§ 6 Programmiersprachen für die Prozessautomatisierung

Kapitel 6 - Lernziele

- Die Vorgehensweisen bei der Erstellung von Prozessautomatisierungs-programmen kennen
- Programmiersprachen nach Sprachhöhe unterscheiden können
- Die Programmiersprachen für SPSen kennen
- Einfache Beispiele in SPS-Sprachen programmieren können
- Die wichtigsten Echtzeit-Konzepte für Programmiersprachen kennen
- Wissen, wie Echtzeit-Konzepte in Ada 95 umgesetzt sind
- Ein Echtzeit-Programm in Ada 95 entwerfen können
- C/C++ bezüglich Echtzeit einschätzen können
- Wissen, worauf die Portabilität von Java beruht
- Die Echtzeit-Erweiterungen von Java verstanden haben

§ 6 Programmiersprachen für die Prozessautomatisierung

Vorgehensweisen bei der Erstellung der Programme

Speicherprogrammierbare Steuerungen
- textuelle Programmiersprachen
- graphische Programmiersprachen

Mikrocontroller
- Assembler
- niedere maschinenunabhängige Programmiersprachen

PC und IPC
- Softwarepakete
- universelle Echtzeitprogrammiersprache

Prozessleitsysteme
- Funktionsbausteintechnologie

Arten von Programmiersprachen

Klassifikation nach Art der Notation

–	textuelle Programmiersprachen	Ada, C, SPS-Anweisungsliste
–	graphische Programmiersprache	SPS-Kontaktplan

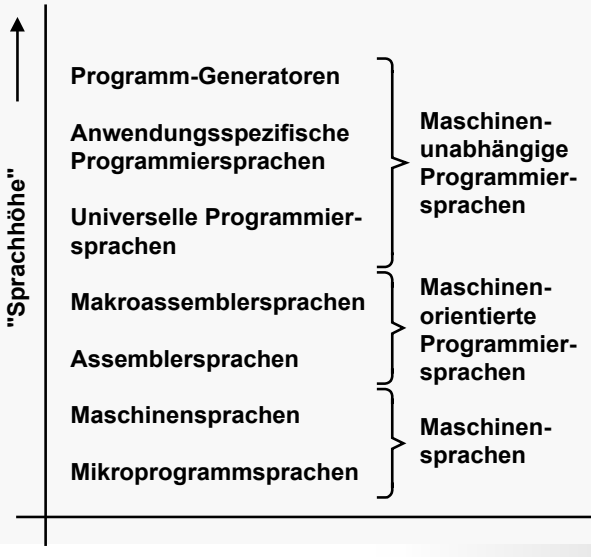
Klassifikation nach dem Programmiersprachenparadigma

–	prozedurale Programmiersprachen	C, Ada 83
–	funktionale Programmiersprachen	LISP
–	logische Programmiersprachen	PROLOG
–	objektorientierte Programmiersprachen	C++, Smalltalk, Ada 95

Klassifikation nach der Sprachhöhe

–	**hoch:**	an der Verstehbarkeit durch den Menschen orientiert
–	**nieder:**	an den Hardware-Eigenschaften eines Computers orientiert

Klassifizierung nach der "Sprachhöhe"



Mikroprogrammsprachen

- Realisierung der Ablaufsteuerungen zur Ausführung der Maschinenbefehle
  - fest verdrahtete Verknüpfungsglieder
  - Mikroprogramme
- Mikroprogramme (Firmware)
  - Speicherung in schnellen Schreib-/Lesespeicher (RAM's) oder in Festwertspeicher (ROM's)
  - nicht zugänglich für Anwenderprogrammierung
- Maschinensprachen
  - Sprachelemente:

    Befehle und Daten in Form von Bitmustern

  - Zusammenfassung als Oktal-bzw. Hexadezimalzahl
  - mühsame Handhabung
  - nicht für Anwendungsprogrammierung

Assemblersprachen

Ziel:
Vermeidung der mühsamen Handhabung der Maschinensprachen unter Beibehaltung der Eigenschaften der Maschinenbefehle

- Ersetzung der oktalen/hexadezimalen Schreibweise des Operationsteils der Befehle durch symbolische, mnemotechnisch günstige Buchstabenabkürzungen
- Einführung eines symbolischen Namens anstelle der zahlenmäßigen Darstellung des Adressteils
- Eindeutige Zuordnung zwischen den Befehlen der Assemblersprache und den Befehlen der Maschinensprache
- Abhängigkeit von gerätetechnischen Eigenschaften der jeweiligen Rechenanlage

Makroassemblersprachen (Makrosprachen)

– weiteres Hilfsmittel zur einfacheren Handhabung: Makros

Makro: Abkürzung für eine bestimmte Befehlsfolge

– Unterscheidung
- Makrodefinition
- Makroaufruf
- Makroexpansion

– Aufbau einer Makrodefinition

MAKRO Makroname (P_1, P_2, ..., P_N)
Makrokörper
Endzeichen
Makroaufruf
Makroname (A_1, A_2, ..., A_N)

– Eindeutige Zuordnung von Makroassemblerbefehlen zu Befehlen der Maschinensprache

– Einem Makrobefehl entsprechen mehrere Maschinenbefehle

Beispiel Makro Dreiadressadd

Makrodefinition:
MAKRO Dreiadressadd (P_1, P_2, P_3)
LADE P_1
ADDIERE P_2
SPEICHERE P_3
ENDE

Makroaufruf:
Dreiadressadd (A,B, SUMME)

Makroexpansion:
...
LADE A
ADDIERE B
SPEICHERE SUMME
...

Unterschied zwischen Unterprogramm-Aufrufen und Makros

- Unterprogramm wird nur einmal gespeichert, kann mehrfach aufgerufen und verwendet werden
- Makro wird an jeder Stelle an der es aufgerufen wird, expandiert

Klassifizierung von Makros

- Standardmakros: fest vorgegeben
- Anwendermakros: vom Anwender selbst definierbare Makros für häufig vorkommende Befehlsfolgen

Universelle Programmiersprachen

universell $\triangleq$ **nicht auf ein Anwendungsgebiet ausgerichtet**

universelle niedere Programmiersprachen — **Systemprogrammier-sprachen**

- Zweck: Erstellung von Systemprogrammen
  - Compiler
  - Editoren
  - Betriebssysteme
  - Treiberprogramme
- Ziel: 1. Ausnutzung der Hardwareeigenschaften
  2. Portabilität
- Beispiel: C

universelle höhere Programmiersprachen

- Zweck: Erstellung von allgemeinen Programmen
- Ziel: 1. einfache Formulierbarkeit
  2. umfangreiche Compilerprüfungen
  3. Portabilität
- Beispiel: Ada, Java, Smalltalk

Anwendungsspezifische Sprachen

Deskripitive Sprachen, nicht-prozedurale höhere Sprachen, very high level languages

Unterscheidung zu prozeduralen Sprachen

– keine Beschreibung des Lösungsverfahrens, sondern Beschreibung der Problemstellung selbst

– Einschränkung auf bestimmtes Anwendungsgebiet

Bsp.:	FUP	Funktionsplan
	KOP	Kontaktplan
	AWL	Anweisungsliste
	EXAPT	für Werkzeugmaschinensteuerungen
	ATLAS	für automatische Prüfsysteme

Vorteile/Nachteile:

+	**bequeme, anwendungsspezifische Ausdrucksweise**
-	**Inflexibilität**

Programm-Generatoren (Fill-in-the-blanks-Sprachen)

– Formulierungsverfahren für Programme

– Beantwortung von Fragen in Form von Menues am Bildschirm durch den Anwender (Konfigurierung)

– Umsetzung der Antworten durch den Programm-Generator in ein ausführbares Programm

– Vorteil: keine Programmierkenntnisse notwendig

– Nachteil:
- **Einengung auf bestimmtes Anwendungsgebiet**
- **Abhängigkeit von einem bestimmten Hersteller**

Anwendungsgebiete von Programm-Generatoren im Bereich Prozessautomatisierung

– Leittechniksysteme in der Energie-und Verfahrenstechnik
  - TELEPERM-M (Siemens)
  - PROCONTROL-B (ABB)
  - CONTRONIC-P (Hartmann & Braun)

– Speicherprogrammierbare Steuerungen in Form von Anweisungsliste oder Kontaktplan
  - SIMATIC (Siemens)

§ 6 Programmiersprachen für die Prozessautomatisierung

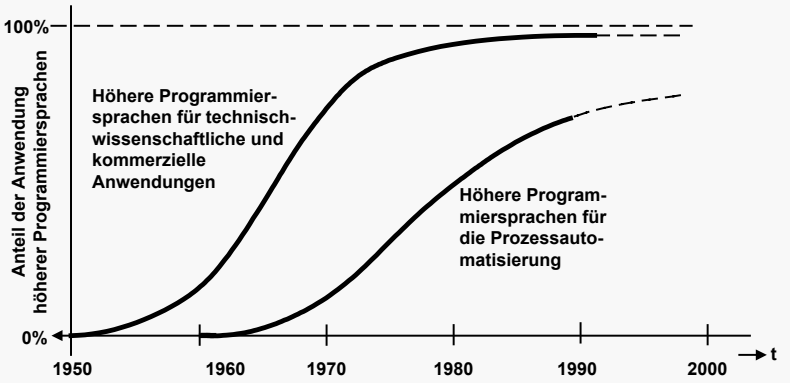
Problematik der Echtzeit-Programmierung

– Höhere Programmiersprachen wie BASIC, FORTRAN, COBOL, PASCAL sind für Echtzeit-Programmierung ungeeignet
  - keine Echtzeitsprachmittel
  - keine Einzelbitoperationen
  - keine Befehle für Prozess-Ein-/Ausgabe

– Einsatz von höheren Programmiersprachen im Vergleich zu Assemblersprachen bringt Erhöhung von Speicherbedarf und Rechenzeit mit sich
  - Produktautomatisierung: Speicherplatz und Rechenzeit kritisch
  - Anlagenautomatisierung: Rechenzeit unter Umständen kritisch

Verfügbarkeit
- Compiler für Zielrechner
- Echtzeit-Betriebssystem

Vergleich der Verwendung höherer Programmiersprachen



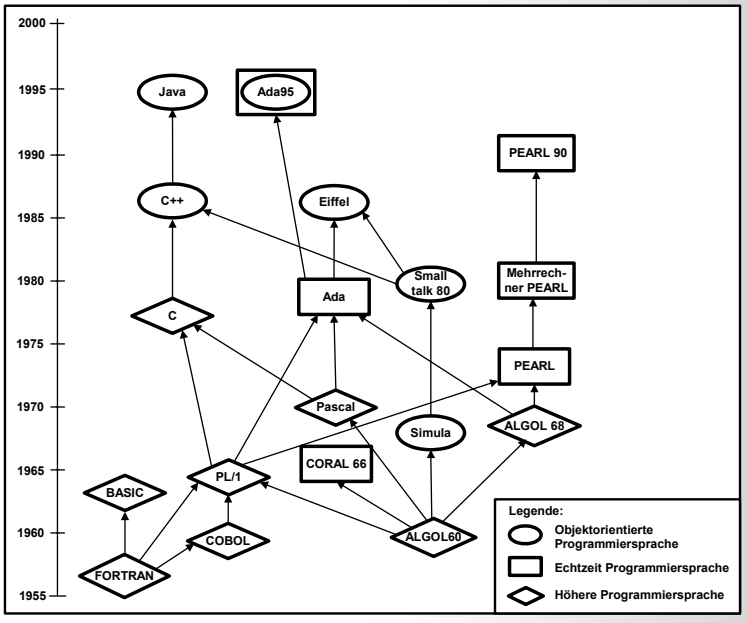
Vor- und Nachteile der Assemblerprogrammierung

+	Speicher- und Rechenzeit-Effizienz
-	höhere Programm-Entwicklungskosten
-	geringe Wartbarkeit
-	Probleme mit der Zuverlässigkeit
-	schlechte Lesbarkeit, geringer Dokumentationswert
-	fehlende Portabilität

Einsatz von Assemblersprachen

ja: Automatisierung von Geräten oder Maschinen, wo es um Serien- oder Massenanwendungen geht, kleine Programme

nein: langlebige, große zu automatisierende Prozesse



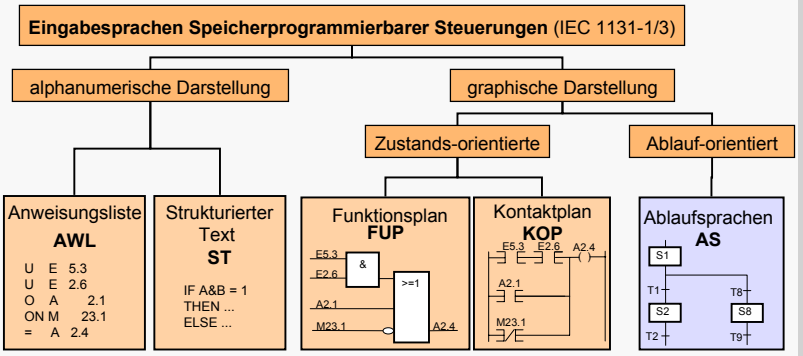
§ 6 Programmiersprachen für die Prozessautomatisierung

Programmiersprachen für SPS-Systeme (1)

– Keine feste Programmiersprache für SPS-Systeme

– Unterschiedliche Arten der Programmierung, auch abhängig vom Hersteller

– IEC 1131 definiert grafische und textuelle Grundsprachen



– IEC 1131 definiert keine Befehle!

⇨ sowohl deutsche als auch englische Bezeichner für Operatoren

Programmiersprachen für SPS-Systeme (2)

– Basis aller Programmiersprachen sind Logikverknüpfungen

– Erweiterung um Möglichkeiten der Zeitverarbeitung

– Darstellungsart je nach Aufgabenstellung unterschiedlich geeignet

- Zustandsorientierte Programmteile besser in FUP oder KOP
- Ablauforientierte Programmteile besser in AWL oder AS

– Darstellungsarten lassen sich ineinander überführen

ABER:

Bestimmte Operationen nur in AWL programmierbar (Bit-Schiebeoperationen)

Einführungsbeispiel:
Ausgang 1(A1) und Ausgang 2 (A2) sind nur dann gesetzt, wenn entweder Eingang 3 (E3) gesetzt ist oder wenn die beiden Eingänge1 (E1) und 2 (E2) gleichzeitig gesetzt sind.

Anweisungsliste (AWL)/Instruction List (IL)

– Assemblerähnliche Programmiersprache

– Alle Funktionen einer SPS uneingeschränkt programmierbar

– Einheitlicher Aufbau einer Befehlszeile

```
Marke(opt.):   Operator  Operand     Kommentar(opt.)
```

– Programmierung durch Verknüpfung von Signalen

Umsetzung des Beispiels in AWL (Tafelanschrieb)

```
Start:      U(    E1
            U     E2
             )
            O     E3
            =     A1
            =     A2
```


Strukturierter Text (ST)

– Höhere, Pascal-ähnliche Sprache
  • Zuweisungen
    z.B.: alarm := ein AND aus;
  • Unterprogrammaufrufe
    z.B.: alarmleuchte(S:=ein, R:=aus);
  • Kontrollanweisungen
    z.B.: IF alarm
    THEN alarmleuchte(S:=ein, R:=aus);
    END_IF;

– Zusätzliche Sprachmittel für Zeitverarbeitung und Prozessdatenzugriff

– Für die Programmierung umfangreicher Systeme geeignet

Umsetzung des Beispiels in ST (Tafelanschrieb)

```
A1 :=    E3 OR (E1 AND E2);
A2 :=    A1;
```


Kontaktplan (KOP) / Ladder Diagramm (LD)

– Einfache Darstellung

– Lehnt sich an Stromlaufplan der Relaistechnik an

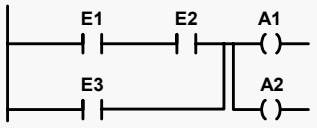
– Symbole für „Schließer“, „Öffner“, und „Relaisspule“

– Symbolisierter Stromfluss von links nach rechts

– I/O-Zustände werden auf Schalterzustände abgebildet

– Programm wird von oben nach unten gelesen

Umsetzung des Beispiels in KOP (Tafelanschrieb)



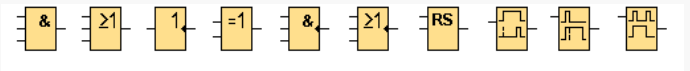
Nachteil: Komplexe mathematische Funktionen schlecht darstellbar

Funktionsbausteinsprache (FBS)/ Funktionsplan (FUP)

– Lehnt sich an bekannte Symbole für Funktionsbausteine (DIN40900) an

– Symbolumfang nicht auf logische Grundelemente beschränkt

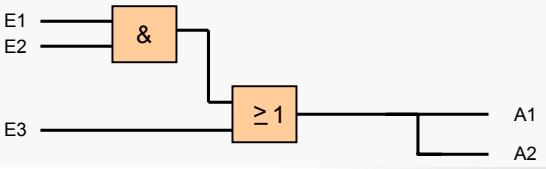
⇨ Merker, Zähler, Zeitgeber und frei definierbare Blöcke möglich



– I/O-Zustände werden direkt als Ein-/Ausgangssignale verwendet

– Übersichtliche Darstellung

Umsetzung des Beispiels in FUP (Tafelanschrieb)



§ 6 Programmiersprachen für die Prozessautomatisierung

Entstehungsgeschichte von Ada

Ada
Name der 1. Programmiererin
Zu Ehren der Mathematikerin Augusta Ada Byron, Countess of Lovelace, Tochter von Lord Byron benannt. Ada Lovelace (1815- 1851) arbeitete mit Charles Babbage an seiner “Difference-and Analytic-Engine”. Auf Ada Lovelace geht die Idee zurück, diese Maschine mit Lochkarten zu programmieren.

Entwicklung von Ada

1975	Gründung von HOLWG zur Entwicklung einer Programmiersprache für das DOD (USA) für “embedded systems”
1980	Festschreibung der Sprachdefinition, Ada 80
1983	ANSI Ada Standard, Ada 83
1987	ISO-Norm für Ada
1988	DIN-Norm (DIN 66268)
1995	Festschreibung von Ada9X zu Ada95, Spracherweiterungen zur Unterstützung der objektorientierten Programmierung

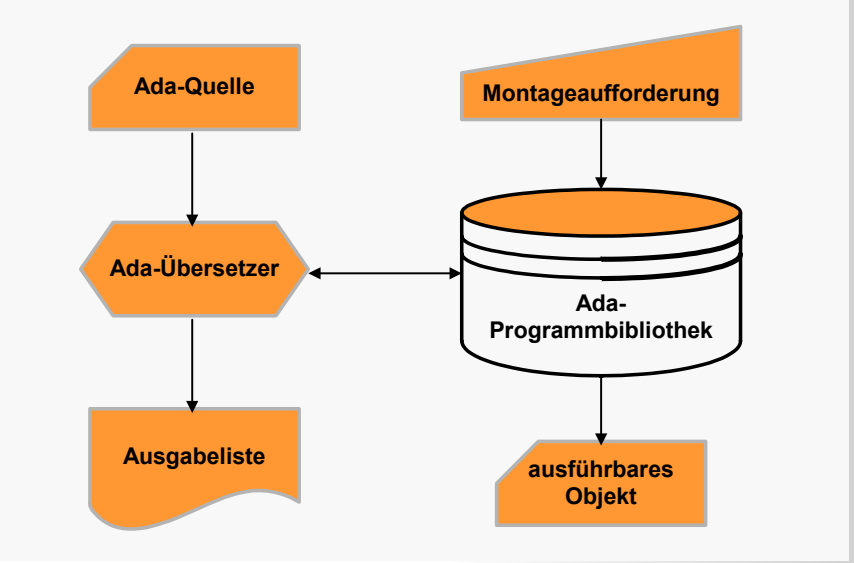
Eigenschaften von Ada

- Eignung für sehr umfangreiche Softwaresysteme bis über 10^7 Zeilen
- Modularisierungskonzept, das eine koordinierte, parallele Entwicklung ermöglicht
- “State of the art”-Sprachkonzepte zur Unterstützung von Überprüfungen zur Übersetzungszeit
- Wiederverwendbarkeit von Softwarekomponenten
- Sichere, moderne und effiziente Echtzeit-Konzepte
- Durchgängiges Fehlerbehandlungskonzept mit Laufzeitüberprüfungen
- Vereinbarkeit der Portierbarkeit mit Maschinennähe
- Internationale Normung von Syntax und Semantik
- Prüfung jedes Ada-Compilers in einer umfangreichen Validierungsprozedur auf die Einhaltung der Norm (mehrere tausend Testprogramme)

Programmstruktur

- Übersetzbare Einheiten
  - Unterprogramme
    - Funktionen
    - Prozeduren
  - Pakete (Module)
- Aufbau von Übersetzungseinheiten
  - Vereinbarungen
  - Anweisungen
  - Pragmas zur Steuerung des Übersetzungsvorgangs

Bibliotheksbezogene Programmgenerierung



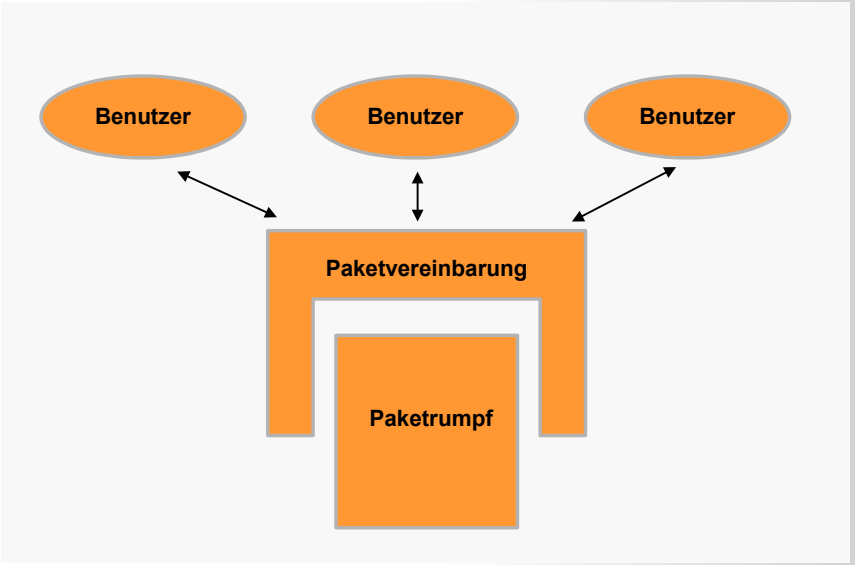
Programmeinheiten

- Pakete (Module)
- Tasks (Rechenprozesse)
- protected units (Monitore)
- generische Einheiten
  (parametrisierte Module und Unterprogramme)

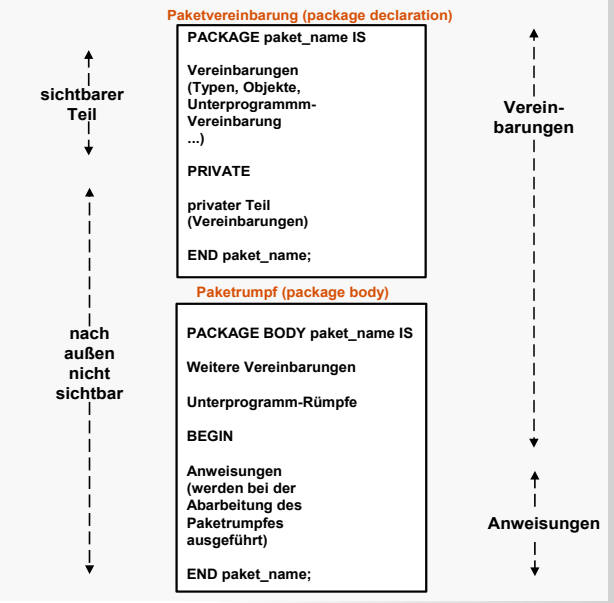
Aufbau von Programmeinheiten

- **Spezifikation:** **Schnittstelle nach außen**
- **Rumpf:** **Realisierung der Programmeinheit**

Paketvereinbarung, Paketrumpf und ihre Benutzer



Aufbau eines Pakets



Beispiel eines Ada-Programms

```
PACKAGE Stack IS
      PROCEDURE    Push (X: REAL);
      FUNCTION     Pop RETURN REAL;
END;
PACKAGE BODY stack IS
      Max:  CONSTANT:=100;
      S:    ARRAY (1..Max) OF REAL;
      Ptr:  INTEGER RANGE 0..Max;
      PROCEDURE    Push (X:REAL)IS
            BEGIN
                  Ptr:= Ptr + 1
                  S (Ptr):= X;
            END Push;
      FUNCTION Pop RETURN REAL IS
            BEGIN
                  Ptr:= Ptr - 1
                  RETURN S (Ptr + 1);
            END pop;
      BEGIN                                  Aufruf  Stack.Push (Y):
                  Ptr:= o                            ...
      END Stack;                                     A:=     Stack.Pop ();
```


Tasks

- Tasks sind keine Übersetzungseinheiten
- Ablauf autonom und nur lose an den Ablauf anderer Ablaufeinheiten gebunden
- Vereinbarung erfolgt im Vereinbarungsteil eines Blocks, eines Unterprogramms, eines Paketrumpfs oder einer anderen Task

Beispiel:

```
PROCEDURE Arrive_at_Airport is
      TASK Rent_a_Car;
      TASK BODY Rent_a_Car is
      ...
      END;
BEGIN
      Book_Hotel;
END   Arrive_at_Airport
```


Algorithmik

Für die algorithmische Programmierung stehen unter Ada 95 die gängigen Sprachkonstrukte zur Verfügung. Dazu gehören:

– Vereinbarungen (z.B. PI: CONSTANT:=3.141_159)

– Datentypen (z.B. INTEGER, REAL, STRING ...)

– Typedefinitionen (z.B. ARRAY, RECORD, ACCESS...)

– Selektion (z.B. IF, CASE...)

– Schleifen und Sprunganweisungen (z.B. LOOP, FOR, EXIT, GOTO...)

Ein-/ Ausgabe

– nicht explizit definiert
– Paket mit Standard E/ A-Funktionen wird bereitgestellt
– Bsp.: Get, Put_line, Page, New_Line

Ausnahmesituationen

– vordefinierte Ausnahmesituationen
  - Constraint-Error
    Bsp.: Division durch Null
  - RAISE Constraint-Error

– Behandlung von Ausnahmesituationen
  EXCEPTION
    When Constraint-Error => Ueberlauf:= TRUE

Sprachkonstrukte für die Echtzeitprogrammierung

- Tasks
  Aktivierung durch Eintritt in die Umgebung
- Verzögerungsanweisung
  DELAY 3.0;
- Taskabbruch
  ABORT
- Synchronisierung
  - Rendezvous-Konzept
    ENTRY
    ACCEPT auf Empfängerseite
  - selektive Synchronisierung
    SELECT
    Eingangsaufruf
    OR
    Eingangsaufruf
  - zeitbedingtes Warten
    SELECT
    ...
    OR DELAY

Rendezvous-Konzept (1)

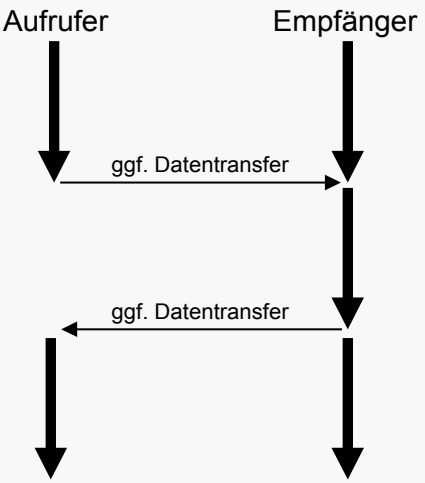
Handshake-Verfahren

- Beide Partner müssen warten
- Aus der Sicht des Aufrufers hat das Rendezvous die Gestalt eines Prozeduraufrufs

Aufrufvereinbarung
ENTRY

Aufrufpunkt
ACCEPT

Rendezvous-Konzept (2)



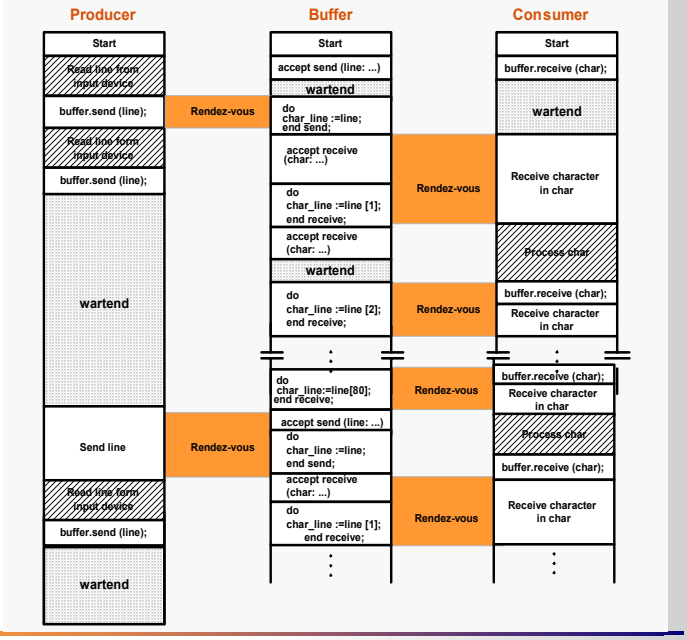
Erzeuger-Verbraucher-Problem (1)

```
task buffer is
      type line_bf is array (1..80) of character;
      entry send (line: in line-bf);
      entry receice (char: out character);
end buffer;
  task body buffer is
      char_line: line_bf;
      position : integer;
begin
      loop
            accept send (line :in line_bf) do
              char_line := line;
            end send;
            for position in 1..80 loop
              accept receive (char: out character) do
                 char := char_line [position];
              end receive;
            end loop;
      end loop;
end buffer;
```

Erzeuger-Verbraucher-Problem (2)

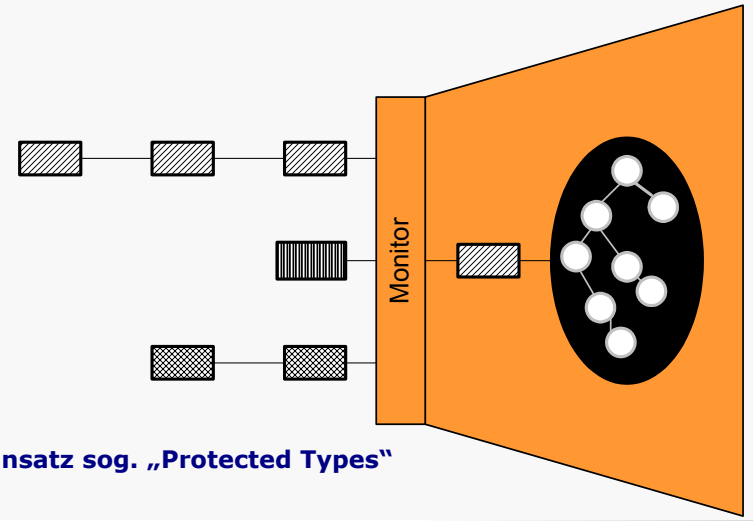
```
task producer;
task body producer is
       use buffer;
       input_line : line_bf;
begin
       loop
         ... (* Eine Zeile vom Eingabegerät in input_line lesen *)
          buffer.send(inout_line);
       end loop;
end producer;

task consumer;
task body consumer is
       use buffer;
       input_char : character;
begin
       loop
         buffer.receive(input_char);
         ... (* Das eingelesene Zeichen bearbeiten *)
       end loop;
end consumer;
```


Möglicher Ablauf eines Mehrtask-programms



Geschützte Typen

Monitor zur Sequenzialisierung von Zugriffen



Einsatz sog. „Protected Types"

Beispiel

```
PACKAGE Fifo_Puffer_Verwaltung IS
        PROTECTED TYPE Fifo_Puffer IS
                ENTRY  Deponiere (N:IN Info_Type);
                ENTRY  Gib_Frei (N:OUT Info_Type);
        END Fifo_Puffer;
END Fifo_Puffer_Verwaltung
```


Spracherweiterungen in Ada 95

Objektorientierte Programmierung

- Deklaration neuer Typen, die ähnliche Eigenschaften haben wie der Elterntyp, Klassen, Objekte
- Vererbung, Modifikation, Hinzufügen von Komponenten der Datenstruktur bzw. der darauf definierten Operationen


```
type Geldtyp is ...;
type Girokonto is tagged
        record
            Inhaber: String( 1...40);
            Nummer: Positive;
            Stand: Geldtyp;
        end record;
```

```
type Firmenkonto is new
Girokonto with record
        Branche: String(1..10);
        Eigenkapital: Geldtyp;
        end record;
```

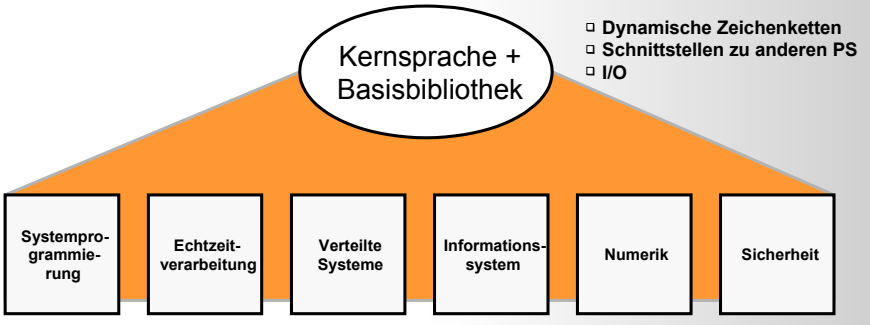
```
type Privatkonto is new
Girokonto
        with record Dispolimit:
        Geldtyp; end record;
```

```
type Mitarbeiterkonto is new
Firmenkonto with record
        Abteilung: String (1...5);
        Position: String (1... 10);
        end record;
```


Erweiterungen für die Echtzeitverarbeitung

- Dynamische Prioritäten
- Beeinflussung der Bearbeitung von Warteschlangen
- Beeinflussung des Abbruchverhaltens von Tasks
- Veränderungen des Taskzustandsmodells
- Festlegung des Zeitbegriffs und Beeinflussung der Zeitgebung

Spracherweiterungen in Ada 95



§ 6 Programmiersprachen für die Prozessautomatisierung

Entstehungsgeschichte von C und C++

1978 Entwicklung des Betriebssystems UNIX durch Dennis Richie in der Programmiersprache C

1986 Erweiterung von C für die objektorientierte Programmierung zur Programmiersprache C++

Entwicklungsziele von C und C++

C:
- Maschinennahe effiziente Systemprogrammiersprache
- flexibel wie Assembler
- Kontrollflussmöglichkeiten höherer Programmiersprachen
- universelle Verwendbarkeit
- begrenzter Sprachumfang

C++:
- Erweiterung von C um Objektorientierung
- Beibehaltung der Effizienz von C
- Verbesserung der Produktivität und Qualität

Hybride Programmiersprache

Concurrent C: Erweiterung von C um Konzepte zur Echtzeitverarbeitung

Sprachkonzepte von C

–	Vier Datentypen:	char, int, float, double
–	Zusammenfassung von Daten:	Vektoren, Strukturen
–	Kontrollstrukturen:	if, switch, while, do-while, for, continue, break, exit, goto
–	Ein-/Ausgabe:	Bibliotheksfunktionen
–	große Vielfalt an Bit-Manipulationsmöglichkeiten	
–	schwaches Typkonzept	
–	getrennte Kompilierbarkeit von Sourcefiles	
–	schwaches Exception-Handling	
–	keine expliziten Möglichkeiten für Parallelverarbeitung	

Programmaufbau

Einfügen bestimmter Bibliotheken bzw. Dateien
Festlegen der Namen für Konstanten und Makros

Deklaration von globalen Variablen;

```
main ( )
{
        Variablen, die innerhalb der Funktion main bekannt sind,
        müssen deklariert werden;
        Anweisungen;
}
```

```
Funktionstype Funktionsname (Liste der Parameter)
{
        Variablen, die innerhalb der Funktion bekannt sind, müssen hier
        deklariert werden;
        verschiedene Anweisungen;
        return (Wert);
}
```

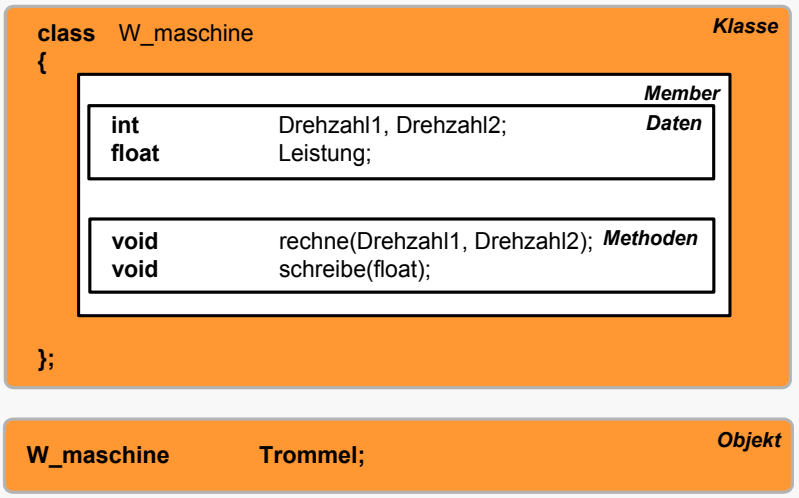

Beispiel

```
#include <stdio.h>                    // Standard-Ein-/Ausgabebibliothek

main ( )                              // Kennzeichnung des Programm-
{                                     // anfangs

        printf („Mein erstes Programm/n“)        // Ausgabeanweisung

}
```

Sprachkonzepte von C++

- Klasse (class)
  - Datenstruktur mit Daten und Methoden (memberfunction)
- Konstruktor (constructor)
  - Anlegen einer Instanz einer Klasse (Objekt)
- Destruktor (destructor)
  - Freigabe von Klassenobjekten
- Überladen von Funktionen (Overloading)
- Datenkapselung (encapsulation)
- Vererbung (inheritance)
- Polymorphismus
  - Auslösung unterschiedlicher Verarbeitungsschritte durch Botschaften

Struktur eines C++-Programms



Eignung von C und C++ für die Echtzeitprogrammierung (1)

– C und C++ enthalten keine Echtzeit-Sprachmittel

– Einsatz von Echtzeit-Betriebssystemen zur Realisierung von Echtzeitsystemen

Aufruf von Betriebssystemfunktionen im C-Programm

– Bereitstellung von Bibliotheken

Eignung von C und C++ für die Echtzeitprogrammierung (2)

Programmiersprachen C und C++

⇨ Am häufigsten verwendete Programmiersprache für Echtzeit-Anwendungen

- große Anzahl und Vielfalt an Unterstützungswerkzeugen
- gut ausgebaute Programmierumgebungen
- für die meisten Mikroprozessoren sind Compiler verfügbar
- Anschluss an Echtzeitbetriebssysteme wie QNX, OS9, RTS, VxWorks

⇨ **Vorsicht bei objektorientierten Sprachmitteln**
- **nicht-deterministisches Laufzeitverhalten**
- **schlechte Speicherplatzausnutzung**

§ 6 Programmiersprachen für die Prozessautomatisierung

Entstehungsgeschichte von Java

1990 Konzept der Programmiersprache Java durch die Fa. Sun (James Gosling, Bill Joy)

Ziel: Programmiersprache für die Unterhaltungselektronik (interaktives Fernsehen)

Namensgebung nach der Kaffeesorte Java

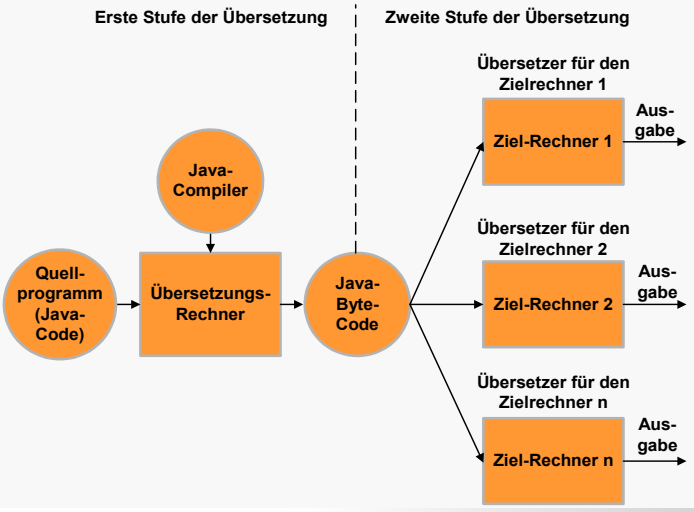
1995 Neuorientierung der Entwicklungsrichtung zu einer Sprache, um Programme im World Wide Web zu übertragen und auszuführen

Sprachkonzepte von Java

- Objektorientierte Konzepte
- Interpretierung des Codes
  - schneller Entwicklungszyklus
  - schlechtes Laufzeitverhalten und hoher Speicherplatzbedarf
  - höhere Portabilität
- Bereitstellung eines Speichermanagers
- Verzicht auf herkömmliche Zeigertechnik
- Strikte Typprüfung zur Kompilier- und Laufzeit
- Leichtgewichtsprozesse
- GUI-Klassenbibliothek

Portabilität

Die Portabilität von Java wird durch zweistufiges Übersetzungsverfahren erreicht.



Unterschiede zu C++

- Keine Preprozessoranweisungen wie #define oder #include
- keine typedef-Klauseln
- Structures und Unions in Form von Klassen
- keine Funktionen
- keine Mehrfachvererbung
- kein goto
- kein Überladen von Operatoren
- umfangreiche Klassenbibliothek
  - Basisklassen (Object, Float, Integer)
  - GUI-Klassen
  - Klassen für Ein-und Ausgabe
  - Klassen für Netzwerkunterstützung

Eignung von Java für die Entwicklung von Echtzeitsystemen

Anwendungsfelder

- rapid prototyping im Client/Server-Bereich
- multimediale Darstellungen (Video, Sound, Animation)
- Intranetapplikationen
- Echtzeitanwendungen

⇨ **Speicherverwaltung (garbage collection)**

⇨ **hoher Speicherbedarf**

⇨ **schlechtes Laufzeitverhalten**

Echtzeitsprachmittel in Java

- Eingabe und Ausgabe von Prozesswerten
  - vergleichbar mit C/ C++
- Parallelität
  - keine Prozessunterstützung
  - Leichtgewichtsprozesse
  - Zeitscheibenverfahren
- Synchronisierung
  - Monitore
  - Semaphorvariablen
- Interprozesskommunikation
  - nur für Leichtgewichtsprozesse über gemeinsame Daten
- Bitoperationen
  - vergleichbar mit C/ C++

Java-Beispiele

Java Applikation

```
class HelloWorldApplication
{
        public static void main(String argv[ ])
        {
                System.out.println(„Hello World!“);
        }
}
```

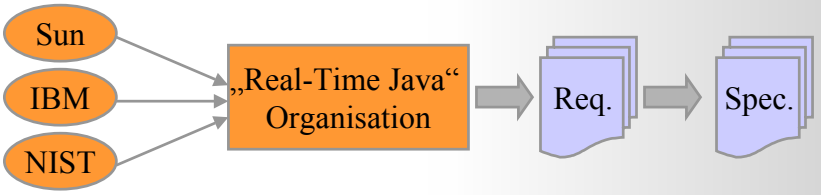
Java Applet

```
import java.applet.*;
import java.awt.Graphics;
public class HelloWorldApplet extends Applet
{
        public void paint(Graphics g)
        {
                g.drawString(„Hello World!“),5,25);
        }
}
```


Entwicklungsgeschichte

Java ist „von Haus aus“ nicht Echtzeit-fähig, dies wird aber vielfach benötigt.

➜ Notwendigkeit einer Spracherweiterung



– parallele und unkoordinierte Anstrengungen Java Echtzeit-fähig zu machen

– Gründung der „Real-Time Java“ Organisation (1998)

– Arbeitsgruppe veröffentlicht die Anforderungen für Real-Time Java (1999)

– Verabschiedung der Echtzeit-Spezifikation für Java (RTSJ) durch den „*Java Community Process*“

Real-Time Specification for Java (RTSJ)

Die RTSJ beinhaltet Spracherweiterungen sowie Erweiterungen der Java Virtual Machine (JVM) um Java Threads erzeugen zu können, deren Korrektheits-Bedingungen auch Rechtzeitigkeits-Vorgaben enthalten.

Der Schwerpunkt der Erweiterungen liegt dabei im Bereich

– des Speicherzugriff und der Speicherverwaltung

– der Behandlung von asynchronen Ereignissen

Insgesamt gibt es 8 Problembereiche mit erweiterter Sprachsemantik

Erweiterungen (1)

- **1. Scheduling**
  Sicherstellung, dass Sequenzen von eingeplanten Objekten rechtzeitig und berechenbar ausgeführt werden
- **2. Speichermanagement**
  Erweiterung des Speichermodells um Echtzeitanwendungen deterministisches Verhalten zu ermöglichen
- **3. Synchronisation**
  Spezifikation der Zuteilungsalgorithmen mit Unterstützung der „priority inheritance“ und „priority ceiling“-Methoden und unter Vermeidung des Problems der Prioritäteninversion
- **4. Asynchrone Ereignis-Behandlung**
  Gewährleistung, dass das Programm mit einer großen Anzahl gleichzeitiger Ereignisse (bis zu mehreren 10000) umgehen kann.

Erweiterungen (2)

- **5. Asynchrone Ausführungsunterbrechung (ATC)**
  Möglichkeit der Ausführungsunterbrechung eines Threads bei Eintreffen eines asynchronen Ereignisses (z.B. Timerablauf)
- **6. Asynchrone Thread-Terminierung**
  Gewährleistung einer planmäßigen Terminierung und Speicherfreigabe von Threads ohne Deadlockgefahr
- **7. Physikalischer Speicherzugriff**
  Spezielle API (Application Programming Interface) für einen direkten Speicherzugriff
- **8. Exceptions**
  Definition von neuen Exceptions (Ausnahmen) und neuer Behandlung von Exceptions im Zusammenhang mit ATC und Speicherreservierung

Frage zu Kapitel 6.2

Für die Steuerung der Zentralverriegelung in einem Kfz soll ein Steuergerät mit einem Mikrocontroller eingesetzt werden.

Erläutern Sie, warum es vorteilhaft sein kann, die Software in einer Assemblersprache zu erstellen.

Antwort

Der erzeugte Code aus einem Assemblerprogramm ist effizienter und benötigt weniger Speicherplatz.

Dadurch können die Kosten für die Serienproduktion (eventuell) geringer ausfallen, da ein billigerer Mikrocontroller und weniger Speicher benötigt werden.

Bei großen Stückzahlen fällt dies wesentlich mehr ins Gewicht als die höheren Entwicklungskosten, die durch die Programmierung in einer Assemblersprache anfallen.

Frage zu Kapitel 6.4

Nennen Sie einige wesentliche Unterschiede zwischen der Echtzeit-programmiersprache Ada 95 und der SPS-Sprache FBS in Hinblick auf

Antwort

	Ada	FBS
Notation	textuelle Sprache	grafische Sprache
Sprachhöhe	Universelle höhere Sprache	Sprache vor allem für Steuerungen
Einsetzbarkeit	Geeignet für große Projekte	Geeignet für kleine Projekte
Echtzeiteigenschaften	Umfassende Echtzeit-Spracheigenschaften	Nur eingeschränkte Spracheigenschaften für die Echtzeitprogrammierung

Frage zu Kapitel 6.4

Ada 95 wird auf Grund einiger seiner Spracheigenschaften als Echtzeit-Progammiersprache angesehen. Warum ist Ada besonders Echtzeit-tauglich? Da.....

Antwort

- ☐ Ada keine Objekt-Orientierung besitzt.
- ☑ Ada Rechenprozesse unterstützt.
- ☐ Ada schneller wie andere Programmiersprachen ist.
- ☐ Ada interpretiert wird.
- ☐ Ada eine hybride Programmiersprache ist.
- ☑ Ada ein Rendevous-Konzept bietet.
- ☑ Ada über Methoden für Laufzeitüberprüfungen und eine Ausnahmebehandlung verfügt.

Frage zu Kapitel 6.4

Folgender Ablauf soll in Ada in Form zweier Tasks implementiert werden:

Ein Passant wartet auf ein Taxi und fährt mit diesem zur Kirche. Danach bezahlt er die Fahrt und geht seines Weges.

Nachfolgend finden Sie 3 Codeauszüge für dieses Problem. Welcher ist der Richtige und was machen die anderen?

Frage zu Kapitel 6.4 - Codeauszüge

Variante 1

```
task passant;

task body passant is
begin
     -- do something
     taxi.drive;
     -- do something
end passant;


task taxi is
     entry drive;
end taxi;

task body taxi is
begin
     -- do something
     accept drive;
     drive.to(church);
     -- drive away
end taxi;
```

Variante 2

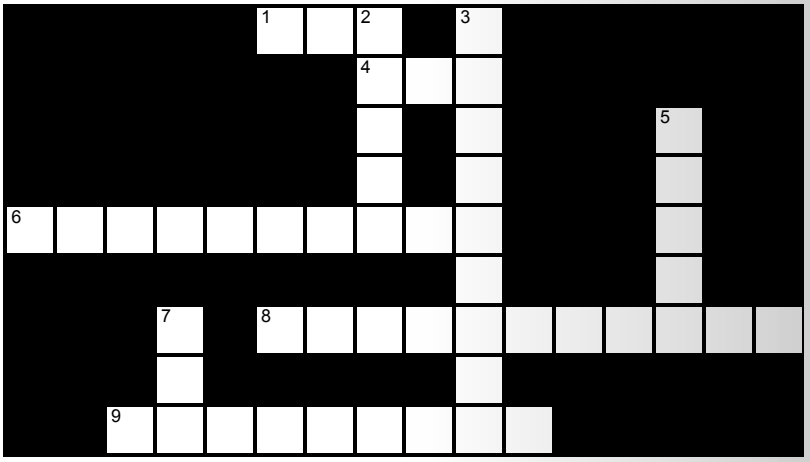
```
task passant;

task body passant is
begin
     -- do something
     taxi.drive;
     -- do something
end passant;

task taxi is
      entry drive;
end taxi;

task body taxi is
begin
     -- do something
     accept drive do
      drive.to(church);
     end drive;
    -- drive away
end taxi;
```

Variante 3

```
task passant is
     entry taxi;
end passant;

task body passant is
begin
     -- do something
     accept taxi;
     -- do something
end passant;

task taxi is
      entry passant;
end taxi;

task body taxi is
begin
     -- do something
     accept passant;
     drive.to(church);
     -- drive away
end taxi;
```

Variante 2 ist die richtige Umsetzung für das beschriebene Problem!

Kreuzworträtsel zu Kapitel 6



Kreuzworträtsel zu Kapitel 6

Waagerecht

1	Grundlage für die Plattformunabhängigkeit von Java (3)
4	Echtzeit-Programmiersprache (3)
6	Ada-Konzept zur Synchronisierung (10)
8	Schaltplan-ähnliche Darstellung für SPS-Programme (11)
9	Maschinennahe Programmiersprache (9)

Senkrecht

2	Hilfsmittel zur Vereinfachung der Maschinensprache (5)
3	Engl. Bezeichnung für eine gemeinsame Ausführung eines Programmstücks auf Empfängerseite (9)
5	Paket in Ada (5)
7	Abkürzung für eine Blockschaltbild-orientierte SPS-Programmiersprache (3)