Abschnitt 7

Pakete

Moduln

Beispiel: Stack

Private Typen

Objekte (aus der Grundvorlesung übernommen)

Pakete sind Realisierungen von Moduln in Ada

Ein Modul ist eine in sich abgeschlossene Programmeinheit, die aus Konstanten, Variablen, (strukturierten) Datentypen und Algorithmen (Funktionen, Operationen, Unterprogrammen) besteht. Moduln bilden die Bausteine, aus denen ein großes Softwaresystem zusammengesetzt wird. Man spricht dann von einem "modularen" Aufbau solcher Systeme.

Ein Modul besteht aus zwei Teilen: der Spezifikation, die nach außen mitgeteilt wird, und der Implementierung, die nur innen bekannt ist. In der Spezifikation werden alle Größen, die von anderen Programmeinheiten genutzt werden dürfen, aufgeführt; in der Implementierung wird deren Realisierung in einer konkreten Programmiersprache geregelt.

Darstellung in Ada:

Der Spezifikationsteil und der Implementierungsteil werden voneinander getrennt. Der Spezifikationsteil, der im Programm früher als der Implementierungsteil stehen muss, hat die Form

package <Name des Pakets> is
<Folge von einfachen Deklarationen> end <Name des Pakets>;

Der Implementierungsteil heißt "body" (dt.: "Rumpf") und hat die Form

package body <Name des Pakets> is
<Folge von Deklarationen> end <Name des Pakets>;

Alle Teile der Spezifikation, die im Body programmiert werden, müssen wörtlich dort wieder vorkommen, s.u. Die Spezifikation darf nur Deklarationen und insbesondere keine Implementierungsteile (bodies) enthalten.

Standardbeispiel "Stack" für reelle Werte. Spezifikationsteil:

```
package  Stack  is
type StZelle;
type RefStZelle is access StZelle;
type StZelle is  record inhalt: float; next: RefStZelle; end record;
procedure Push (A: in float; S: in out RefStZelle);
procedure Pop (S: in out RefStZelle);
function Top (S: in RefStZelle) return float;
function Isempty (S: in RefStZelle) return Boolean;
function Empty return RefStZelle;
end Stack;
```

Anschließend (aber nicht unbedingt direkt danach) muss die Implementierung folgen (die Typen StZelle und RefStZelle wurden bereits in der Spezifikation vollständig deklariert):

```
package body  Stack  is

procedure Push (A: in float; S: in out RefStZelle) is
    begin  S := new StZelle'(A,S);  end;
procedure Pop (S: in out RefStZelle) is
    begin  S := S.next;  end;
function Top (S: in RefStZelle) return float is
    begin  return S.inhalt;  end;
function Isempty (S: in RefStZelle) return Boolean is
    begin  return S = null; end;
function Empty return RefStZelle is begin return null; end;

end Stack;
```


Einsatz des Stacks in einem Programm zum Spiegeln einer Folge reeller Zahlen:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Spiegeln_mit_Stack is
package Stack is begin ... end;          -- Spezifikationsteil, s.o.
package body Stack is ... end;           -- Implementierungsteil, s.o.
K: Stack.RefStZelle; C: float;

begin  K := Stack.Empty;
    while not End_Of_File loop
        get(C); Stack.Push(C, K); end loop;
    while not Stack.Isempty(K) loop
        put(Stack.Top(K)); Stack.Pop(K); end loop;
end;
```

Abkürzende Schreibweise mittels "use":

```
with Ada.Float_Text_IO; use Ada.Float_Text_IO;
procedure Spiegeln_mit_Stack is
package Stack is begin ... end;          -- Spezifikationsteil, s.o.
package body Stack is ... end;           -- Implementierungsteil, s.o.
use Stack;
K: RefStZelle; C: Character;

begin  K := Empty;
    while not End_Of_File loop
        get(C); Push(C,K); end loop;
    while not Isempty(K) loop
        put(Top(K)); Pop(K); end loop;
end;
```


Die Datentypen, mit denen man den Stack realisiert (hier: eine Liste; man könnte auch ein Feld verwenden), kann man geheim halten. In unserem Fall wird der Typ StZelle im Programm auch nicht gebraucht. Hierzu gibt man nur den Typ RefStZelle bekannt und verlagert die Realisierung in den Implementierungsteil. Um dies auszudrücken, bezeichnet man den bekannt gegebenen Typ in der Spezifikation als "private". Er kann dann zur Deklaration außerhalb des Pakets verwendet werden, man kann ihn aber nur mit Hilfe der Operationen "=", "/=" und ":=" und der Prozeduren und Funktionen im Spezifikationsteil bearbeiten. (Will man die Operationen =, /= und := dem Bnutzer auch noch verbieten, so schreibt man "limited private".) Diesen privaten Typ listet man dann hinter dem Schlüsselwort private am Ende der Spezifikation auf.

```
package Stack is
type RefStZelle is private;
procedure Push(A: in float; S: in out RefStZelle);
procedure Pop(S: in out RefStZelle);
function Top(S: in RefStZelle) return float;
function Isempty(S: in RefStZelle) return Boolean;
function Empty return RefStZelle;

private                -- Größen ab hier sind nicht mehr sichtbar
   type StZelle;
   type RefStZelle is access StZelle;
   type StZelle is  record inhalt: float; next: RefStZelle;
                    end record;
end Stack;
```

Implementierungsteil und Verwendung (Folien 5 und 6) bleiben unverändert.

Ein Paket ist selbst eine Deklaration. Es kann auch Konstanten und Variablen enthalten, die mittels der Punktnotation verwendet werden können, sofern sie sichtbar sind. Dies kann direkt oder über Funktionen erfolgen.

Das Paket darf am Ende der Implementierungsteils eine Initialisierung besitzen der Form begin <Anweisung>. Diese Initialisierung wird nur einmal und zwar beim Abarbeiten der Deklaration ausgeführt. Hiermit können insbesondere die Variablen des Pakets mit Anfangswerten versehen werden.

Wir modifizieren den obigen Stack leicht, indem wir den Speicher in den Implementierungsteil verlagern und zugleich dafüpr sorgen, dass kein Unterlauf entstehn kann. Nun ist alles außer Push, Pop und Top verborgen.

Spezifikation (auch Schnittstelle genannt):

```
package  Keller  is
Bottom: constant Float := 2003.0;
procedure Push (A: in float);
procedure Pop;
function Top return float;
end Keller;
```

Es folgt der Implementierungsteil, der nach außen unsichtbar bleibt. Die Konstante Bottom ist dort bekannt, weil sie im Spezifizierungsteil deklariert wurde.


```
package body  Keller  is
type StZelle;
type RefStZelle is access StZelle;
type StZelle is record inhalt: float; next: RefStZelle; end record;
KS: RefStZelle;
procedure Push (A: in float) is
    begin  KS := new StZelle'(A, KS);  end;
procedure Pop return RefStZelle is
    begin  if KS.next /= null then KS := KS.next; end if;  end;
function Top return float is
    begin  return KS.inhalt;  end;
begin KS := Push(Bottom);    -- Initialisierung des Pakets
end Keller;
```

Einsatz dieses Pakets Keller in einem Programm zum Spiegeln einer Folge reeller Zahlen (mit der use-Klausel kann man noch das Präfix "Keller" sparen):

```
with Ada.Float_Text_IO; use Ada.Float_Text_IO;
procedure Spiegeln_mit_Keller is
package Keller is begin ... end;          -- Spezifikationsteil, s.o.
package body Keller is ... end;           -- Implementierungsteil, s.o.
C: float;

begin
    while not End_Of_File loop
        get(C); Keller.Push(C); end loop;
    while Keller.Top /= Keller.Bottom loop
        put(Keller.Top); Keller.Pop; end loop;
end;
```


Der Typ "float" kann natürlich durch einen anderen ersetzt werden und man erhält einen entsprechenden Stack.

Die Unabhängigkeit des Codes von konkreten Typen erreicht man durch die Generizität, die auch für Pakete gilt. Die Syntax ist im Wesentlichen wie bei Unterprogrammen.

```
generic  type Element is private;
Bottom: constant Element;
package  Keller  is
procedure Push(A: in Element);
procedure Pop;
function Top return Element;
end Keller;
```

(Es folgt dann der Implementierungsteil, in dem Element als ein Parameter für den konkreten Typ verwendet wird.)

Ein konkretes Paket deklariert man dann wie üblich:

```
package Zeichenstack is new Keller (Character,'ß');

with Ada.Float_Text_IO; use Ada.Float_Text_IO;
procedure Textspiegeln_mit_Keller is
generic ...
package Keller is begin ... end;          -- Spezifikationsteil, s.o.
package body Keller is ... end;           -- Implementierungsteil, s.o.
...
    declare package Zeichenstack is new Keller (Character,'ß');
    C: Character;
    begin
        while not End_Of_File loop
            get(C); Keller.Push(C); end loop;
        while Keller.Top /= Keller.Bottom loop
            put(Keller.Top); Keller.Pop; end loop;
    end; ...
```


Ein Modul sollte folgende Eigenschaften besitzen:

- Er bildet eine in sich abgeschlossene Einheit, die eine klar umrissene Aufgabe bearbeitet.
- Er hat eine genau definierte Schnittstelle nach außen (genannt "Spezifikation" oder "Interface"); nur die hier genannten Eigenschaften und Fähigkeiten sind nach außen hin sichtbar ("visibility"). Dies ist eine Art Benutzungsanweisung für alle, die diesen Modul einsetzen wollen.
- Seine Arbeitsweise ("Implementation") ist außen nicht bekannt. Er besitzt somit zwei Sichten (views): Die Außenansicht für den Benutzer (dies ist die Schnittstelle) und die Innensicht des Erstellers. Die Innensicht bleibt "gekapselt" oder nach außen "versteckt" (Prinzip des "information hiding").
- Er ist überschaubar, leicht zu testen und einfach zu warten.
- Er lässt sich in Bibliotheken aufbewahren und hierdurch leicht in beliebige Programmsysteme einbauen.

Moduln sind somit die einfachste Art, ein "Client-Server"-Modell zu realisieren:

Hierbei erstellt ein "Server" (Dienstleister, Hersteller) einen "Dienst", den er als Angebot (= Schnittstelle) nach außen bekannt macht. Wie er diese Dienstleistung tatsächlich realisiert, wird dabei nicht bekannt gegeben.

Ein "Client" (Kunde, Benutzer) benötigt eine Dienstleistung und verlässt sich darauf, dass die im Angebot eines Servers genannten Eigenschaften auch zutreffen.

Um dies Modell in die Praxis umzusetzen, benötigt man ein Netz (Internet), in dem ein Kunde die Angebote mit Hilfe von Suchmaschinen auffinden und sich mit Browsern anzeigen lassen kann. (Der Kunde "browst" und "surft" im Internet.)

with und use können in allen Deklarationsteilen stehen.

Erläuterung des Sprachelements with:
with M bedeutet, dass an dieser Stelle die Spezifikation des Moduls M sichtbar gemacht wird. Alles, was in dessen Spezifikation steht, kann ab hier über die Punktnotation "M. ---" benutzt werden.

Erläuterung des Sprachelements use:
use M bedeutet, dass ab dieser Stelle für einen Namen Funk, der in M vereinbart wird, die Punktnotation "M.Funk" durch Funk ersetzt werden kann, dass also die "Qualifizierung" (= der Zugriff auf die in M deklarierten Größen) ohne "M." erfolgen darf. Für Namenskonflikte ist der Programmierer selbst verantwortlich, wobei das Überladen erlaubt ist.

Altes Beispiel der Sortierprozedur neu vorgestellt:

```
generic
   Max: Positive;
   type T is private;
   with function ">"(L,R:T) return Boolean;
package Sortpaket is
   type VektorT is array (1 .. Max) of T;
   procedure Sort (A: in out VektorT);
end Sortpaket;
```



```
package body Sortpaket is
  generic   type Element is private;
  procedure Tausch (A, B: in out Element);
  procedure Tausch (A, B: in out Element) is
    H: Element; begin  H:=A; A:=B; B:=H; end Tausch;
  procedure Sort (A: in out VektorT) is
    Weiter: Boolean := True;
    procedure Austausch is new Tausch(T);
  begin
    while Weiter loop
      Weiter := False;
      for I in 1..Max-1 loop
        if A(I)>A(I+1) then Weiter := True; Austausch(A(I), A(I+1));
        end if;
      end loop;
    end loop;
  end Sort;
end Sortpaket;
```

Verwendung im weiteren Verlauf des Programms (dies wurde nicht verifiziert, bitte selbst in einem Programm prüfen!):

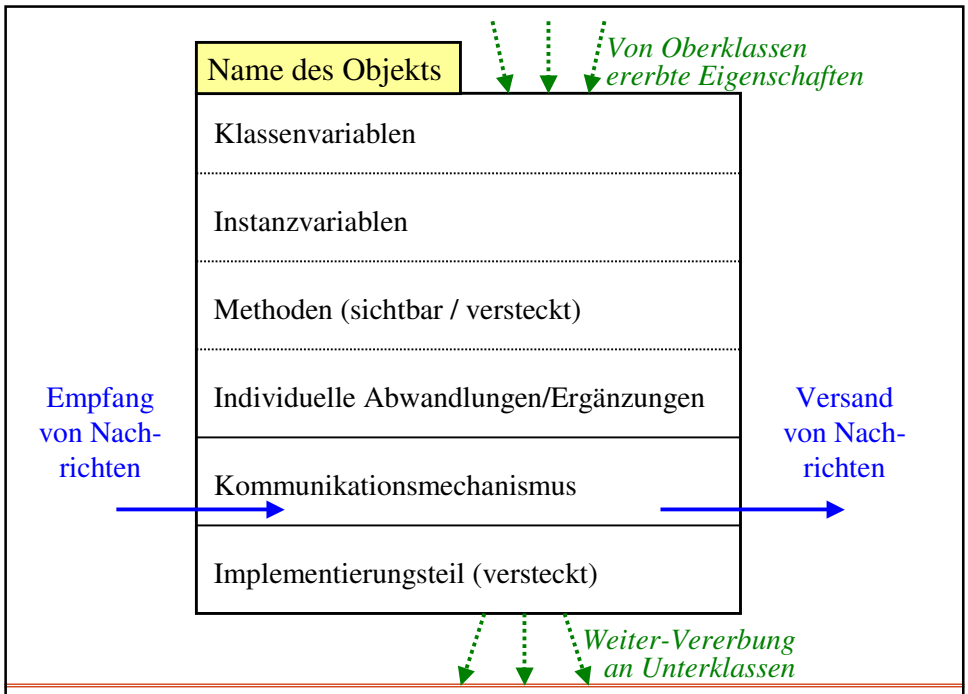
```
        -- Im Deklarationsteil:
with Sortpaket; with Ada.Integer_Text_IO;
package SortpaketInt is new Sortpaket(500, integer, ">");
R: SortpaketInt.VektorT;
        -- Im Anweisungsteil:
...                     -- Fülle das Feld R mit ganzen Zahlen
SortpaketInt.Sort(R);
...
```

Hinweis: Wenn Sie im package SortpaketInt die Operation ">" durch "<" ersetzen, so wird absteigend sortiert!

Objekte

Objekte sind in sich geschossene Einheiten, die

- wie Moduln aufgebaut sind: Es gibt ein Schema, genannt "Klasse", das vor allem aus "Attributen" (das sind die einzelnen Datenstrukturen der Variablen) und "Methoden" (das sind die algorithmischen Teile) einschl. der Angaben zur Sichtbarkeit besteht und aus dem ein neues Objekt erzeugt werden kann; das Objekt ist eine Instanz (oder ein "Exemplar" oder eine "Ausprägung") dieser Klasse.
- einen individuellen Zustand besitzen (Speicherzustand der Klassen- und Instanzvariablen),
- miteinander kommunizieren können; dies geschieht durch Nachrichtenaustausch ("message passing"),
- durch Vererbung ihre Eigenschaften an neue Objekte bzw. Klassen weitergeben können.



Prinzipien der Objektorientierung:

1. Es gibt nur Objekte. Jedes Objekt ist eindeutig identifizierbar über seinen Namen.
2. Alles wird über Klassen, Instanzbildung, Zustände, Methoden, Nachrichten und Vererbung realisiert.
3. Objekte handeln in eigener Verantwortung (und sie geben nur bekannt, *was* sie bearbeiten, niemals, *wie* sie dies tun).
4. Klassen werden in Bibliotheken aufbewahrt und stehen allen Programmen und Klassendefinitionen zur Verfügung.
5. Programmieren bedeutet, Klassen festzulegen, hieraus Objekte zu erzeugen und diesen Aufgaben zu übertragen, indem man ihnen geeignete Nachrichten schickt. Die Auswertung der Objekte erfolgt hierbei erst zur Laufzeit (Polymorphie, dynamische Bindung der Objekte an Variable).

Wenn "alles" Objekte sind, so sind konsequenterweise auch Klassen, Nachrichten und die (formalen) Parameter Objekte.

Eine Methode der Klasse *"Klasse"* ist "new". Diese Methode erzeugt aus der Klasse eine Instanz, also ein konkretes Objekt. Jede Klasse ist eine Unterklasse der Klasse *"Klasse"* und hat somit diese Methode ererbt, kann also Instanzen von sich selbst erzeugen.

Eine konkrete Nachricht, die ein Objekt A an ein Objekt B schickt, besitzt meist aktuelle Parameter. Diese sind ebenfalls Objekte. Ebenso erwartet das Objekt A, dass das Objekt B ihm eine Nachricht mit konkreten Objekten als aktuellen Parametern zurückschickt.

Hinweise (1):

a. Klassen bilden *Hierarchien* oder zyklenfreie Abhängigkeitsgraphen (Ober- / Unterklassen, einfache / mehrfache Vererbung).
b. In typisierten Sprachen verweist eine Variable auf ein Objekt ihrer Klasse oder einer ihrer Oberklassen. Anderenfalls muss die Zuordnung laufend auf ihre *Zulässigkeit* überprüft werden.
c. Ist das zu aktivierende Objekt identifiziert, so muss man ermitteln, *welches die angeforderte Methode konkret ist* (eventuell muss man diverse Oberklassen durchsuchen) und ob es sie ausführen kann.
d. Zu jeder objektorientierten Sprache gibt es umfangreiche *Klassenbibliotheken*, aus denen man sich sein Programm aufbauen kann.
e. In der Praxis benötigt man eine *Entwurfsumgebung*, um Programme im Team zu entwickeln, um Erläuterungen, Entwurfsprozesse und verschiedene Dokumentationen zu erstellen und um die Klassenbibliothek zu erweitern.

Hinweise (2): Probleme in der Praxis, kleine Auswahl:

f. Die Sprache muss Erweiterungsmöglichkeiten von Moduln haben, ohne dass hierbei die privaten Informationen bekannt werden. (Das ist oft nicht realisierbar. Ada: child library units.)
g. Es müssen verschiedene Sichtweisen auf ein Objekt möglich sein. (Hierfür kann man Mehrfachvererbung nutzen.)
h. Es müssen Teile getrennt voneinander übersetzbar sein und abgelegt werden, allerdings darf hierdurch die Sichtbarkeit nicht beeinträchtigt werden. (Stichwörter in Ada: separate, stub.)
i. Die Klassenbibliotheken sollten sowohl den Quellcode als auch bereits getrennt compilierte Teile besitzen, und zwar so, dass diese leicht in ein Programm (z.B. über with und use) eingefügt werden können.

Hinweise (3): Probleme in der Praxis, kleine Auswahl:

j. Die Eindeutigkeit von Namen ist zu gewährleisten, z.B. durch Umbenennung importierter Größen (in Ada renames:
```
with Stack_für_Zeichen;
X: StackZ renames Stack_für_Zeichen.S;
function "*" (X,Y: Vektor) return float renames Skalarprodukt;)
```
k. Die Sichtbarkeit in der Vererbungshierarchie ist genau festzulegen. (Beispiel: In der Regel sehen Unterklassen nicht, wie ihre Oberklassen die Methoden realisiert haben; daher können sie diese auch nicht verwenden, um Umdefinitionen vorzunehmen. Ändert man eine Methode ab, so muss man aber meist Zugriff auf jene soeben ausgeblendete Methode haben. In Ada: über Punkt-Notation. In Java durch "super".)
l. Die Sichtbarkeitsregeln müssen auch in der Klassenbibliothek gelten. Beispielsweise wird durch "with" die Sichtbarkeit einer anderen Klasse importiert (wann und wo endet diese?).

Hinweise (4): Probleme in der Praxis, kleine Auswahl:

m. Wie entwirft man "objektorientiert"? Man unterscheidet zwischen OOA und OOD, also "objektorientierte Analyse" und "objektorientierter Entwurf" ("D" = design = Entwurf). In der OOA wird ein Sollkonzept/Pflichtenheft erstellt und für dieses werden geeignete Klassen mit Zusatzinformationen (zeitliche Abläufe, einzuhaltende Bedingungen, Zusammenwirken der Einheiten, ...) erarbeitet. Im OOD werden hieraus die tatsächlichen Klassen, möglichst aus einer Klassenbibliothek und ergänzt um zusätzlich erforderliche Hilfs-Klassen erstellt und die Realisierung in einer Programmiersprache skizziert. (Anschließend folgt die Codierung.)

n. Die Zeit, dass man Lösungen zu Problemen von Grund auf neu schrieb, ist vorbei. Heute versucht man, eine Problemlösung so zu beschreiben, dass sie mit Hilfe der vorhandenen Klassen realisiert werden kann. Nur für wenige Probleme werden noch neue Klassen, neue Datentypen, neue Vorgehensweisen entwickelt (die anschließend in die Klassenbibliothek übernommen werden).

Beispiel: Addieren von Zahlen

Zur Illustration des Prinzips objektorientierter Denkweise:
Die Addition zweier Zahlen "7 + 28" läuft aus objektorientierter Sicht folgendermaßen ab:

Das Objekt 7 bekommt die Nachricht, es möge seine Methode "+" auf sich und das beigefügte Objekt (aktueller Parameter 28) anwenden und das Ergebnis zurückschicken.

Die Zahl 7 ist eine Instanz der Klasse "integer", die aus einem Zustand (dies ist der Wert 7 seiner Instanzvariablen INH) und aus den zulässigen Methoden "+", "abs", "*", "-" usw. sowie den allgemeinen Methoden "send" (=schicke das Objekt, das auf "send" folgt, an den Sender zurück), "write" (drucke den Wert von INH aus) usw. besteht.

Wir haben also eine Klasse "integer" (Oberklasse sei "Zahlen")

integer	Vererbt von "Zahlen" werden: Typ "Binärfolge" und hierauf die Operationen plus, minus, mal, "<", "=", wie sie üblich sind. Der Ausbau zu "integer" erfolgt jetzt:
Klassenvariablen	Null: constant Binärfolge := 0; Zehn: constant Binärfolge := 1010
Instanzvariablen	INH: Binärfolge
Methoden	function "+" (X: integer) return integer; function quad return integer; function "abs" return Binärfolge; ...
Kommunikation	procedure return1 (A:Binärfolge) is begin X := new Integer; X.INH := A; send X end; procedure send (A:Object) is begin "schicke A an den Sender zurück" end; ...
Implementierung	procedure "+" (X: integer) is begin return1 (plus (self.INH, X.INH)) end; procedure abs is begin if self.INH<0 then send minus(Null,self.INH) else send self.INH fi end; procedure quad (X: integer) is begin return1 (mal (self.INH, self.INH)) end; ...

Es bleibt der jeweiligen Sprache überlassen, ob die im sichtbaren Bereich aufgelisteten Funktionen tatsächlich als Funktionen oder auf andere Weise (z.B. wie hier als Prozeduren; aber durch "return1" wird die richtige Sicht nach außen hergestellt) implementiert werden.

Z := 7 + 28 wird daher wie folgt ausgerechnet (die beiden Zahlen werden als Binärfolge dargestellt):

X := new integer; X.INH := 111;

Y := new integer; Y.INH := 11100;

Z := X."+"(Y);

X."+"(Y) bedeutet: Schicke an X die Nachricht, dass dessen Operation "+" ausgeführt werden solle. Diese Funktion erwartet ein Objekt der Klasse integer als Parameter, daher wird Y als aktueller Parameter mitgegeben. Das Objekt X antwortet dann mit einem Objekt der Klasse integer, das an die Variable Z gebunden wird.

Beispiel: Geometrische Größen

Geometrische Größen entwickelt man gerne auseinander. Man beginnt mit einem Punkt, geht zur Strecke und zum Polygon über, dann betrachtete man Dreieck, Viereck und n-Ecke, führt Kreise (= Punkt plus Strecke) und Ellipsen ein usw.

Zugleich kann man schrittweise weitere Eigenschaften hinzufügen (insbesondere die Fläche) und Unterklassen bilden (z.B.: Viereck → Trapez → Raute → Quadrat), wobei zugleich die Flächenberechnung jeweils neu definiert werden kann.

Die Darstellung verändern wir nun etwas, indem wir die Objekte wie Moduln ausformulieren und die Oberklasse(n) explizit angeben.

Wir definieren also eine Klasse "Punkt" (Oberklasse sei real):

Punkt	real
X: real; Y: real;	
procedure Drucken	
procedure Drucken is begin < sende an das Objekt *Drucker* die Nachricht drucke_ein_Pixel_an_die_Position (self.X, self.Y) > end	

Wir definieren nun eine Klasse "Strecke":

Strecke	Punkt, real
Anfang: Punkt; Ende: Punkt;	
procedure Drucken, DruckeAnfang, DruckeEnde; function Länge ( ) return real	
procedure Drucken is begin < sende an das Objekt *Drucker* die Nachricht drucke_eine_ Strecke_von_bis (self.Anfang.X, self.Anfang.Y, self.Ende.X, self.Ende.Y) > end; procedure DruckeAnfang is begin Anfang.Drucken end; procedure DruckeEnde is begin Ende.Drucken end; function Länge ( ) return real is begin return (square_root(quad(self.Anfang.X - self.Ende.X) + quad(self.Anfang.Y - self.Ende.Y))) end;	

Wir definieren nun eine Klasse "Polygon" mit dem generischen Parameter N (Oberklassen sind Strecke und integer):

Polygon (N: natural, N > 2)	Strecke, natural
Eckpunkte: array [1..N] of Punkt; private Streckenzug: array [1..N] of Strecke;	
procedure Drucken; function Länge ( ) return real	
procedure Drucken is var I: natural; begin for I := 1 to N do Streckenzug[I].Drucken od end; function Länge ( ) return real is var I: natural; L: real := 0.0; begin for I := 1 to N do L := L + Streckenzug[I].Länge od; return L end;	

var I, K: natural;
begin for I := 1 to N do Streckenzug[I] := new Strecke;
Streckenzug[I].Anfang := Eckpunkte[I];
if I = N then K:=1 else K:=I+1 fi;
Streckenzug[I].Ende := Eckpunkte[K] od
end
	Initialisierung

Beachten Sie hierbei:

Mit der Klasse "Strecke" ist auch deren Oberklasse "Punkt" eine Oberklasse von Polygon. Daher können wir deren Eigenschaften hier verwenden.

Am Ende haben wir einen Initialisierungsteil angefügt, mit dem wir die Folge der Strecken, die durch die Eckpunkte definiert ist, errechnen. Diese Information geben wir nach außen nicht bekannt ("private"). Den Streckenzug verwenden wir intern zum Drucken und zur Längenberechnung.

Sie erkennen hier einen Vorteil der OOP (= objektorientierten Programmierung): Wenn wir die Klasse "Punkt" von "real" auf "integer" umschreiben würden, so braucht an Strecke und Polygon nichts geändert zu werden! (Wiederverwendbarer Quellcode.)

Nun müsste man eigentlich noch

- eine Boolesche Funktion "kreuzungsfrei" hinzufügen, die genau dann den Wert true ergibt, wenn sich je zwei Strecken des Polygons nicht schneiden. Hierzu würde man in der Klasse "Strecke" eine Methode einfügen:

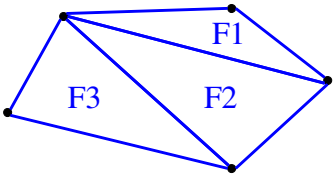
  function schnitt (S: Strecke) return Boolean is
  var ...
  begin ... end;

  (Selbst ausformulieren; Erläuterungen auf später folgenden Folien.)

- eine Boolesche Funktion "konvex" hinzufügen, die genau dann den Wert true ergibt, wenn das Polygon kreuzungsfrei ist (dann hat es ein "Inneres") und wenn alle Winkel zum Inneren des Polygons hin höchstens 180 Grad betragen.

Nun müsste man eigentlich noch

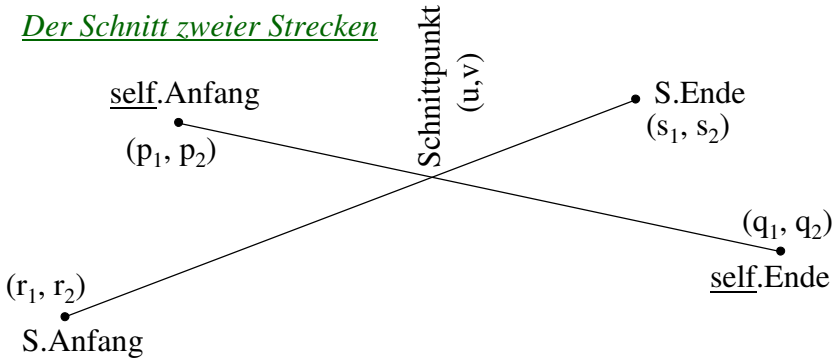
- für konvexe Polygone eine "Fläche" hinzufügen. Diese erhält man, indem man von einem Punkt aus alle aufeinander folgenden Dreiecksflächen aufsummiert (die Dreiecksfläche ist durch drei Seitenlängen eindeutig bestimmt und hieraus leicht zu berechnen); Beispiel:



Gesamtfläche =
F1 + F2 + F3.

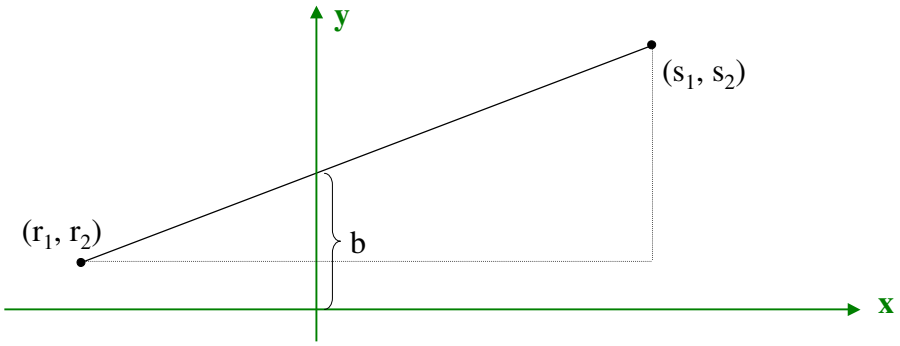
- die Flächen ausweiten auf kreuzungsfreie Polygone.
- den Schwerpunkt eines Polygons bestimmen oder andere "Mittelpunkte".

Der Schnitt zweier Strecken



Den Schnittpunkt erhält man als Schnittpunkt der beiden Geraden, die zu den Strecken gehören. Für zwei Geraden $y = a_1 \cdot x + b_1$ und $y = a_2 \cdot x + b_2$ ergibt sich der Schnittpunkt (u,v) durch $v = a_1 \cdot u + b_1$ und $v = a_2 \cdot u + b_2$, also für $a_1 \neq a_2$:

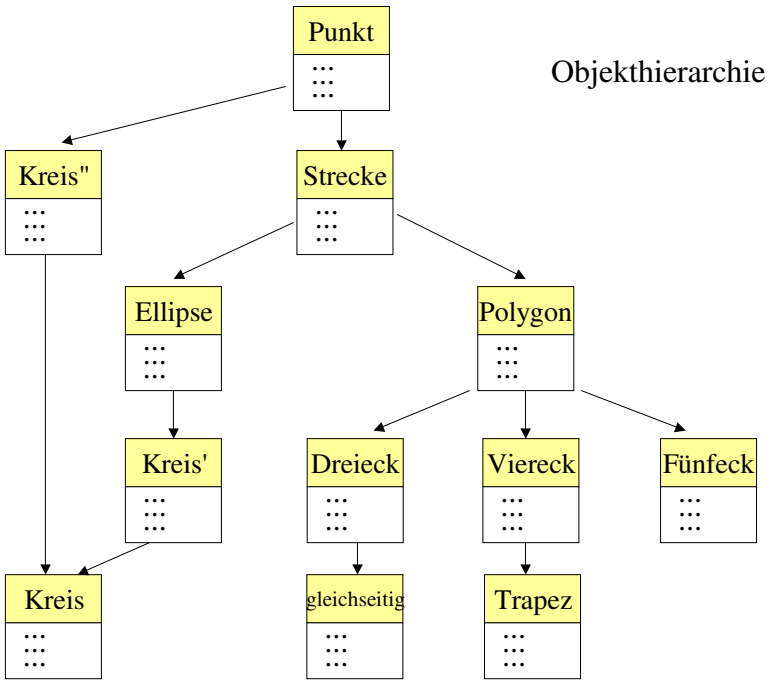
$$u = \frac{b_2 - b_1}{a_1 - a_2} \quad \text{und} \quad v = a_2 \cdot \frac{b_2 - b_1}{a_1 - a_2} + b_2$$



Ermittlung der Steigung a und des y-Abschnitts b der zu einer Strecke gehörenden Geraden $y = a \cdot x + b$:

$$a = \frac{s_2 - r_2}{s_1 - r_1} \quad \text{und} \quad b = s_2 - a \cdot s_1$$

Ein Schnitt liegt vor, wenn die x- und die y-Koordinate des Schnittpunkts echt zwischen den entsprechenden Koordinaten der beiden gegebenen Strecken liegt. Bitte selbst zu Ende durchdenken und fertig programmieren.



Objekthierarchie