetzer nicht verwendet werden. Nur die Struktur des Pakets wurde als Orientierungshilfe bei der Entwicklung herangezogen.

5.2.5 Typprüfung

Im folgenden Schritt werden die zuvor gefundenen Namen und Typen aufgelöst und die statische Typkorrektheit des zu übersetzenden Programms überprüft[6]. Die Typprüfung umfasst sowohl die auch in Java durchgeführten Prüfungen, also zum Beispiel die Prüfung, ob die Typen in einer Zuweisung typkompatibel sind, als auch die zur Sicherstellung der Korrektheit von Wertklassen notwendigen zusätzlichen Prüfungen.

Bei einer Wertklasse stellt die Typprüfung sicher, dass in ihrer Schnittstelle keine Objektklassen verwendet werden, dass alle konstituierenden Eigenschaften Werttypen haben und dass in der Implementierung der Methoden keine Objektklassen verwendet werden.

Implementiert wird die Typprüfung wie auch die anderen Schritte als ein Baumparser, der den AST durchläuft. Dabei werden die im vorherigen Schritt an die AST-Knoten geschriebenen Attribute ausgelesen, um Zugriff

[6]Dieser Schritt ist noch nicht implementiert, daher wird hier die geplante Implementierung beschrieben.

auf die Sichtbarkeitsbereiche und Symboltabellen zu erhalten. Diese werden dann verwendet, um die – ebenfalls aus den Attributen der AST-Knoten ausgelesenen – Namen und Typen aufzulösen.

Ergibt der Versuch, einen Namen oder Typ aufzulösen, dass dieser in einer anderen Datei deklariert ist, so wird diese externe Datei eingelesen. Dadurch, dass in Java und VJ der Name einer Datei mit dem Namen einer in dieser Datei deklarierten und von Code in anderen Übersetzungseinheiten genutzten Klasse übereinstimmen muss, kann die entsprechende Datei einfach gefunden werden. Handelt es sich um eine im Quelltext vorliegende Datei, werden die in ihr enthaltenen Deklarationen ebenso analysiert wie bei der aktuell in der Übersetzung befindlichen Datei, die zusätzlichen Informationen ergänzen die vorhandene Symboltabelle.

Handelt es sich dagegen um eine im Bytecode vorliegende Datei, so müssen aus dieser die relevanten Informationen ausgelesen werden. Dabei genügt es, die öffentliche Schnittstelle der in der Datei deklarierten Klasse zu analysieren, denn andere Elemente dieser Klasse können ohnehin nicht verwendet werden.

Um festzustellen, ob es sich bei einer im Bytecode vorliegenden Klasse um eine Wertobjektklasse handelt, kann wieder die Basisklasse `vj.lang.Value` verwendet werden. Ist diese Klasse die Basisklasse der im Bytecode vorliegenden Klasse, dann wurde letztere aus einer Wertklasse erstellt.

Es ist geplant, für das Einlesen von im Bytecode vorliegenden Klassen das Paket ASM[7] zu verwenden.

5.2.6 Umwandlung in Java-Syntax

Die Umwandlung[8] von VJ in Java erfolgt durch Umwandlung des AST in einen AST, der nur noch Elemente enthält, die auch die Sprache Java unterstützt. Der umgewandelte AST enthält also keine Wertklassendeklarationen und keine Wertwähler-Aufrufe mehr und die konstituierenden Eigenschaften in Wertklassen wurden in Exemplarvariablen umgewandelt.

Für die automatisch generierten Bestandteile einer Wertobjektklasse werden im Umwandler entsprechende AST-Teilbäume erstellt und in den AST eingefügt. Außerdem werden in diesem Schritt Ausdrücke, in denen mit dem `==`-Operator auf Gleichheit geprüft wird, in der in Abschnitt 5.1.6 beschriebenen Weise umgeformt.

[7] `http://asm.objectweb.org/`

[8] `src/java/de/uni_hamburg/informatik/vjcompiler/parser/VJToJavaTransformer.g`

5.2.7 Generierung von Java-Quelltext

Aus dem umgeformten AST wird schließlich im letzten Schritt[9] Java-Quelltext erzeugt. Der JavaEmitter basiert auf dem JavaEmitter von Parr [Jav04b] und wurde für Java 5 erweitert.

Um die richtige Auswertungsreihenfolge sicherzustellen, umschließt der JavaEmitter fast alle ausgegebenen Ausdrücke mit Klammern. Viele dieser Klammern sind überflüsseig. Sie schaden aber nicht, weil Klammern in Java ausschließlich die Reihenfolge der Auswertung bestimmen und ansonsten – mit einer Ausnahme, die aber kaum praxisrelevant sein sollte – keine Bedeutung haben (vgl. [GJSB05]).

5.3 Stand der Entwicklung

Der Stand der Entwicklung zum Zeitpunkt der Fertigstellung dieser Arbeit kann wie folgt zusammengefasst werden:

- Die lexikalische und syntaktische Analyse sind vollständig implementiert.
- Das gleiche gilt für die Aufbereitung des AST.
- Der Aufbau der Symboltabelle ist fast vollständig implementiert. Noch nicht berücksichtigt werden Typparameter und Annotationen.
- Die Typprüfung ist bisher nicht implementiert.
- Bei der Umwandlung in Java fehlt die Generierung der Methoden `equals` und `hashCode`. Diese können nach dem gleichen Prinzip wie der Konstruktor generiert werden, für die Generierung der `equals`-Methode sollten jedoch die Typinformationen verfügbar sein, weil je nach Typ einer konstituierenden Eigenschaft diese entweder mit dem `==`-Operator verglichen werden muss (wenn es sich um einen primitiven Typ oder einen Aufzählungstyp handelt) oder mit `equals` (wenn es sich um eine Wertklasse handelt).
- Die Generierung des Java-Quelltextes ist vollständig implementiert, allerdings ist die Implementierung in der vorliegenden Form relativ unübersichtlich und somit nur schwer wartbar.

[9] `src/java/de/uni_hamburg/informatik/vjcompiler/parser/JavaEmitter.g`

Allgemein wurden während der Entwicklung bisher nur wenig Fehlerfälle getestet. Es ist daher davon auszugehen, dass der Übersetzer sich nicht immer wie erwartet verhalten wird, wenn als Eingabe fehlerhafte Quelltexte verwendet werden.

Priorität bei einer Weiterentwicklung sollte die Implementierung der Typprüfung haben, und hier insbesondere die Kontrolle, dass in Wertklassen die für diese geltenden Einschränkungen eingehalten sind. Ohne diese Prüfung kann der Compiler nicht verhindern, dass Wertklassen implementiert werden, deren Verhalten nicht den konzeptionellen Eigenschaften von Werten entspricht.

Kapitel 6

Zusammenfassung

In dieser Arbeit wurden zunächst die konzeptionellen Unterschiede zwischen Werten und Objekten untersucht und dargestellt. Die Untersuchung einer Reihe von vorhandenen objektorientierten Programmiersprachen hat gezeigt, dass Werte als Konzept in diesen Sprachen nicht ausreichend unterstützt werden.

Daher wurde in dieser Arbeit ein Konzept für die Modellierung von Wertklassen entwickelt, dass als Basis für den Entwurf von objektorientierten Sprachen, die sowohl Werte als auch Objekte unterstützen, dienen kann.

Mit VJ wurd eine solche Sprache entworfen. Die Sprache vereinfacht die Entwicklung von Wertklassen, weil Entwickler nicht mehr gezwungen sind, die dabei zu beachtenden Konventionen und Einschränkungen zu kennen. Die Sprache garantiert, dass Exemplare eines als Wertklasse deklarierten Typs entsprechend den konzeptionellen Eigenschaften von Werten verhalten.

Außerdem wurde für VJ ein Übersetzer entwickelt, der VJ-Programme in Java-Quelltexte übersetzt.

Ansatzpunkte für weitere Arbeiten liefert zunächst der Übersetzer, dessen Entwicklung abgeschlossen werden sollte. Um einen komfortablen Umgang mit der Sprache zu ermöglichen, wäre außerdem eine Einbindung in moderne Entwicklungsumgebungen wie Eclipse[1] wünschenswert.

Konzeptionell wurde bislang nicht geklärt, welche alternativen Möglichkeiten zur Modellierung von Werten neben dem in dieser Arbeit verwendeten Konzept der konstituierenden Eigenschaften bestehen. Das Konzept der konstituierenden Eigenschaften eignet sich nur schlecht für die Modellierung von Werttypen, die nicht aus einer festen Anzahl Bestandteile zusammengesetzt sind, zum Beispiel Mengen und Listen.

[1] `http://www.eclipse.org/`

Anhang A

Die Klasse `vj.lang.Value`

```
package vj.lang;

/**
 * Base class for all value classes.
 */
public abstract class Value {

    /**
     * Throws a @link{java.lang.CloneNotSuppotedExeption}.
     * @throws CloneNotSupportedException always thrown.
     */
    @Override
    final protected Object clone()
    throws CloneNotSupportedException {
        // The VJ langauge specification requires throwing
        // a CloneNotSupportedException.
        throw new CloneNotSupportedException();
    }

    /**
     * Does nothing.
     */
    @Override
    final protected void finalize() {
        // Do nothing, as defined in the VJ language
        // specification.
```

```
    }

    /**
     * Returns the empty string.
     * @return the empty string.
     */
    @Override
    public String toString() {
        // Returns the empty string. This implementation
        // ensures that the same string will be returned
        // by multiple representations of the same value.
        // The method is not final, so it can be
        // overriden by value classes.
        return "";
    }
}
```

Anhang B

Änderungen an der Java-Grammatik

Dieser Anhang zeigt die wesentlichen Änderungen, die an der Java-Grammatik [Jav04a] durchgeführt wurden, um die VJ-Grammatik zu erstellen. Die vollständige VJ-Grammatik befindet sich in der Datei `src/java/de/uni_hamburg/informatik/vjcompiler/parser/vj.g` der VJ-Übersetzer-Quelltexte.

Im `tokens`-Abschnitt der Grammatik wurden die folgenden Token zusätzlich definiert:

```
CONST="const";
VALUECLASS="valueclass";
VALUECLASS_DEF;
COMPILATION_UNIT;
SUPER="super";
QUALIFIED_SUPER;
FIELD_ACCESS;
NAME;
IMPORT_ON_DEMAND;
STATIC_IMPORT_ON_DEMAND;
CLASS_LITERAL;
QUALIFIED_NEW;
QUALIFIED_THIS;
INVOCATION_ON_EXPRESSION;
EXPRESSION_NAME;
```

In der Regel `compilationUnit` wird der erhält der erzeugte AST einen

Wurzelknoten vom Typ `COMPILATION_UNIT`.

Der Regel `typeDefinitionInternal` wurde eine Alternative hinzugefügt, die die Deklaration von Wertklassen erlaubt:

```
// Protected type definitions production for reuse in other productions
protected typeDefinitionInternal[AST mods]
: classDefinition[#mods] // inner class
| interfaceDefinition[#mods] // inner interface
| enumDefinition[#mods] // inner enum
| annotationDefinition[#mods] // inner annotation

| valueclassDefinition[#mods]  // added for VJ
;
```

Die Regel `valueclassDefinition` wurde erstellt:

```
// valueclass declarations, based on classDefinition rule from Java grammar
valueclassDefinition![AST modifiers]
: "valueclass" IDENT
// it _might_ have type paramaters
(tp:typeParameters)?
// it might implement some interfaces...
ic:implementsClause
// now parse the body of the class
cb:valueclassBlock
{#valueclassDefinition = #(#[VALUECLASS_DEF,"VALUECLASS_DEF"],
modifiers,IDENT,tp,ic,cb);}
;
```

Die Regel `valueclassBlock` wurde erstellt:

```
// valueclass body, based on "classBlock" rule from Java grammar
valueclassBlock
: LCURLY!
( valueclassField | SEMI! )*
RCURLY!
{#valueclassBlock = #([OBJBLOCK, "OBJBLOCK"], #valueclassBlock);}
;
```

Die Regel `valueClassField` wurde erstellt. Die Regel ist fast identisch mit der Regel `classField` für Klassen, lediglich die Deklaration von Exemplarinitialisierern wird nicht erlaubt. Die Deklaration von Konstruktoren

wird von dieser Regel aber akzeptiert. Konstruktoren in einer Wertklasse werden in der nachfolgenden Verarbeitung des AST von den Baumparsern nicht akzeptiert, sodass die Deklaration von Konstruktoren dennoch zu einem Fehler führen wird.

```
// valueclass members, based on "classField" rule from Java grammar
valueclassField!
: // method, constructor, or variable declaration
mods:modifiers
( td:typeDefinitionInternal[#mods]
{#valueclassField = #td;}

| (tp:typeParameters)?
// A generic method/ctor has the typeParameters before the return type.
// This is not allowed for variable definitions, but this production
// allows it, a semantic check could be used if you wanted.
t:typeSpec[false] // method or variable declaration(s)
( IDENT // the name of the method

// parse the formal parameter declarations.
LPAREN! param:parameterDeclarationList RPAREN!

rt:declaratorBrackets[#t]

// get the list of exceptions that this method is
// declared to throw
(tc:throwsClause)?

( s2:compoundStatement | SEMI )
{#valueclassField = #(#[METHOD_DEF,"METHOD_DEF"],
 mods,
 tp,
 #(#[TYPE,"TYPE"],rt),
 IDENT,
 param,
 tc,
 s2);}
| v:variableDefinitions[#mods,#t] SEMI
{#valueclassField = #v;}
)
```

```
)

// "static { ... }" class initializer
| "static" s3:compoundStatement
{#valueclassField = #(#[STATIC_INIT,"STATIC_INIT"], s3);}
;
```

Anhang C

VJ-Sprachspezifikation

C.1 Einführung

Die Programmiersprache VJ basiert auf der Sprache Java und erweitert diese um einen zusätzlichen Typkonstruktor, mit dem sich *Wertklassen* deklarieren lassen.

C.2 Werttypen

Werttypen im Sinne dieser Spezifikation sind Wertklassen (C.3)[1], Aufzählungstypen (§ 8.9), die Klasse `java.lang.String` und die primitiven Typen (§ 4.2).

C.3 Wertklassen

Eine Wertklassendeklaration definiert einen neuen, benannten Referenztyp. Wertklassen können überall dort deklariert werden, wo Objektklassen (§ 8.1) deklariert werden können:

```
<ClassDeclaration> ::=
        <NormalClassDeclaration>
      | <EnumDeclaration>
      | <ValueClassDeclaration>
```

[1] Querverweise in dieser Spezifikation haben die Form (C.1.2), wenn sie auf Abschnitte innerhalb dieser Spezifikation verweisen, und die Form (§ 1.2), wenn sie auf die Java-Sprachspezifikation [GJSB05] verweisen.

```
<ValueClassDeclaration> ::=
        <ValueClassModifier>* 'valueclass' <Identifier>
        <TypeParameters>? <Interfaces>? <ValueClassBody>

<ValueClassModifier> ::=
        <Annotation>
      | 'public'
      | 'private'
      | 'static'
      | 'strictfp'
```

Wertklassen können keine anderen Klassen erweitern (`extends`) und können nicht als Basisklassen für andere Klassen verwendet werden. Wertklassen können jedoch Interfaces implementieren, sodass ihre Exemplare teilweise polymorph verwendet werden können (siehe Abschnitt C.3.4). Für Superinterfaces gelten dabei dieselben Regeln wie für Superinterfaces von Objektklassen in Java (§ 8.1.5), mit der zusätzlichen Einschränkung, dass es sich bei allen Superinterfaces einer Wertklasse um *Wertinterfaces* (C.4) handeln muss.

C.3.1 Klassenrumpf und Member-Deklarationen

Im Rumpf der Wertklasse werden deren Member deklariert. Mögliche Member einer Wertklasse sind *konstituierende Eigenschaften* (C.3.2), Methoden (C.3.4) und Wertklassen (C.3.6).

Objektklassen (§ 8.1) oder Interfaces (§ 9) können keine Member einer Wertklasse sein. Ebenfalls unzulässig sind Exemplarinitialisierer (§ 8.6) sowie die Deklaration von Konstruktoren (§ 8.8).

```
<ValueClassBody> ::=
        '{' <ValueClassBodyDeclaration>* '}'

<ValueClassBodyDeclaration> ::=
        <ValueClassMemberDeclaration>

<ValueClassMemberDeclaration> ::=
        <ConstituentDeclaration>
      | <MethodDeclaration>
      | <ValueClassDeclaration>
      | ';'
```

C.3.2 Konstituierende Eigenschaften

Die konstituierenden Eigenschaften einer Wertklasse definieren die Gleichheit und damit auch Identität für Exemplare dieser Klasse. Zwei Werte sind identisch, wenn ihre konstituierende Eigenschaften identisch sind.

Die konstituierenden Eigenschaften einer Klasse werden wie folgt deklariert:

```
<ConstituentDeclaration> ::=
        <ConstModifier>* <Type> <ConstDeclarators> ';'

<ConstModifier> ::=
        <Annotation>
      | 'public'
      | 'private'

<ConstDeclarators> ::=
        <Identifier>
      | <ConstDeclarators> ',' <Identifier>
```

Konstituierende Eigenschaften sind unveränderlich. In der Deklaration einer konstituierenden Eigenschaft kann kein Initialisierer angegeben werden, weil sonst der Wert für alle Exemplare des Typs gleich wäre.

Beim Typ einer konstituierenden Eigenschaft muss es sich um einen Werttyp (C.2) handeln, ansonsten schlägt die Übersetzung fehl.

Die Sichtbarkeit konstituierender Eigenschaften wird ähnlich festgelegt wie die Sichtbarkeit der Felder von Objektklassen (§ 6.6). Wird in der Deklaration nicht das Schlüsselwort `private` angegeben, dann kann die konstituierende Eigenschaft aus anderen Klassen heraus gelesen werden; entweder nur von im selben Package deklarierten Klassen, oder, wenn das Schlüsselwort `public` angegeben wird, von allen Klassen. Ein verändernder Zugriff ist unabhängig von der Sichtbarkeit nicht möglich.

C.3.3 Wertwähler

Jede Wertklasse besitzt implizit einen *Wertwähler* (*Picker*). Der Wertwähler hat die gleiche Sichtbarkeit wie die Klasse. Er hat für jede in der Klasse deklarierte konstituierende Eigenschaft unabhängig von deren Sichtbarkeit

genau einen formalen Parameter, dessen Typ der Typ der zugehörigen konstituierenden Eigenschaft ist. Die Reihenfolge der formalen Parameter entspricht der Reihenfolge, in der die konstituierenden Eigenschaften in der Klasse deklariert sind.

Beispiel

Die folgende einfache Wertklasse besitzt einen Wertwähler mit zwei Parametern, dabei hat der erste Parameter den Typ `int` und der zweite den Typ `Currency`. Die Sichtbarkeit des Wertwählers ist `public`.

```
public valueclass Money {
    private int amount;
    public Currency currency;
}
```

C.3.4 Methodendeklarationen

In Wertklassen können nur Methoden deklariert werden, die einen Wert zurückgeben. Beim Rückgabetyp muss es sich um einen Werttyp (C.2) oder ein Wertinterface (C.4) handeln. Falls eine Methode formale Parameter hat, so muss der Typ jedes ihrer formalen Parameter ein Werttyp sein.

Methoden einer Wertklasse dürfen in ihrer throws-Klausel keine geprüften Exceptions deklarieren.

Um die referentielle Transparenz bei der Verwendung von Werten sicherzustellen, dürfen in der Implementierung von Methoden keine Objekte erzeugt werden und keine Klassenmethoden von Objektklassen aufgerufen werden, mit Ausnahme des Aufrufs von Klassenmethoden der Klasse `java.lang.Math`.

Polymorphe Verwendung von Werten

Werte können polymorph verwendet werden, wenn sie einer Variablen vom Typ `java.lang.Object` zugewiesen werden oder wenn sie Wertinterfaces (C.4) implementieren und einer Variablen vom Typ eines implementierten Interfaces zugewiesen werden. Die Methoden `clone()`, `equals(Object)`, `finalize()` und `hashCode()` können jedoch nicht redefiniert werden. Die Redefinition von `toString()` ist zulässig.

Der Aufruf von `clone()` wirft eine `CloneNotSupportedException`.

Die Methoden `equals(Object)` und `hashCode()` werden von Wertklassen automatisch so implementiert, dass `equals` die Gleichheit der Werte

prüft, das heißt der Aufruf von `equals` gibt genau dann `true` zurück, wenn ein Wert übergeben wird und die konstituierenden Eigenschaften des übergebenen Werts gleich denen des Werts sind, an dem der Aufruf erfolgt. Die Methode `hashCode` verhält sich so, dass sie den Vertrag von `hashCode` erfüllt.

Der Aufruf von `finalize()` hat keine Wirkung.

Der Rückgabewert von `toString()` ist eine undefinierte Zeichenkette, wenn die Methode nicht vom Entwickler der Klasse redifiniert wurde.

C.3.5 Beschränkung der Wertmenge (Gültigkeitsprüfung)

Soll für eine Wertklasse nicht jede mögliche Kombination konstituierender Eigenschaften einen gültigen Wert ergeben, so kann die Wertmenge durch eine Gültigkeitsprüfung entsprechend beschränkt werden. Der Entwickler einer Klasse kann dazu eine Klassenmethode mit dem Namen `isValid` definieren. Als formale Parameter muss diese Methode für jede konstituierende Eigenschaft der Klasse genau einen Parameter vom Typ der konstituierenden Eigenschaft besitzen, in der Quelltextreihenfolge der konstituierenden Eigenschaften. Die Methode muss den Rückgabetyp `boolean` haben und genau dann `true` zurückgeben, wenn die übergebenen Werte eine gültige Kombination konstituierender Eigenschaften sind, der durch diese beschriebene Wert also existiert.

Wird die Methode vom Entwickler nicht definiert, so wird sie automatisch zur Verfügung gestellt. Die automatisch erstellte Methode gibt bei jedem Aufruf `true` zurück.

Es ist eine Vorbedingung für den Aufruf eines Wertwählers, dass der Wert mit den als Parameter angegebenen konstituierenden Eigenschaften existiert. Die Einhaltung dieser Vorbedigung wird bei jedem Aufruf eines Wertwählers automatisch durch Aufruf von `isValid` geprüft. Gibt die Methode `false` zurück, existiert der Wert also nicht, schlägt der Aufruf des Wertwählers mit einer `NoSuchValueException` fehl.

Beispiel

Die folgende Wertklasse definiert die Menge positiver Ganzzahlen basierend auf der Menge aller Ganzzahlen:

```
public valueclass PositiveInteger {
    public int value;

    public static boolean isValid(int value) {
```

```
        return value >= 0;
    }
}
```

Der Versuch, einen negativen Wert auszuwählen, würde zur Laufzeit fehlschlagen:

```
PositiveInteger i1 =
    const PositiveInteger(42);  // ok
PositiveInteger i2 =
    const PositiveInteger(-1);  // throws NoSuchValueException
```

C.3.6 Geschachtelte Typdeklarationen

Innerhalb einer Wertklasse können weitere Wertklassen als geschachtelte Klassen deklariert werden.

C.4 Wertinterfaces

Damit eine Wertklasse ein Interface (§ 9) implementieren kann, darf das Interface keine Methode enthalten, die nicht auch in einer Wertklasse deklariert werden dürften.

Ein Interface ist ein *Wertinterface*, wenn für jede der in dem Interface enthaltenen Methoden gilt, dass diese einen Rückgabetyp ungleich `void` hat, dieser Rückgabetyp ein Werttyp (C.2) oder ein Wertinterface ist, der Typ jedes formalen Parameters der Methode ein Werttyp ist und die throws-Klausel der Methode keine geprüften Exceptions deklariert. Diese Einschränkungen gelten nicht für Methoden, die mit der gleichen Signatur und dem gleichen Rückgabetyp in `Object` deklariert sind.

C.5 Ausdrücke

C.5.1 Wertwähler-Aufruf

Durch den Aufruf des Wertwählers (C.3.3) wird ein Wert eines Werttyps ausgewählt. Ein Wertwähler-Aufruf ist eine `<PrimaryExpression>` (§ 15.8). Das Ergebnis des Ausdrucks ist der ausgewählte Wert.

```
<ValueSelectionExpression> ::=
        'const' <ValueClassType> '(' <ArgumentList>? ')'
```

Beispiel

Ein Wert des Typs `Money` aus Abschnitt C.3.3 ann wie folgt ausgewählt werden:

```
Money m = const(100, Currency.EUR);
```

C.5.2 Referenzvergleichsoperatoren == und !=

Um die Gleichheit bzw. Identität von Werten mit Hilfe des ==-Operators und des !=-Operators (§ 15.21) prüfen zu können, muss dessen Implementierung gegenüber gewöhnlichem Java angepasst werden.

Wenn der Typ beider Operanden ein gewöhnlicher Objekttyp ungleich `Object` ist, ändert sich die Implementierung der Operatoren nicht.

Ist der Typ mindestens eines Operanden `Object` oder ist der Typ beider Operanden ein Wertinterface (C.4) oder ist der Typ eines Operanden ein Wertinterface und der des anderen eine Wertklasse (C.3), dann wird die Vergleichsoperation so übersetzt, dass zur Laufzeit geprüft wird, ob der Typ beider Operanden eine Wertklasse ist, und der Vergleich wird durch Aufruf der `equals`-Methode dieser Wertklasse durchgeführt, falls dies der Fall ist.

Ist der Typ beider Operanden eine Wertklasse, so wird die Vergleichsoperation in einen Aufruf der `equals`-Methode dieser Wertklasse übersetzt.

C.6 Die Klasse `java.lang.String`

Die Klasse `String` gilt in VJ als Werttyp (C.2). Dies gilt auch für den ==-Operator, d. h. das folgende Programm gibt „`true`“ aus:

```
public class Test {
    public static void main(String[] args) {
        String s1 = const String("foo");
        String s2 = const String("foo");
        System.out.println(s1 == s2);
    }
}
```

Werte vom Typ `String` können wie im obigen Beispiel mit dem Schlüsselwort `const` ausgewählt werden. Aus Kompatibilitätsgründen ist jedoch auch die Verwendung von `new` möglich.

Anhang D

Inhalt der CD

- `compiler/` Ausführbare Version des VJ-Übersetzers.
- `compiler-src/` Quelltexte des VJ-Übersetzers.
  - `examples/` Beispiel-VJ-Quelltexte.
  - `lib/` Verwendete Bibliotheken.
  - `src/` Die eigentlichen Quelltexte.
  - `tests/` VJ-Quelltexte, die bestimmte Funktionen des Übersetzers testen; die Quelltexte sind nicht unbedingt gültige Quelltexte.
- `quellen/` In dieser Arbeit verwendete elektronische Quellen.

Der VJ-Übersetzer befindet sich in einer ausführbaren Jar-Datei und wird aufgerufen mit Angabe der zu übersetzenden Datei als Kommandozeilenparameter:

`java -jar vjc.jar` *quelltext.vj*

Die Übersetzung wird auf der Konsole ausgegeben. Der Übersetzer verwendet Log4J zum Logging, das Logging kann über die Datei `log4j.properties` konfiguriert werden.

Literaturverzeichnis

[App98] Appel, Andrew W.: *Modern Compiler Implementation in Java*. Cambridge : Cambridge University Press, 1998

[ASU86] Aho, Alfred V. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Priciples, Techniques and Tools*. Reading, Massachusetts : Addison-Wesley Publishing Company, 1986

[Blo01] Bloch, Joshua: *Effective Java*. Boston, Massachusetts : Addison-Wesley, 2001

[BRS+98] Bäumer, Dirk ; Riehle, Dirk ; Siberski, Wolf ; Lilienthal, Carola ; Megert, Daniel ; Sylla, Karl-Heiz ; Züllighoven, Heinz: Values in Object Systems / Ubilab. UBS AG, Zurich, Switzerland, 1998 (98.10.1). – Forschungsbericht

[BSK+04] Breitling, Holger ; Schmolitzky, Axel ; Koch, Jörn ; Lippert, Martin ; Roock, Stefan ; Ritterbach, Beate: *Towards a Language Support of User-defined Value Types in Object-Oriented Programming Languages*. 2004. – unveröffentlichtes Manuskript

[Ecm05] Ecma International: *Standard ECMA-334 – C# Language Specification*. 3rd. June 2005. – Veröffentlicht im Internet unter `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`. Letzter Abruf: 24.03.2006

[Fol97] Howe, Denis (Hrsg.). *Free On-line Dictionary of Computing, Referential Transparency*. Veröffentlicht im Internet unter `http://foldoc.org/foldoc.cgi?referential+transparency`. Letzter Abruf 09.08.2006. 1997

[FS02] Fürter, Martin ; Spreckelmeyer, Annika: *Konzepte und Ansätze zur Unterstützung der Implementierung fachlicher Werte in*

objektorientierten Programmiersprachen am Beispiel des Java-Rahmenwerks JWAM, Universität Hamburg, Fachbereich Informatik, Diplomarbeit, April 2002

[GJSB05] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *The Java™Language Specification*. 3rd. Upper Saddle River, New Jersey : Addison-Wesley, 2005

[GR83] Goldberg, Adele ; Robson, David: *Smalltalk-80: The Language and its Implementation*. Reading, Massachusetts : Addison-Wesley, 1983

[GS00] Grogono, Peter ; Sakkinen, Markku: Copying and Comparing: Problems and Solutions. In: *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming* Bd. 1850. London, UK : Springer-Verlag, 2000. – ISBN 3–540–67660–0, S. 226–250

[Hei05] Heiden, Markus. *Generierung von Fachwerten aus einem abstrakten Sprachmodell*. Studienarbeit, Universität Hamburg, Fachbereich Informatik. Juli 2005

[Jav04a] Studman, Michael (Hrsg.). *Java 5 (aka 1.5) grammar*. Veröffentlicht im Internet unter `http://www.antlr.org/grammar/1090713067533/index.html`. Letzter Abruf: 26.01.2006. 2004

[Jav04b] Parr, Terence (Hrsg.). *Java Emitter*. Veröffentlicht im Internet unter `http://www.antlr.org/share/1101058268251/java.tree.g`. Letzter Abruf: 12.05.2006. 2004

[KR99] Kölling, Michael ; Rosenberg, John: On Creation, Equality and the Object Model. In: *Technology of Object-Oriented Languages and Systems (TOOLS) 32*. Melbourne, Australia : IEEE, January 1999, S. 210–221

[Mac82] MacLennan, B. J.: Values and objects in programming languages. In: *SIGPLAN Notices* 17 (1982), Nr. 12, S. 70–79. – ISSN 0362–1340

[Mey97] Meyer, Bertrand: *Object-Oriented Software Construction*. 2nd. Upper Saddle River, New Jersey : Prentice Hall, 1997

[Mül99] Müller, Klaus. *Konzeption und Umsetzung eines Fachwertkonzepts*. Studienarbeit, Universtität Hamburg, Fachbereich Informatik. Juli 1999

[Mös03] Mössenböck, Hanspeter: *Softwareentwicklung mit C#*. Heidelberg : dpunkt.verlag, 2003

[Par05] Parr, Terence: *ANTLR Reference Manual*. University of San Franciso, Januar 2005. – Veröffentlicht im Internet unter `http://www.antlr.org/doc/index.html` sowie als Teil der ANTLR-Distribution. Letzter Abruf: 07.08.2006.

[Rit03] Ritterbach, Beate: Eigene Werttypen in Java. In: *JavaSpektrum* 4 (2003), S. 46–50

[Sau01] Sauer, Joachim. *Konfiguration von Fachwerten mit XML-Definitionen*. Studienarbeit, Universität Hamburg, Fachbereich Informatik. Juli 2001

[Seb02] Sebesta, Robert W.: *Concepts of Programming Languagaes*. 5. Auflage. Boston, Massachusetts : Addison-Wesley, 2002

[Sun04] Sun Microsystems, Inc.: *JDK™5.0 Documentation*. 2004. – Veröffentlicht im Internet unter `http://java.sun.com/j2se/1.5.0/docs/index.html`. Letzter Abruf: 05.08.2006.

[Sun06] Sun Microsystems, Inc.: *JDK™6 Documentation*. 2006. – Dokumenation zur Betaversion beta2-b79. Veröffentlicht im Internet unter `https://mustang.dev.java.net/`. Letzter Abruf: 09.04.2006.

[Wer98] van der Werf, Peter: Values and Objects Revisited. In: *JOOP* 11 (1998), Nr. 4, S. 25–34

[WM92] Wilhelm, Reinhard ; Maurer, Dieter: *Übersetzerbau: Theorie, Konstruktion, Generierung*. Berlin : Springer-Verlag, 1992

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Alle Quellen, die aus dem World Wide Web entnommen oder in einer sonstigen digitalen Form verwendet wurden, sind der Arbeit beigefügt.

Hamburg, den 11. August 2006
Jörg Rathlev