Musterlösungen zur Klausur "Neuere Programmiersprachen" am 259.1999

Zu Aufgabe 1

```
type KOEFFIZIENTEN is
        array (INTEGER range <>) of INTEGER;


type POLYNOM (GRAD : INTEGER) is
    record
        EINTRAEGE : KOEFFIZIENTEN (O..GRAD);
    end record;


function GRAD (P : POLYNOM) return INTEGER is
begin
    for 1 in reverse P.EINTRAEGE'RANGE loop
        if P.EINTRAEGE(I) /= 0 then
            return 1;
        end if;
    end loop;
    return -1;
end GRAD;

function AUSWERTUNG (P : in POLYNOM; X : in INTEGER) return INTEGER is
    WERT : INTEGER;
    G : INTEGER;
begin
    G := GRAD(P);
    WERT : = P.EINTRAEGE(G);
    for 1 in reverse O..(G-1) loop
        WERT := WERT * X + P.EINTRAEGE(I);
    end loop;
    return WERT,
end AUSWERTUNG;
```

Zu Aufgabe 2

Teil a)

Der Typ PRIORITY könnte auch außerhalb des Pakets definiert sein. Wird er im Paket definiert, so darf er nicht private sein.

```
package PRIORITY-QUEUE is

    subtype PRIORITY is INTEGER range 1..100;

    procedure ENQUEUE (E: in ELEMENT; P: in PRIORITY);
    -- Fügt das Element E mit der Priorität P in die Priority Queue ein, falls dieses
    -- Element in der Priority Queue noch nicht vorkommt.

    procedure DEQUEUE (E: out ELEMENT);
    -- Liefert im Ausgabeparameter E das Element mit der höchsten Priorität und
    -- löscht es aus der Priority Queue.

    function EXISTS (E: in ELEMENT) return BOOLEAN;
    -- Testet, ob das Element E in der Priority Queue vorhanden ist

    function ISEMPTY return BOOLEAN;
    -- Testet, ob die Priority Queue leer ist.

    procedure DELETE (E: in ELEMENT);
    -- Löscht das Element E aus der Priority Queue.

end PRIORITY-QUEUE;
```

Teil b)

```
procedure CHANGE-PRIORITY (      OLD-ELEMENT in ELEMENT,
                                 NEW_PRIORITY: in PRIORITY) is
begin
    if EXISTS (OLD-ELEMENT) then
        DELETE (OLD-ELEMENT);
        ENQUEUE (OLD-ELEMENT, NEW-PRIORITY);
    end if;
end CHANGE-PRIORITY,
```

Zu Aufgabe 3

Zunächst formt man die Gleichung so um, daß auf der rechten Seite 0 steht. Dann schreibt man für die linke Seite eine Ada Funktion GLEICHUNG. Mit dieser wird schließlich eine Ausprägung der generischen Funktion SOLVE erzeugt, hier SOLVE-GLEICHUNG genannt, die man anschließend aufruft, um eine Lösung der Gleichung zu erhalten.

```
function GLEICHUNG (X: FLOAT) return FLOAT is
begin
    return EXP(X) + X - 6.0;
end GLEICHUNG;

function SOLVE-GLEICHUNG is new SOLVE (GLEICHUNG);

LOESUNG := SOLVE-GLEICHUNG;
```

Zu Aufgabe 4

Für den FIFO-Puffer wird eine Schlange realisiert, und zwar mit einem Array der Länge CAPACITY und zwei Indices HEAD und TAIL, die zyklisch über das Array laufen. HEAD zeigt auf das Feld, in das als nächstes eingefügt werden kann, TAIL auf das Feld, aus dem aktuell entfernt werden kann. Die Schlange ist leer, wenn HEAD = TAIL ist. Sie ist voll, wenn HEAD + 1 = TAIL, die genaue Bedingung lautet (HEAD mod CAPACITY) + 1 = TAIL. Es ist dann zwar noch ein Feld frei, so kann man aber ohne zusätzlichen Aufwand eine volle von einer leeren Schlange unterscheiden.

```
task PRODUZENT is
end PRODUZENT,

task PUFFER is
    entry PUT ( P: in PRODUKT );
    entry GET ( P: out PRODUKT );
end PUFFER;

task KONSUMENT is
end KONSUMENT,
```

```
task body PRODUZENT is
    P : PRODUKT;
begin
    loop                                -- In einer Endlosschleife wird jeweils
        PRODUZIERE ( P );               -- ein Produkt produziert und in den Puffer
        PUFFER.PUT ( P );               --  eingefügt.
    end loop;
end PRODUZENT,


task body PUFFER is
    CAPACITY : constant := 100;
    BUFFER : array ( 1 ..CAPACITY ) of PRODUKT;
    HEAD : POSITIVE := 1;
    TAIL : POSITIVE := 1;
begin
    loop
        select
            when not ((HEAD mod CAPACITY) + 1 = TAIL) =>
                                                -- Wenn Puffer nicht voll...
            accept PUT ( P: in PRODUKT ) do
                BUFFER (HEAD) := P;
                HEAD := (HEAD mod CAPACITY) + 1;
            end PUT,
        or
            when not ( HEAD = TAIL ) =>         -- Wenn Puffer nicht leer...
            accept GET ( P: out PRODUKT ) do
                P := BUFFER (TAIL);
                TAIL := (TAIL mod CAPACITY) + 1;
            end GET,
        or
            terminate;
        end select;
    end loop;
end PUFFER;


task body KONSUMENT is
    P: PRODUKT;
begin
    loop                                -- In einer Endlosschleife wird jeweils
        PUFFER.GET ( P );               -- ein Produkt aus dem Puffer geholt und
        KONSUMIERE ( P );               --  konsumiert.
    end loop;
end KONSUMENT,
```

Zu Aufgabe 5

Die Position des letzten schreibenden Zugriffs sowie der Wert an dieser Position werden im den Instanzvariablen *lastPos* und *lastValue* gespeichert, um den ursprünglichen Zustand wiederherstellen zu können.

class name	UndoArray
superclass	Array
instance variable names	lastPos lastValue

```
at: anInteger put: anObject
    lastPos := anInteger.
    lastValue := self at: anInteger.
    ^super at: anInteger put: anObject

undo
    lastPos notNil ifTrue: [
        self at: lastPos put: lastValue.
        lastPos := nil].
```

Zu Aufgabe 6

Teil a)

```
timesRepeat: aBlock
    1 to: self do: [ :index | aBlock value ]
```

Teil b)

Im allgemeinen dienen Blöcke dazu, Programmstücke mehrfach oder auch gar nicht ausführen zu lassen. Bei ***and:*** und ***or:*** geht es darum, den zweiten Teil einer Konjunktion bzw. Disjunktion nicht auszuführen, entweder um die Ausführung zu beschleunigen oder um Laufzeitfehler zu vermeiden, wie z.B. bei:

```
(x > 0) and: [ a / x > 0 ]
```

Zu Aufgabe 7

Teil a)

Die Methode erzeugt eine neue Collection, die die gleiche Klasse und die gleiche Größe wie die Empfänger-Collection hat. Dann wird für jedes Element der Empfänger-Collection *aBlock* ausgewertet und das Ergebnis in die neue Collection eingefügt. Am Ende wird die neue Collection zurückgegeben. Der Originalname ist *collect:*.

Teil b)

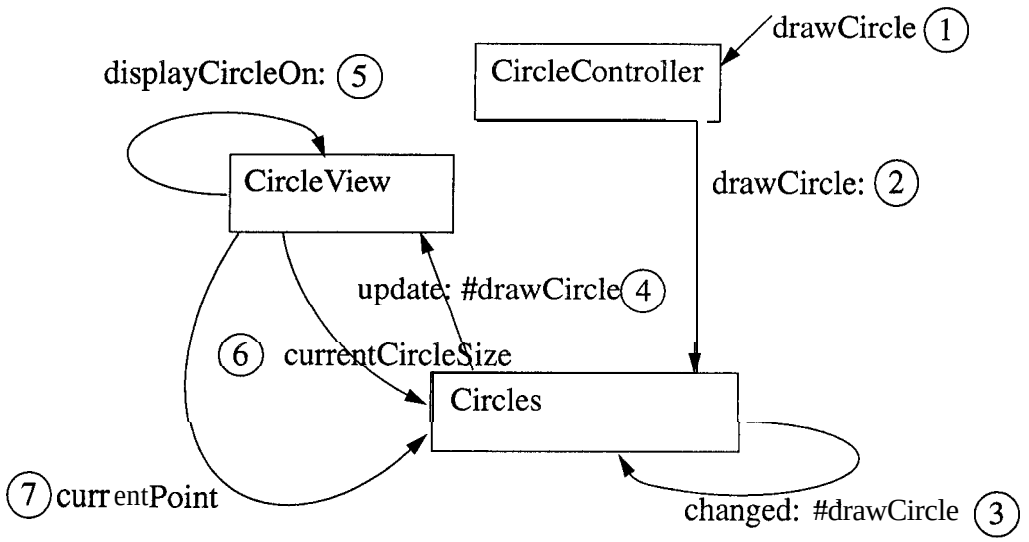
detect: aBlock ifNone: exceptionBlock

```
self do: [ :each | (aBlock value: each) ifTrue: [^each]].
^exceptionBlock value
```

Zu Aufgabe 8

Teil a)

Eine solche Darstellung kann- man unterschiedlich detailliert machen. Welches Objekt die Ausgangsbotschaft *drawCircle* sendet, soll offen bleiben.



Teil b)

In der Klasse *CircleController* wird lediglich die Beschriftung der Menüeinträge geändert. Die gesendete Botschaft *drawCircle* müßte man eigentlich auch umbenennen (etwa *drawCircleOrSquare* oder *drawFigure*).

menu

```
^ Menu
    labels: 'Kreis/Quadrat Zeichnen \ Loeschen' withCRs
    values: #(#drawCircle #cleanUp)
```

Die Klasse *Circles* braucht zusätzliche Instanzvariablen für die neue Auswahlliste (*figureListHolder*). Dementsprechend werden auch Zugriffsmethoden benötigt (*figureListHolder*). In der Methode *initialize* wird die neue Instanzvariable geeignet initialisiert. Schließlich wird noch *openWindow* gemäß der neuen Fensteraufteilung geändert. Um den Änderungsaufwand gering zu halten, wird die Methode *drawCircle:* nicht umbenannt (s.o.). Ebenso wird der Aspekt *#drawCircle* nicht geändert.

class name Circles
super class Model
instance variable names currentPoint circeSizeListHolder figureTypeListHolder

instance methods

initialize

```
"Initialisierung der darzustellenden Liste und der Listenposition."
| circesizelist figureTypeList |
circesizelist := List new.
circesizelist add: '20'.
circeSizeList add: '30'.
circeSizeList add: '50'.
circesizelist add: '80'.
circeSizeList add: ' 100'.
circeSizeListHolder := SelectionInList new.
circeSizeListHolder list: circesizelist.
circeSizeListHolder selection: '20'.

figureTypeList := List new.
figureTypeList add: 'Kreis'.
figureTypeList add: 'Quadrat'.
figureTypeListHolder := SelectionInList new.
figureTypeListHolder list: figureTypeList.
figureTypeListHolder selection: 'Kreis'.
```

circeSizeListHolder

```
"Liefert die Liste der Kreisgrößen."
^ circeSizeListHolder
```

figureTypeListHolder

"Liefert die Liste zur Auswahl zwischen Kreisen und Quadraten."

```
^ figureTypeListHolder
```

currentCircleSize

"Liefert den aktuellen Kreisdurchmesser. Mit *asNumber* wird der String, der die aktuell selektierte Kreisgröße darstellt, in eine Zahl umgewandelt."

```
^ circeSizeListHolder selection asNumber.
```

currentFigureType

"Liefert 'Kreis' oder 'Quadrat'."

```
^ figureListHolder selection.
```

currentPoint

"Liefert den aktuellen Punkt."

```
^ currentPoint.
```

drawCircle: aPoint

"Der neue Punkt wird gespeichert und der Update-Mechanismus ausgelöst."

```
currentPoint := aPoint.
self changed: #drawCircle.
```

cleanUp

"Der Update-Mechanismus wird ausgelöst."

```
self changed: #cleanUp.
```

class methods

openWindow

"Die Anwendung wird initialisiert, das Fenster zusammengebaut und geöffnet."

```
| aModel aCircleView aSizeListHolder aSizeListView aFigureTypeListHolder
aFigureTypeListView aCompositePart aWindow |
aModel := self new initialize.

"Die CircleView für die Zeichenfläche"
aCircleView := CircleView
                    model: aModel.

"Die SequenceView für die Kreisgrößen"
aSizeListHolder. := aModel circeSizeListHolder.
aSizeListView := SequenceView new.
aSizeListView       model: aSizeListHolder listHolder;
                    selectionChannel: aSizeListHolder selectionIndexHolder.
```

```
"Die SequenceView für die Auswahl zwischen Kreis und Quadrat"
aFigureTypeListHolder. := aModel figureTypeListHolder.
aFigureTypeListView := SequenceView new.
aFigureTypeListView    model: aFigureTypeListHolder listHolder;
                       selectionChannel: aFigureTypeListHolder selectionIndexHolder.

aCompositePart := CompositePart new.
aCompositePart     add: aCircleView
                   borderedIn: ( 0 @ 0 extent: 0.8 @ 1).
aCompositePart     add: aSizeListView
                   borderedIn: ( 0.8 @ 0 extent: 0.2 @ 0.5).
aCompositePart     add: aFigureTypeListView
                   borderedIn: ( 0.8 @ 0.5 extent: 0.2 @ 0.5).
aWindow := ScheduledWindow
                   model: aModel
                   label: ' Kreise und Quadrate
                   minimumSize: 400 @ 300.
aWindow component: aCompositePart.
aWindow open.
```

In *CircleView* wird die Methode *update:* geändert. Als Reaktion auf den Aspekt *#drawCircle* wird zunächst geprüft, ob ein Kreis oder ein Quadrat gezeichnet werden soll (mit der Botschaft *actKindOfObject* ans Model). Für das Zeichnen des Quadrats wird eine neue Methode *displaySquareOn:* geschrieben.

update: aSymbol

```
"Abhängig vom Parameter wird eine Zeichenoperation ausgeführt. Das Zeichnen erfolgt
dabei jeweils sowohl auf dem Bildschirm als auch in der Pixmap."
aSymbol = #drawCircle iffrue:
    [   model currentFigureType = Kreis' iffrue:
            [   self displayCircleOn: self graphicscontext.
                self displayCircleOn: pixmap graphicscontext ].
        model currentFigureType = 'Quadrat' iffrue:
            [   self displaySquareOn: self graphicscontext.
                self displaySquareOn: pixmap graphicscontext ] ].
aSymbol = #cleanUp iffrue:
    [   self displayWhiteOn: self graphicscontext.
        self displayWhiteOn: pixmap graphicscontext ].
```

displaySquareOn: aGC

```
"Zeichnet ein Quadrat auf dem graphischen Kontext aGC. Alle benötigten Daten werden
vom Model erfragt."
| extent |
extent :=(model currentCircleSize) @ (model currentCircleSize).
aGC paint: ColorValue black.
aGC lineWidth: 1.
aGC displayRectangularBorder: (model currentPoint extent: extent).
```