Eiffel-Die Programmierspra-che

Vorstellung der OO-Programmier-sprache Eiffel

Ursprünge Eiffels

Eiffel ist nach Gustave Eiffel bannannt. Eiffel wurde 85' bis 87' von Bertrand Meyer entworfen und von seiner Firma realisiert. Dem Autor haben alle anderen Sprachen nicht gefallen, und so hat er eben selbst eine gebaut. Pate standen dabei SIMULA, ADA und MODULA-2.

Spracheigenschaften

- Wirth-artige Sprache
- Groß-kleinschreibung unwichtig
- OO-Sprache
- Minimalistisch
- Einmalprozeduren
- Ausnahmebehandlung

Minimalismus

- Keine Statischen Klassen
- Keine Felder als Sprachmittel
- Keine globalen Symbole
- Kein überladen von Methoden
- Kein case
- Keine geschachtelten Klassen
- Keine Standardparameter

OO-Eigenschaften

- Mehrfachvererbung
- Generizität (template)
- Alle methoden sind Virtuell
- Es gibt Klassenwälder (mehrere Wurzeln)
- Klassenmerkmale können explizit an andere Klassen exportiert werden
- Aufgeschobene Klassen

Weitere Eigenschaften

- Ursprünglich Eiffel->C -Compiler
- Umsetzung der Vertragsmetapher
- Schnittstellen zu anderen Sprachen
- Garbage Collection
- Objektauslagerung
- Variablen sind stets mit 0 initialisiert

Beispiel

```
deferred class STACK [G]

inherit
        DISPENSER [G]
                export    {NONE} prune, prune_all
                redefine
                        extend, force, put, fill
                end

feature -- Element change

        extend (v: like item) is
                        -- Push v onto top.
                deferred
                ensure
                        item_pushed: item = v
                end;

        put (v: like item) is
                        -- Push v onto top.
                deferred
                ensure
                        item_pushed: item = v
                end;

        fill (other: LINEAR [G]) is
                local
                        temp: ARRAYED_STACK [G]
                do
                        create temp.make (0);
                        from
                                other.start
                        until
                                other.off
                        loop
                                temp.extend (other.item);
                                other.forth
                        end;
                        from
                        until
                                temp.empty or else notextendible
                        loop
                                extend (temp.item);
                                temp.remove
                        end
                end;

end -- class STACK
```

Schlüsselwörter

and	debug	Equal	invarint	not	Result
as	defered	external	is	old	retry
BOOLEAN	div	false	language	once	STRING
check	Do	feature	like	or	then
class	else	Forget	local	REAL	true
CHARACTER	elsif	from	loop	redefine	until
Clone	end	if	mod	rename	variant
Create	ensure	inherit	name	require	Void
Current	export	INTEGER	nochange	rescue	

Erläuterung von Schlüsselwörtern

- Void — Prüft auf Nichtigkeit
- like — liefert den Typ des Arguments
- Current — entspricht SELF, this
- Result — Ergebnisvariable einer Funktion
- Create — Erzeugt ein Objekt (opt. Parameter)
- Forget — Vergisst das assoziierte Objekt
- Clone — Kopier-Konstruktor
- Equal — Flacher Objektvergleich

Datentypen

Wie in Java gibt es die abstrakte Datentypen (Klassen) und die Basistypen (INTEGER, CHARACTER, REAL, BOOLEAN).

Sie benötigen genau so viel Platz, wie ihre Daten, haben jedoch vom Compiler her Methoden. Eine Sonderrolle spielt STRING. Dabei handelt es sich um eine Klasse aus der Bibliothek. Vernünftigerweise erlaubt der Compiler Stringliterale. Diese sind dann dierekt wieder Objekte.

Von diesen Klassen kann geerbt werden.

Operatoren & Interpunktion

Vergleichsoperatoren

=
/=
>
<
>=
<=

Arithmetische Operatoren

+
-
* /
^ mod
div

ADT

Create
Forget
Clone

Bool'sche Operatoren

and and then
or or else
not

Kommentar

--

Rest

; ,
: .

Zuweisung

:=

Besondere Boolsche Ausdrücke

- a **and** b
  - vollständige Auswertung
- a **or** b
  - vollständige Auswertung
- a **and then** b
  - b wird nur ausgewertet, wenn a true war.
- a **or else** b
  - b wird nur ausgewertet, wenn a false war.
- **not** a

Funktionen, Prozeduren und lokale Variablen

```
class ALTER
export set, get
feature
  jahre, mondjahre: INTEGER;
  set( neu: INTEGER ) is
  do
    jahre:= neu;
  end;

  get: INTEGER is
  do
    Result:=jahre;
  end;
```

```
  umrechnen is
  local h:INTEGER;
  do
    h:=jahre;
    jahre:=mondjahre;
    mondjahre:=h;
  end;
end
```

Speicherverwaltung

- alle ADT sind Referenzen
- Garbage Collector
- Auslagerung von Objekten
- Persistente Objekte
- fast nur Halde
- Klasse MEMORY

Create: allokiert
Forget: verliert Objekt
a:=b : Referenzzuweisung
a=b : Referenzvergleich
Equal: flacher Objektvergleich

```
local
  a, b: REAL;
  s, t, u: HUND;
do
  s.Create("Waldi");
  u.Create("Hasso");
  t:=s -- Waldi gibt's 1 mal.
  s.Forget; -- Waldi existiert noch
  u.Create("Rocky");
     -- Hasso ist Vergangenheit.
  t:=u; -- Waldi auch
     -- Rocky gibt's 1 mal
end
```

Vertragsmetapher

Ziel: sichere Sprache

Es gibt Anbieter und Kunden

Beide müssen Regeln Einhalten

Es gibt:

- Routinenvorbedingungen
- Routinennachbedingungen
- Klasseninvarianten
- Schleifenvarianten
- Schleifeninvarianten
- Checks

Vertragsmetapher

- Wesentlich dabei: Die Bedingungen werden vererbt.
- Eine geerbte Vorbedingung darf nur abgeschwächt und eine geerbte Nachbedingung nur verschärft werden.
- Vor- und Nachbedingung sind beliebig komplexe bool'sche Ausdrücke

Vorbedingung: (**require**)

Sie wird jedesmal vor dem Eintritt in die Routine überprüft. Bei nichterfüllung wird eine Ausnahme generiert.

```
....
divide( z: COMPLEX ) is
require
   z.roh /= 0
do
  -- rechne
ensure
   roh = old roh / z.roh;
   polar
end
```

Nachbedingung: (**ensure**)

Sie wird jedesmal vor dem Eintritt in die Routine überprüft. Bei nichterfüllung wird eine Ausnahme generiert.

```
....
add( z: COMPLEX ) is
do
  -- rechne
ensure
   (x = old x + z.x and
    y = old y + z.y) or
   polar_correct
end
```

Klasseninvarianten und Checks

- Klasseninvarianten sind Bedingungen (bool'sche Ausdrücke, die während der ganzen Lebenszeit eines Objekts nicht verletzt werden dürfen.
- Checks sind Prüfungen, die innerhalb einer Methode angewendet werden.
- Wird eines dieser Konstrukte verletzt gibt es eine Ausnahme.
- Klasseninvarianten akkumulieren sich bei Vererbung (logisch **and**).

```
class STACK
feature
   num_elemente : INTEGER;
invariant
   pos_anzahl: num_elemente >=0;
end
```

```
Metoderl: CHARACTER is
do
   machwas_null;
   check
      wirklich: element = 0
   end
end
```

Schleifen Var- und Invarianten

Auch sie dienen der Sicherheit von Software, aber auch zur Dokumentation und Beweisführung.

```
from  Init-Anweisungen
invariant Invariante
variant Variante
until Abbruchbedingung
loop Schleifenanweisungen
end
```

Schleifen und Verzweigungen

```
class SAGWAS
feature
  Hallo( text :STRING; n:INTEGER ) is
  local i:INTEGER;
  do
    from       i:=5
    invariant  n = old n
    variant    i = 1 + old i
    until      i<=100
    loop
      i:=i+1;
      --irgentwas
    end
  end
end
```

```
class SAGWAS
feature
  Hallo2( n:INTEGER ) is
  do
    if n=2 then
     -- mach dies
    elseif n=6 then
     -- mach jenes
    else
     -- tu das
    end
  end
end
```

Mehrfachvererbung

- Geerbt wird mit **inherit**
- Überschrieben wird mit **redefine**
- Die unweigerlichen Konflikte werden durch Umbenennung gelöst
- Durch Umbenennung kommt man auch wieder an die **Create**s der Eltern
- **Create** wird nie Vererbt

```
class TREE
inherit
   LINKABLE
      rename
         value as node_value,
         right as right_sibling
      redefine
         put_between, right_sibling, left_sibling
   LINKED_LIST
      rename
         first_element as first_child, last_element as last_child
      redefine
         first_child, duplicate
feature
...
```

Konstruktoren

- Konstruktoren werden durch **Create** realisiert. Jede Klasse kann ein eigenes Create realisieren. **Create** darf auch Parameter haben. **Create** hat ferner darauf zu achten, daß die Klasseninvariante erfüllt ist.
- Eine weitere Möglichkeit besteht darin ein Objekt zu Klonen. Mit **Clone** erhält man eine Flachkopie des angegebenen Objekts.

```
class HUND
inherit TIER
redefine sag_was
feature
   Create( t_name : STRING ) is
   do
      name := t_name;
   end

   sag_was : STRING is
   do
      sag_was := "Wuff";
   end
end
```

```
Local
  a, b: REAL;
  s, t, u: HUND;
do
  s.Create("Waldi");
  u.Create("Hasso");
  t:=s -- Waldi gibt's 1 mal.
  s.Forget; -- Waldi existiert noch
  u.Create("Rocky");
     -- Hasso ist Vergangenheit.
  t:=u; -- Waldi auch
     -- Rocky gibt's 1 mal
end
```

Destruktoren

Da Eiffel einen Garbage Collector im Laufzeitsystem hat, ist es an und für sich nicht nötig Destruktoren zu haben. Manchmal gibt es aber doch fälle, in denen es nötig ist einen Destruktor zu haben.

Man hilft sich, indem man von der Klasse MEMORY erbt und die methode Destruct überschreibt.

```
class GUFI
inherit MEMORY
features
  Destruct is
  do ...
  end
end
```

Generische Klassen

Eiffel unterstützt generische Datentypen. Dabei wird vollständig und rekursiv Vererbung unterstützt.

```
class COMPLEX[T]
export add, gibreal
feature
  re, im:T;
  gibreal: T is
  do
    Result:=re;
  end

  add( andere :COMPLEX[T] ) is
  do
     re = andere.re + re;
     im = im + andere.im;
  end
end
```

```
benutzer is
local
  a, b:COMPLEX[REAL];
do
  a.Create( 3.4,5);
  b.Create(1,3);
  b.add(a);
  a.Forget;
end
```

Abstrakte Klassen

- Abstrakte Klassen (interface in Java) werden mit dem Schlüsselwort **defered** deklariert.
- Abstrakte Methoden werden ebenso mit dem Schlüsselwort **defered** deklariert.

```
defered class GEOOBJEKT
  export
     draw
  feature
     draw( x,y :INTEGER ) is
        defered
     end
end
```

Selektive Exporte

Um verkettete Listen oder Iteratoren zu Implementieren, benötigen fremde Klassen Zugriff auf normalerweise verborgene Elemente. Dies kann errecht werden duch:

- Geschachtelte Klassen (java)
- Freund-Klassen (C++)
- Selektive Exporte (eiffel)

```
class LISTE[ADT]
  export
    enter,
    elemente { LISTE_ITR },
    anfang { LISTE_ITR, FREUND },
    nochwas
feature
 .........
end
```

Disziplinierte Ausnahmen

Im Gegensatz zu allgemeiner Ausnahmebehandlung wie sie in C++/Java/ADA implementiert ist.

Die allgemeine Ausnahmebehandlung wird oft missbräuchlich eingesetzt. Daher gibt es in Eiffel nur eine speziellere Form davon, die keine Sprünge sonstwohin erlaubt.

Treten auf:

- Numerischer Überlauf
- Speicherplatzmangel
- Abnormale Fälle, die eine schnellstmögliche Beendigung benötigen
- Geräte E/A
- Verletzungen von Zusicherungen
- Elemenzugriff bei leerer Referenz

```
Pop is
   require
      nb_elements >= 0;
   do
      nb_elements := nb_elements
-1;
      ....
   ensure
      nb_elements = old
nb_elements;
   rescue
      nb_elemenrs := 0;
      .... weiterer Code
end
```

Toleranz bei Ausnahmen

Weiterhin gibt es noch die möglichkeit des Wiederaufsetzens. Innerhalb eines **rescue**-Bloks wird das Objekt wieder in einen definierten Zustand gebracht. Mit **retry** geht es dann wieder weiter.

Man beachte dabei: Ziel ist es den Zustand des Objekts so zu verändern, daß die Routine Erfolgreich beendet. Ist dies nicht der Fall, und endet der **rescue**-Block, so ist das Vorhaben gescheitert.

```
mache_was is
local  versuche: INTEGER;
do
   if versuche = 0 then
      implementierung_1
   elsif versuche = 1 then
      implementierung_2
   end
rescue
   versuche := versuche+1;
   if versuche<2 then
      retry
   end
end
```

Globale Konstanten

Wie realisiert man Konstanten

Konstanten von ADTs

Globale Konstanten

Man braucht keine ststichen Klas-
sen/Objekte

```
class PROGR_MIT_KONST
export gibzeichen
inherit KONSTANTEN
feature
  text :STRING is »irgentwas«;

  gibzeichen: CHARACTER is
  do
    Result:=Proz;
  end
end
```

```
Class KONSTENTEN
feature
  pi: REAL is 3.141592;
  Proz: CHARACTER is '%';
  Null: INTEGER is 0;
  OK: BOOLEAN is false;
  variabel: INTEGER;
  globalobjekt: ARRAY[REAL];

  z: COMPLEX is
  do
    Result.Create(3,7);
  end

  gib_obj: ARRAY[REAL] is
  once
    Result:=globalobjekt;
  end
end
```

Klassen und Systeme

Wie wird aus den Klassen ein Programm

- Pro Datei eine Klasse
- Gleiche Namen für Dateien und Klassen
- SDF-Datei

```
ROOT: name einer Klasse
SOURCE: verzeichnisse mit Klassen
EXTERNAL: externe dateien
NO_ASSERTION_CHECK (Y/N)
DEBUG (Y/N): Klassenlisten
PRECONDITIONS (Y/N)
OPTIMIZE (Y/N)
VIRTUAL_MEMORY (Y/N)
GARBAGE_COLLECT (Y/N)
....
```

Noch einige Quelltexte