FH Wedel – University of Applied Sciences
Technische Informatik

Software Praktikum
Pflichtenheft

Eine virtuelle Machine für eine Spezialsprache zur Bildmanipulation

Peter Blicharski, II3839
Malte Hopmann, II4271

12. November 2002

Peter Blicharski, II3839
Malte Hopmann, II4271

Inhaltsverzeichnis

Peter Blicharski, II3839
Malte Hopmann, II4271

1 Allgemeine Problemstellung

1.1 Aufgabenstellung

Die an der Fachhochschule Wedel in der Vorlesung Compilerbau[1] entwickelte Spezialsprache[2] PPL[1] zur Erzeugung und Manipulation von Bildern soll effizienter und maschinenennaher Verwirklicht werden. Zu diesem Zweck soll die vorhandene Virtuelle Maschine, die in Haskell[3] geschrieben wurde, in Java umkonzipiert und umprogrammiert werden.

> *Die Aufgabe kann in zwei Teile gegliedert werden: Die Ablaufsteuerung innerhalb der virtuellen Maschine und die Bildbearbeitungsoperationen. Die Haskell-Implementierung dient für diese Aufgabe als Spezifikation und Prototyp.*

1.2 Entwicklung der Virtuellen Maschine

1.2.1 Vorgaben und Umgebung

Das Programm soll in **Java 2** implementiert werden. Ein Compiler–Backend, welches PPL nach Java übersetzt ist vorhanden. Die Quellen des PPL-Prototyp Compilers, der PPL-VM als Interpretierer und einiger Beispiel als tar.gz[4] Archiv dienen zur Spezifikation und als Einstiegshilfe.

1.2.2 Zeitplan

Ein selbst erstellter Zeitplan liegt der Dokumentation als PDF–Datei bei.

[1] Picture Programming Language

2 Grobentwurf

2.1 Leitfaden für den Grobentwurf

Um eine leichte Erweiterbarkeit des Entwurfs zu gewährleisten, haben wir uns für vier Punkte entschieden, anhand derer die entsprechenden Muster designed und entwickelt werden sollen. Nach Prioritäten gewählt wären dass:

1. Ladbare Bildtypen erweitern
2. Grau– auf Farbbilder umstellen
3. Bildbearbeitungsbefehle hinzufügen
4. Datentypen erweitern

In den jeweiligen Abschintten wird auf die Realisation dieser Punkte expliziter eingegangen.

2.2 Executable

Als Einstieg haben wir das Compilat gewählt, welches aus dem Backend–Compiler erstellt wird:

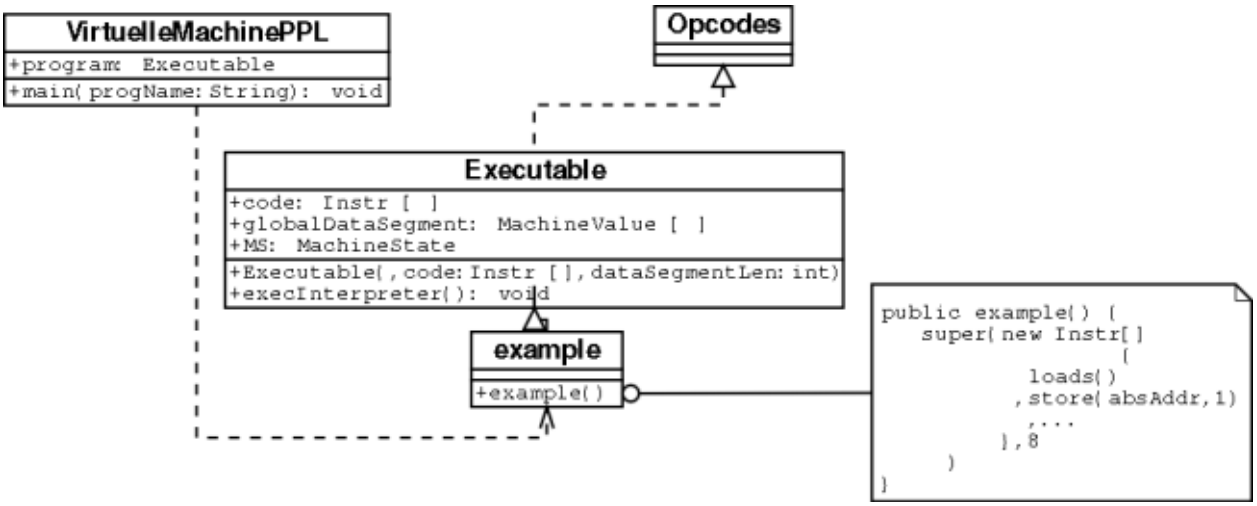
```
package ppl;
import ppl.Executable;

public class example extends Executable {
    public example() {
        super(new Instr []
              { loads("lena.pgm")
              , store(absAddr,1)
              , loads("hello world")
              , svc(writeln)
              , loads("good bye")
              , svc(writeln)
              , undef()
              , store(absAddr,1)
              , compute(terminate)
              }
             , 8
             );
        }
    }
```

Es wird ein Konstruktor implementiert, der eine Instanz der Executable erstellt mit dem auszuführenden Befehlen als Parameter.
Es entsteht folgendes OMT–Diagramm:

Jediglich der Name des Programms muß an die PPL–Virtuelle Maschine übergeben werden. Unter Verwendung des „Dynamic Loading“, kann zur Laufzeit eine Instanz des Programms erstellt werden,

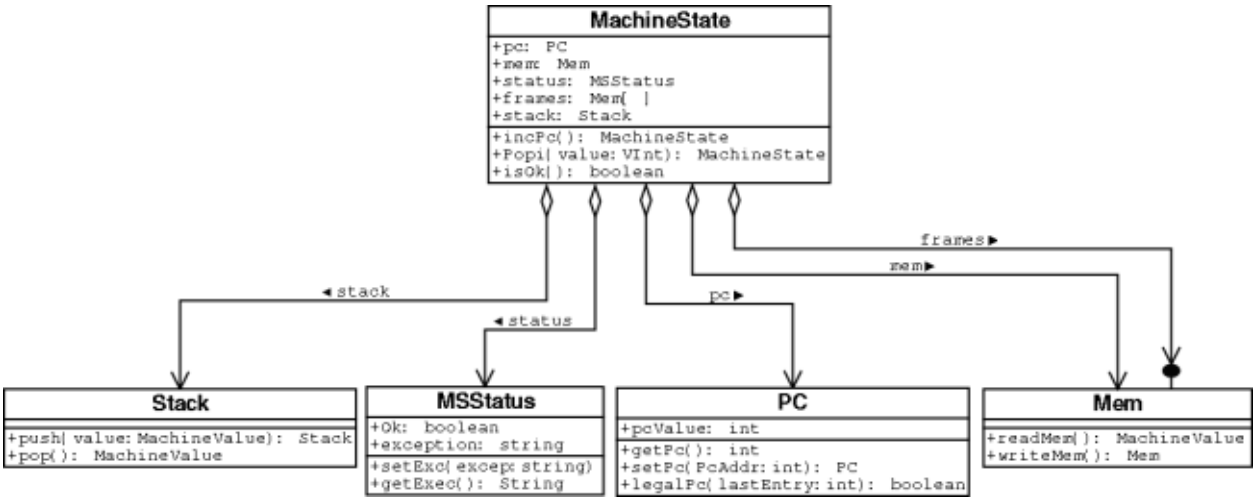
Abbildung 2.1:
OMT: Virtuelle Maschine Applikation



das von der Executable erbt und die Verarbeitung der Befehle übernimmt. Anhand der Liste an Befehlen[1] kann dann entsprechend dem Programmablauf, der jeweils zu bearbeitende Befehl dynamisch zur Laufzeit über die Methode *exec()* der jeweiligen Instanz des Befehls ausgeführt werden. Über die Eigenschaft *MS* vom Typ *MachineState* wird der jeweilige Zustand der Verarbeitung gesichert.

[1] Variable: **code**

Abbildung 2.2:
OMT: MachineState



2.3 MachineState

Die Darstellung der Zustände, in denen sich die Virtuelle Maschine zur Zeit der Verarbeitung befindet, wird in dieser Klasse gespeichert. Die Referenzen, die diese Klasse besitzt, stellen alle wichtigen Eigenschaften zur Verfügung (siehe Abb. 2.2) um die Befehle zu Bearbeiten. Implementiert sind:

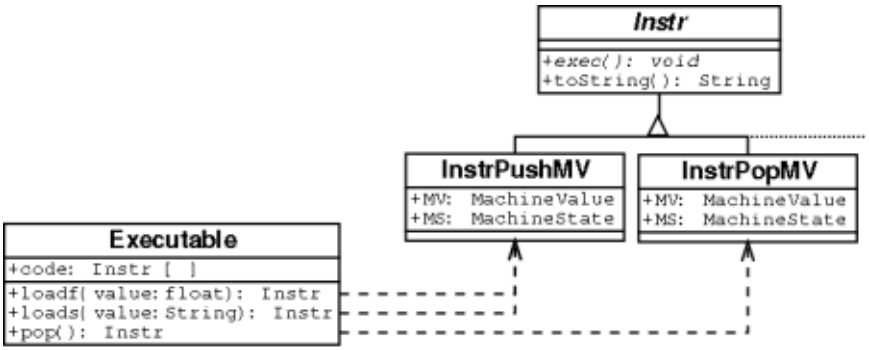
- Programm Counter
- (typed) Stack
- Statusregister
- Heap (mehrere)

Zu dem wird hier die Typüberprüfung statt finden. Eine erste Idee ist, für alle Datentypen einen geeigneten *Pop*–Befehl zu entwickeln, der gleich das Datum auf den richtigen Typen überprüft.

2.4 Instructions

Die grundlegenden Befehle, wie z.B. *pop*, *loadX* und *store* stehen in der Executable.

Abbildung 2.3:
OMT: Instr

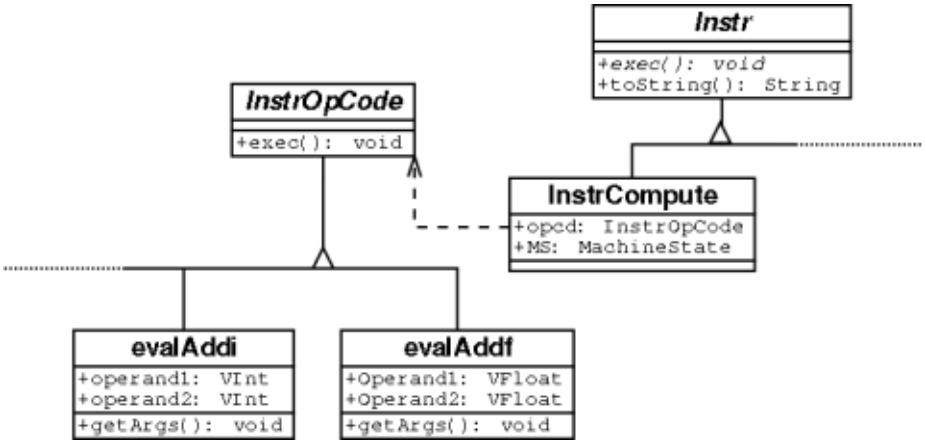


Durch sie werden die entsprechenden Klassen aufgerufen, die eine Instanz der Klasse *Instr* liefern. Auf diese zurückgegebene Instanz kann dann die *exec*–Methode zur Verarbeitung angewandt werden. Diese Befehle[2] implementieren die Befehle, die die Virtuelle Maschine direkt versteht und verarbeiten kann.

2.5 Opcodes

Durch den Befehl *compute* können weiter Berechnungschritte eingeleitet werden. Im Gegensatz zu den grundlegenden Befehlen (siehe Kapitel 2.4 Instructions) werden Berechnungsschritte[3] für den erweiterten Befehlssatz benutzt, der den Befehlssatz der ALU repräsentiert[4].

Abbildung 2.4:
OMT: Compute



[2] Befehle, die eine Verarbeitung ohne Variablenmanipulation vornehmen
[3] Bildbearbeitungsbefehle
[4] Befehle, die eine direkte Variablenmanipulation zur Folge haben oder auf diesen arbeiten

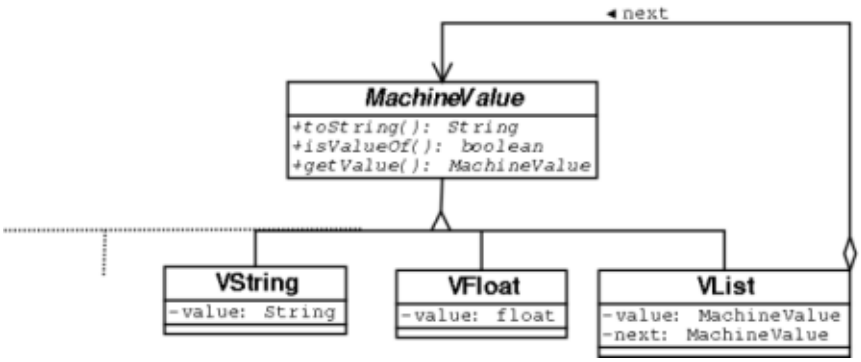
Peter Blicharski, II3839
Malte Hopmann, II4271

3 Grundlegende Datenstrukturen

3.1 MachineValue

Diese abstrakte Klasse dient zur Implementierung aller, von der Virtuellen Maschine zur Verfügung gestellten Datentypen.

Abbildung 3.1:
OMT: MachineValue



Durch die diese Art der Implementierung ist es einfach neue Datentypen hizuzufügen. Man muß sich nur an die Klasse *MachineValue* halten.

3.2 PC

Enthält die Eigenschaft des aktuellen Befehlszählers und folgende Methoden:

- getPC
- setPC
- legalPC

Den Programmzähler zu inkrementieren und Sprünge aus Sicht des Befehlszählers zu realisieren bleibt der Klasse *MachineState* vorbehalten.

3.3 MSStatus

Um Laufzeitfehler zu erfasssen und eine geeignete Fehlerabarbeitung zu ermöglichen soll diese Klasse dienen. Sie besteht zum einen aus dem Fehlerbit *OK*, der den Fehlerfall signalisiert, und zum anderen

aus dem Fehlerstring *exception*, der eine benutzerfreundliche Ausgabe ermöglicht. Folgende Methoden sind unabdinglich:

- getExc
- setExc
- isStatusOk

3.4 Mem

Durch diese Klasse wird der Heap realisiert. Gespeichert werden alle Daten vom Typ *MachineValue*. Folgende Methoden sind implementiert:

- writeMem
- readMem
- legalMemAddr

3.5 Stack

Der Stack beerbt die Klasse *java.util.stack* underweitert diese nur um die Eigenschaft *MachineValue*. dadurch erhalten wir einen typisierten Stack, der nur Werte von diesem Typ aufnehmen kann. Es werden die geerbten Methoden:

- push
- pop
- seek

benutzt. Zu prüfen ist, ob es nützliches ist auf Stack–Overflow zu testen.

Peter Blicharski, II3839
Malte Hopmann, II4271

Literaturverzeichnis

[1] FH–Wedel, Compilerbau Vorlesung, Prof. Schmidt,
http://www.fh-wedel.de/ si/vorlesungen/cb/cb.html

[2] FH–Wedel, Compilerbau PPL, Prof.Schmidt,
http://www.fh-wedel.de/ si/vorlesungen/cb/Beispiele/parser/haskell/pl0/inhalt.html

[3] Official Website,
http://www.haskell.org

[4] PPL–Virtuelle Maschine und weiteres,
http://www.fh-wedel.de/ si/vorlesungen/cb/Beispiele/parser/haskell/pl0/ppl.tar.gz