VORLESUNG

Automatisierung industrieller Workflows

Teil C: Die Sprache SLX

- Einführung und Grundkonzepte -

Joachim Fischer

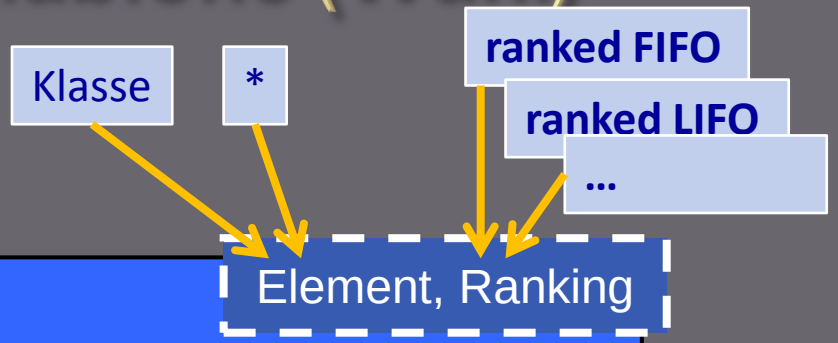
Position

- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Moc
- **Teil C**
  Die ausf
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Typ
7. Set
8. Anweisungen, Prozeduren
9. Einfache Ausgabe
10. Modellierungselemente in SLX (allgemein)
11. Prozesse und Pucks
12. SLX- Laufzeitsystem (Simulator-Kern)
13. Zeitkonzepte
14. Behandlung von Zeit- und Zustandsereignissen

Set als SLX-Typschablone (Wdh.)



Set	
control int size	Erklärung später
place (...)	*einfügen einer Element-Instanz in Set evtl. an eine bestimmte Position*
remove (...)	*entnehmen einer Element-Instanz*
size (...): **int**	*aktuelle Elementanzahl von Set*
position (...): **pointer** (Element)	*Elementinstanz auf einer bestimmten Position von Set*
for_Iterator (...)	*Anwendung einer Folge von Anweisungen für jedes oder ausgewählte Elemente von Set*
first (...)	*erstes Element*
last (...)	*letztes Element*
succuessor (...): **pointer** (Element)	*Nachfolger-Element*
Predecessor (...): **pointer** (Element)	*Vorgänger-Element*
contains (...): **boolean**	*Test, ob Set ein bestimmtes Element enthält*
is_in (...): **boolean**	*Test, ob sich ein Element in einer bestimmten Menge befindet*
is_not_in (...): **boolean**	*Test, ob sich ein Element in einer bestimmten Menge befindet*
empty (...)	*set komplett leeren*
report	Erklärung später
clear	
fin	

Achtung: Operationen haben komplexe Signaturen und werden so nicht angeboten. Stattdessen gibt es flexible Schlüsselwort-gesteuerte Anweisungen

Set-Operationen als Anweisungen (1)

- Einfügen
  place <Pointer> **into** <Set>
- Einfügen an Position
  place <Pointer> **into** <Set> **after** <Pointer>
- Position ermitteln
  int <Position> = **position**(<Pointer>) **in** <Set>
- Entfernen
  remove <Pointer> **from** <Set>
- Leeren
  empty <Set>
- Iterierte Bearbeitung von
  - ausschließlich Objekte einer bestimmten Klasse/Ableitung:
    for (<Pointer> = **each** <Klasse> **in** <Set>) {...}
  - allen Objekten vom Set-Objekt:
    for (<Pointer> = **each object in** <Set>) {...}

Set-Operationen als Anweisungen (2)

- Speziell positionierte Elemente
  - erstes/letztes Objekt einer best. Klasse
    - **first** <Klasse> **in** <Set>
    - **last** <Klasse> **in** <Set>
  - einer beliebigen Klasse
    - **first object in** <Set>
    - **last object in** <Set>
  - Nachfolger/Vorgänger eines Objektes beliebiger Klasse
    - **successor**(<Pointer>) **in** <Set>
    - **predecessor**(<Pointer>) **in** <Set>
  - bestimmter Klasse
    - <Klasse> **successor**(<Pointer>) **in** <Set>
    - <Klasse> **predecessor**(<Pointer>) **in** <Set>

Set-Operationen als Operatoren

- Enthalten/Nicht enthalten: Boolean
  - **boolean** b= <Set> **contains** <Pointer>
  - **boolean** b= <Pointer> **is_in** <Set>
  - **boolean** b= <Pointer> i**s_not_in** <Set>

} äquivalent

  - Kardinalität
  - <Set>. **size**

Achtung: **size** ist eine Control-Variable

```
//*****************************************************************************
//        EX-0001-Set in Anwendung
//*****************************************************************************
public module HafenElemente {

type SchiffsTyp enum {
        fracht, passagier, yacht
};

type ListenTyp enum {
        fracht, passagier, yacht, regist
};

passive class Schiff (SchiffsTyp t, double l, int b) {
        int             meineLadekapazität;
        SchiffsTyp      meinTyp;
        int             meineBesatzung;
        initial {
                meineLadekapazität    = l;
                meinTyp               = t;
                meineBesatzung        =b;
        }
        procedure beschreibung {
                print (meinTyp, meineLadekapazität, meineBesatzung) "Typ= _, Kap= _, Bes= _\n";
        }
} // class Schiff

passive class Hafen {
        set(Schiff) ranked FIFO registration ;
        set(Schiff) ranked (descending meineBesatzung) frachtHafen;
        set(Schiff) passagierHafen;
        set(Schiff) yachtHafen;

        procedure setReport (set(Schiff) s, ListenTyp t) {
                int i;
                pointer(Schiff) schiff;

                print (t) "\n    Inhalt von Liste <_>:  \n";
                if (s.size ==0) { print "    empty\n";}
                for (i=1; i<=s.size; i++) {
                        schiff= position(i) in s;
                        print(i) "    _: ";
                        schiff->beschreibung();
                }
        } //setReport

        procedure meinReport() {
                ListenTyp t;
                print options=bold  "HAFEN REPORT\n";
                setReport(registration, regist);
                setReport(frachtHafen, fracht);
                setReport(passagierHafen, passagier);
                setReport(yachtHafen, yacht);
        }
} //class Hafen

procedure schiffsAnkunft (pointer(Hafen) h, SchiffsTyp t, int anzahl) {
        pointer(Schiff) schiff;
        int i;
        int x, y;
        int anz;
        if (anzahl<1) { anz=1;} else anz= anzahl;
        print (anz, t) "\n+++ Ankunft von _ <_> -Schiffen\n";
        switch (t) {
                case fracht: {
                        for (i=1; i<=anz;  i++) {
                                schiff= new Schiff(t, 10000+x, 10+y);
                                x+=100; y+=1;
                                place schiff into h->registration;
                                place schiff into h->frachtHafen;
                        }
                }
        }
} // schiffsAnkunft

} // module HafenElemente
```

```
module Anwendung {

procedure main() {
        pointer (Hafen) meinHafen;
        pointer (Schiff) einSchiff;
        pointer (Schiff) nocheinSchiff;

        meinHafen= new Hafen();
        meinHafen->meinReport();
        schiffsAnkunft(meinHafen, fracht, 3);
        meinHafen->meinReport();
/*
        einSchiff= position (2) in meinHafen->registration;
        place einSchiff into meinHafen->frachtHafen;

        ** Execution warning at time 0: Object is already in this set
*/

/*
        nocheinSchiff=new Schiff (fracht, 1000, 25);
        place nocheinSchiff into meinHafen->frachtHafen before einSchiff;
        ** Semantic error: "meinHafen" is a ranked set; its ranking cannot be opverridden
*/
        empty meinHafen->registration;
        meinHafen->meinReport();
}

} //module Anwendung
```

```
EX-0000-set-application.slx: SLX-64 CT103  Lines: 1,857  Errors: 0  Warnings: 0  L
Execution begins
HAFEN REPORT

   Inhalt von Liste <regist>:
   empty

   Inhalt von Liste <fracht>:
   empty

   Inhalt von Liste <passagier>:
   empty

   Inhalt von Liste <yacht>:
   empty

+++ Ankunft von 3  <fracht > -Schiffen
HAFEN REPORT

   Inhalt von Liste <regist>:
   1. Typ= fracht , Kap=  10000 , Bes=  10
   2. Typ= fracht , Kap=  10100 , Bes=  11
   3. Typ= fracht , Kap=  10200 , Bes=  12

   Inhalt von Liste <fracht>:
   1. Typ= fracht , Kap=  10200 , Bes=  12
   2. Typ= fracht , Kap=  10100 , Bes=  11
   3. Typ= fracht , Kap=  10000 , Bes=  10

   Inhalt von Liste <passagier>:
   empty

   Inhalt von Liste <yacht>:
   empty
HAFEN REPORT

   Inhalt von Liste <regist>:
   empty

   Inhalt von Liste <fracht>:
   1. Typ= fracht , Kap=  10200 , Bes=  12
   2. Typ= fracht , Kap=  10100 , Bes=  11
   3. Typ= fracht , Kap=  10000 , Bes=  10

   Inhalt von Liste <passagier>:
   empty

   Inhalt von Liste <yacht>:
   empty
Execution complete
Objects created: 15 passive and 1 active  Pucks created: 2  Memory: 2 MB  Time: 0
```

Set-Operationen als Anweisungen (3)

- for-Each-Anweisung
  - *Iteration über die Elemente einer Menge*
- Syntax:

```
for (<Pointer> = each {<Klasse> | object}
        in [reverse] <Set>
        [{before | after | from} <Pointer_in_Set>]
        [with <Boolean_Expression>]
    ) { ... }
```

```
set(*) s;
...
pointer(*) p;
```

```
for (p = each object in s with p->i > 10) {
...
}
```

Set-Operationen als Anweisungen (4)

- For-Each-Anweisung
  - Regeln (Auszug, vollständige Regeln siehe Programmhilfe)

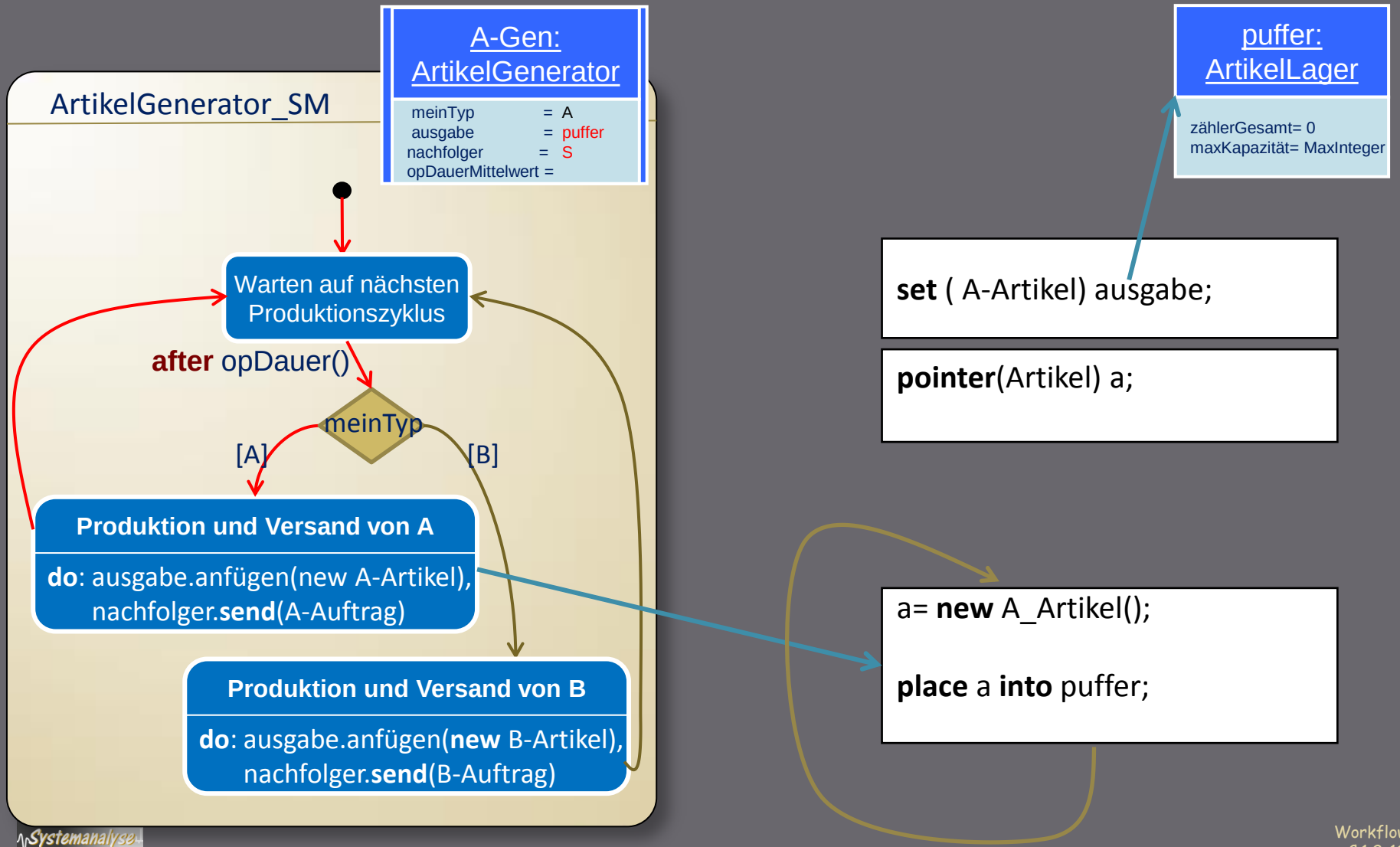
- bei reverse: Iteration in umgekehrter Reihenfolge

- Iterations-Variable
  ist nach Ende der Schleife NULL
  (außer bei vorzeitigem Abbruch der Iteration)

- aktuelles Element
  darf aus Set entfernt werden
  (Nachfolge-Element wird bereits vor Betreten des Schleifenkörpers bestimmt)

- sollte das Nachfolge-Element jedoch nicht mehr im Set enthalten sein, wird die Iteration von vorn gestartet

Set-Anwendung: Maschinenbelegung



Position

- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Moc
- **Teil C**
  Die ausf
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Typ
7. Set
8. Anweisungen, Prozeduren
9. Einfache Ausgabe
10. Modellierungselemente in SLX (allgemein)
11. Prozesse und Pucks
12. SLX- Laufzeitsystem (Simulator-Kern)
13. Zeitkonzepte
14. Behandlung von Zeit- und Zustandsereignissen

Prozeduren/Funktionen

- **existieren in Form**
  - globaler Prozeduren und
  - Klassen-Methoden

- **Definition:**

  procedure <Name> (<Parameter_Def>, ...) [**returning** <Typ>] {

  ...

  }

- **Parameter-Definition:**

  [<Richtung>] <Typ> <Name>

in | **out** | **inout**

Unterschied **out** zu **inout** unklar

Prozeduren

- Richtung: **in**

  nur Lesen ist erlaubt

```
procedure p (in int i) {
        i = 2;
•• Semantic error: "i" is an IN argument;
it cannot be modified
}
```

- Richtung: **inout**

  Lesen & Schreiben ist erlaubt

  Parameter werden per Referenz übergeben!

- Richtung: **out**

  laut Manual ist nur Schreiben erlaubt, aber tatsächlich ist auch Lesen möglich

  Parameter werden per Referenz übergeben!

```
procedure p(out int i) {
   // i == 1
   int j = i; // lesender Zugriff
   i = 2;
}
procedure main() {
   int i = 1;
   p(i);
   // i == 2
}
```

Programmfluss-Steuerung

- **Programmeintrittspunkt**
  - **procedure** main () { }
  - **procedure** main (**int** argc, **string**(*) argv) { }

- **Programmflusssteuerung**

  Verzweigung
  - **if-then-else,**
  - **switch-case**

  Wiederholung
  - **for, while, do-while,**
  - **continue, break**

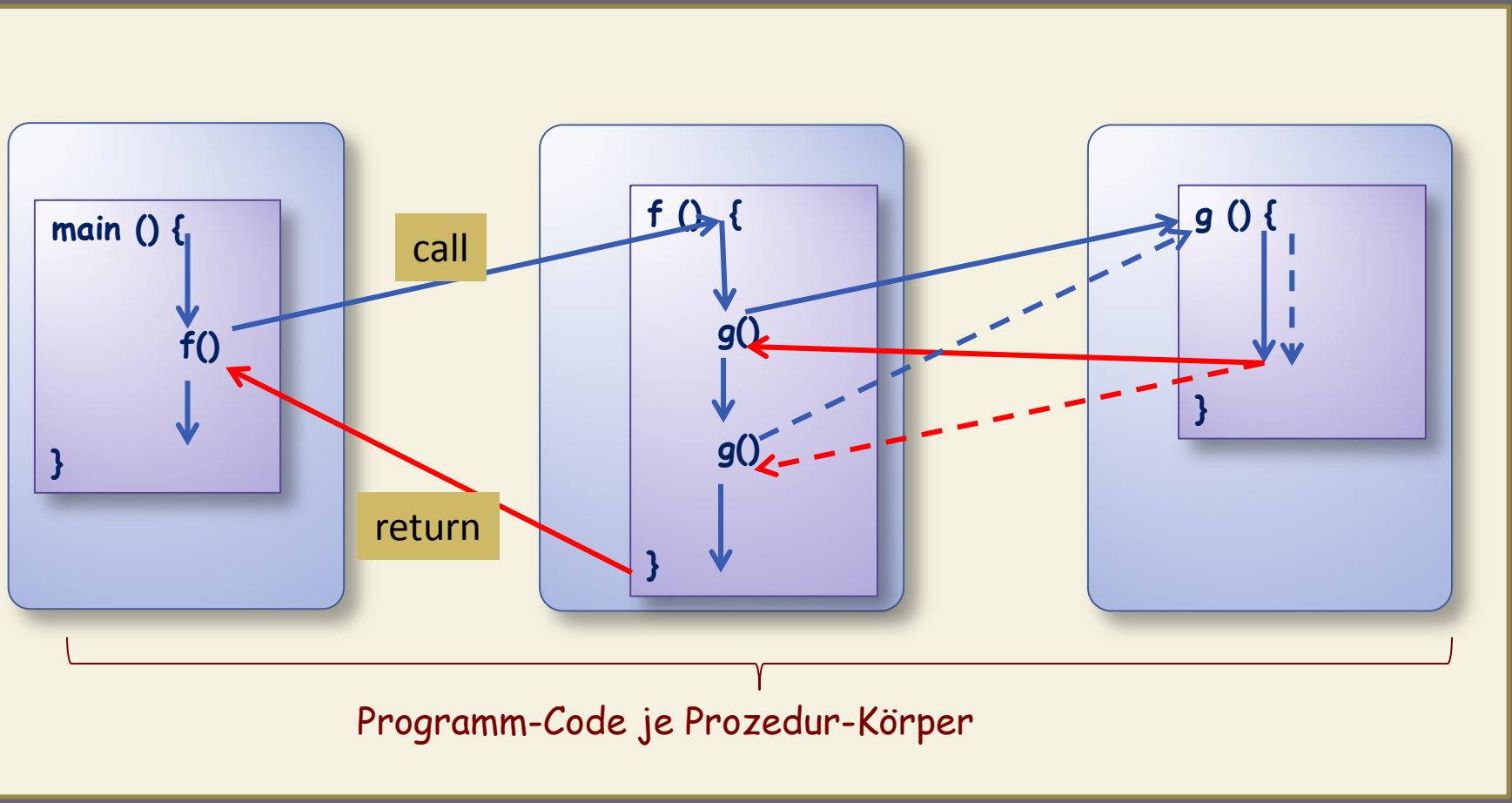
  zusätzlich
  - **forever** (= **while** (**true**)),
  - **goto,**
  - **exit()**


```
procedure main() {
  Lbl:  if (...) { ... }
        else {... };

        while (...) {
          ...
        };

        do {
             ...
        while (...);

        forever {
          ...
         if (...) break;
          ...
         }
       ...
       goto Lbl;
}
```

evtl. Fragen in Übungen

Routinen (Prozeduren/Funktionen)

normaler Funktions-Prozedur-Aufrufmechanismus



Position

- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Moc
- **Teil C**
  Die ausf
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Typ
7. Set
8. Anweisungen, Prozeduren, Koroutinen
9. Einfache Ausgabe
10. Modellierungselemente in SLX (allgemein)
11. Prozesse und Pucks
12. SLX- Laufzeitsystem (Simulator-Kern)
13. Zeitkonzepte
14. Behandlung von Zeit- und Zustandsereignissen

Ein- und Ausgabe

- E/A- Operationen für Standardgeräte
  - Bildschirm
  - Tatstatur

~ vereinfachtes **printf** aus C
Ausgabe entspr.: **write file=stdout**

- Ausgabe: Eingabe später)

Syntax

print [**options** = <style>,...] [**(** <expression> ,... **)**] <picture> ;

Liste von Ausgabe-Optionen

bold
italic
red
underline

Liste von Ausgabe-Größen

Zeichenkette zur Beschreibung des Ausgabeformates

picture	Bedeutung
–	Ausgabe einer ganzen Zahl einschließlich Vorzeichen. Reelle Werte werden gerundet. Die Anzahl der Zeichen definiert den minimalen Zeichenbedarf.
–.___	Ausgabe von reellen Zahlen. Ein Punkt muß geschrieben werden. Die Anzahl der '_' nach dem Punkt bestimmt die Anzahl der auszugebenden Nachkommastellen.
\n	Zeilenumbruch (Newline)
\t	Tabellatorsprung
|	Zeichen zur Ausrichtung der Ausgabe. In Abhängigkeit von der Position dieses Zeichens wird der Ausgabetext links-, rechtsbündig oder zentriert ausgegeben

Print-Anweisung

Beispiel

```
int i = 2;
string(5) s = "hello";

print(i, s)  "_ und dann _";
```

```
2 und dann hello
```

print [**options** = <style>,...] [(<expression> ,...)] <picture> ;

```
print options=bold "Mostly boldface and \Bsome non-boldface\B text\n"
```

Mostly boldface and some non-boldface **text**

```
EX-0000-Print.slx: SLX-64 CT103  Lines: 1,753  Errors: 0  Warnings: 0  Lines/Second: 249,221  Memory: 2 MB
Execution begins
1                    1       1
10                  10      10
100                100     100
1000              1000    1000
10000            10000   10000

x =    0.000  sin(x) =     0.000
x =    1.000  sin(x) =     0.841
x =    2.000  sin(x) =     0.909
x =    3.000  sin(x) =     0.141
x =    4.000  sin(x) =    -0.757
x =    5.000  sin(x) =    -0.959
x =    6.000  sin(x) =    -0.279
x =    7.000  sin(x) =     0.657
x =    8.000  sin(x) =     0.989
x =    9.000  sin(x) =     0.412
x =   10.000  sin(x) =    -0.544
Execution complete
Objects created: 7 passive and 1 active   Pucks created: 2   Memory: 2 MB   Time: 0.08 Seconds
```

EX-0000-Print.slx - main 1/1

```
//*****************************************************************************
//          EX-0000-Print
//*****************************************************************************
module myModule {
        int i;
        double x;

procedure main() {
        for (i=1; i <= 10000; i=i*10) {
                print (i, i, i) "|__________ __________| ______|______ \n";
        }
        print "\n";
        for (i=1; i <= 11; i++) {
                print options= red, italic (x, sin(x)) "x =\R__.___|\R   sin(x) =\R__.___|\n";
                x+= 1;
        }

}
}
```

picture wird je **print**-Anweisung individuell definiert

Log - Time 0

```
EX-0000-Print-Table.slx: SLX-64 CT103  Lines: 1,826  Errors: 0  Warnings: 0  Lines/Second: 251,802  Memory: 2 MB
Execution begins
```

Item Name	Weight Class	Item Weight [kg]	Item Prize Class	Item Prize [€]	Item Number per Pallet	Value per Palett [€]	Paletts per SmallTruck	Paletts per Container	Item Source	Item Sink	Total Item Number
XYZ	heavy	1500	high	10000	2	20000	0	6	A	B	103
Melone	light	30.500		3.68	3	100.50	1	5	B	C	2456

```
Execution complete
Objects created: 7 passive and 1 active  Pucks created: 2  Memory: 2 MB  Time: 0.10 Seconds
```

Picture-Variable lassen sich für mehrere **print**-Anweisungen verwenden

EX-0000-Print-Table.slx - main 1/1

```
//*****************************************************************************
//        EX-0000-Print Table
//*****************************************************************************
module myModule {
        int i;
        double x;

        OEM static string(*) article_format =
        "|__________  __________|  _____.__|  __________|  _____.__|  __________|  _____.__|  __________|  __________|  _______|  _______|  _______|\n";


procedure main() {
        print options=bold
                        ("", "", "Item", "Item", "Item", "Item", "Value", "Paletts", "Paletts", "", "", "Total") article_format;
        print options=bold
                        ("Item", "Weight", "Weight ", "Prize", "Prize", "Number", "per Palett", "per",  "per", "Item", "Item", "Item") article_format;

        print options=bold, underline
                        ("Name", "Class", "[kg]",  "Class", "[€]", "per Pallet", "[€]", "SmallTruck", "Container", "Source", "Sink", "Number") article_format;

        print ( "XYZ", "heavy", 1500, "high", 10000, 2, 20000, 0, 6, "A", "B", 103) article_format;
        print ("Melone", "light", 30.5, "", 3.676, 3, 100.5, 1, 5, "B", "C", 2456) article_format;

}
}
```

Format als String-Variable ‘artikel_format’

3-zeiliger Tabellenkopf

2-zeiliger Tabelleninhalt

Position

- **Teil A**
  Aspekte
  dynamis
- **Teil B**
  Die Moc
- **Teil C**
  Die ausf
  SLX
- **Teil D**
  Modellie

- **C.1**
  Einführung und Ba
- **C.2**
  Stochastische Pro
- **C.3**
  Vertiefung der SL
- **C.4**
  GPSS-Elemente
- **C.5**
  DISCO-Elemente
- **C.6**
  Basissprache (Erg

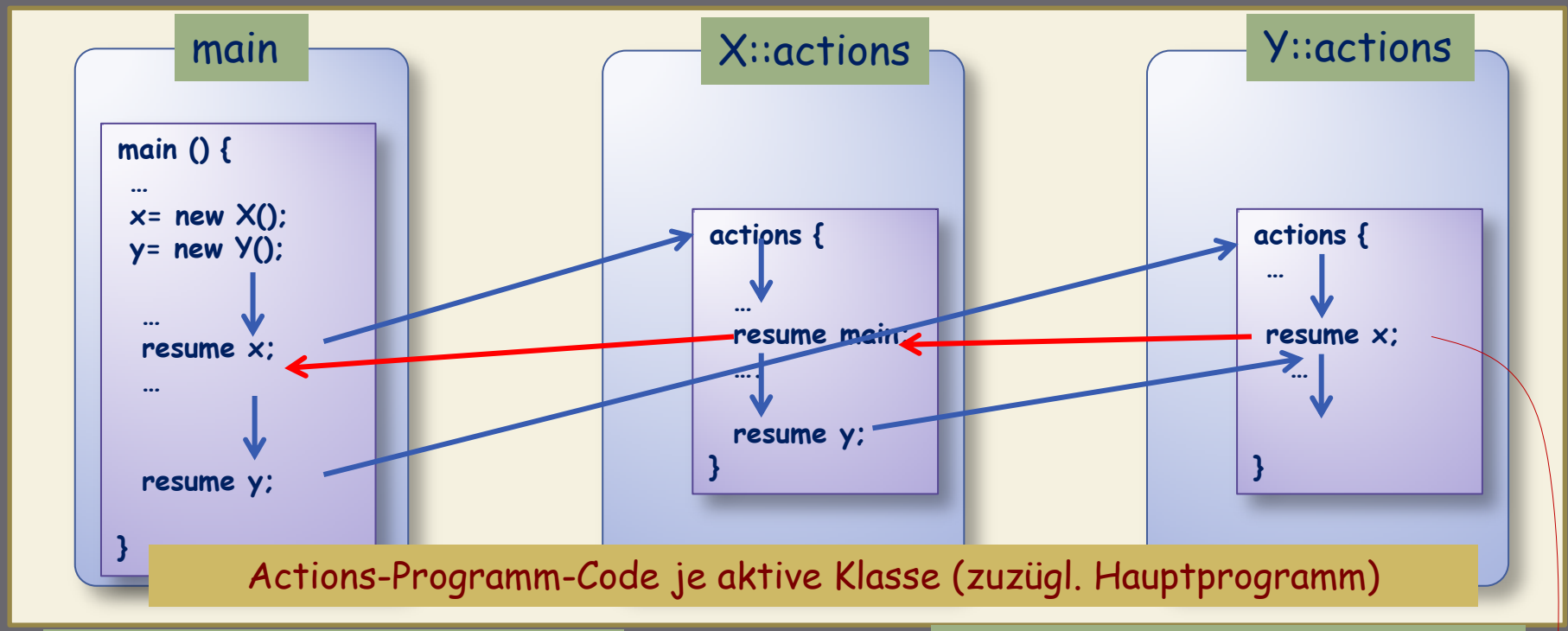
1. SLX-Überblick
2. Syntaxkonventionen
3. Elementare Datentypen
4. Klassen
5. Objekte
6. Typ
7. Set
8. Anweisungen, Prozeduren, Koroutinen
9. Einfache Ausgabe
10. Prozesse und Pucks
11. SLX- Laufzeitsystem (Simulator-Kern)
12. Zeitkonzepte
13. Behandlung von Zeit- und Zustandsereignissen

Action-Property = Koroutinen

```
class X {...}
class Y {...}
```

```
pointer (X) x; // NULL
pointer (Y) y; // NULL
```

Beispiel: 3 Koroutinen



X-Objekte befinden sich i.Allg. in unterschiedlichen Ausführungspositionen

Y-Objekte befinden sich i.Allg. ebenfalls in unterschiedlichen Ausführungspositionen

Zur Speicherung der jeweiligen Auführungspositionen und anderer Statusinformationen benötigt man **Laufzeitobjekte**, die den jeweiligen Objekten aktiver Klassen zugeordnet sind

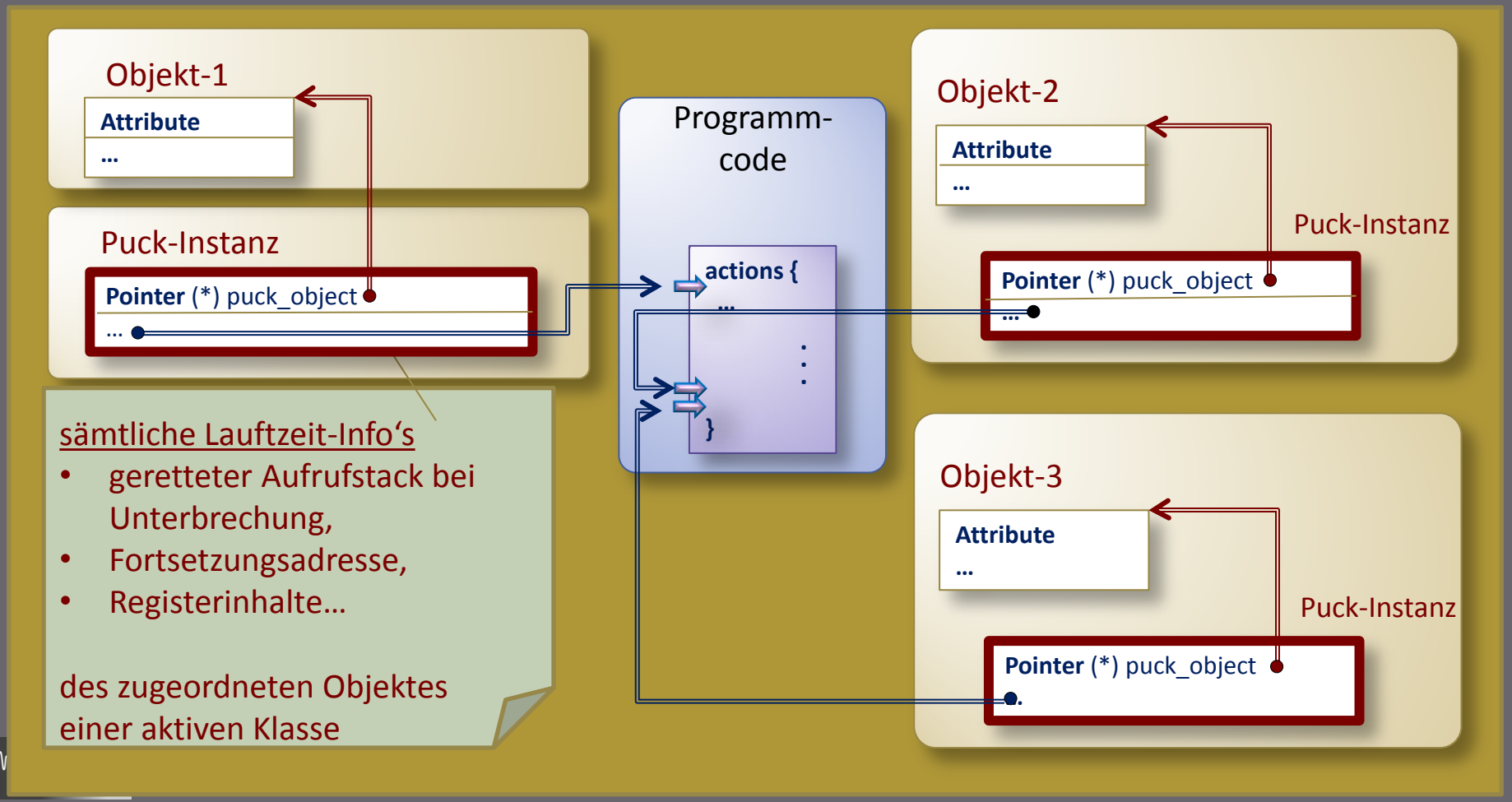
heißen in SLX: **Pucks**

resume-Operation heißt in SLX: **activate**

Pseudoparallele Verhaltensrealisierung

SLX-Simulationsprogramm
beschreibt die sequentialisierte Ausführung paralleler **Prozesse**

Prozess = **Objekt** (aktiver Klasse) + **Puck-Objekt** (Laufzeitinfos) + **Actions-Code**



SLX-Besonderheit

- Jede oo-Simulationssprache besitzt

  Laufzeitkonzepte zur quasiparallelen Ausführung von Objekt-Aktionen :
  Scheduling-Instanzen (Einträge im Ereigniskalender) mit Bezug
  - zum Objekt und
  - zum Abarbeitungszustand

  } Prozess

- SLX einzige(?) Sprache,

  die einem Active-Class-Objekt
  mehrere (beliebig viele) Scheduling-Instanzen zuordnen kann (1:n-Zuordnung)
  → Grundlage für interne Parallelisierung eines Prozesses selbst (später)

  Preis: zusätzlicher Nutzer-Management-Aufwand
  für den Standardfall einer 1:1-Zuordnung

  Scheduling-Objekte heißen in SLX **Pucks**

- Simula, GPSS, ODEMx-Bibliothek(C++), …

  benutzen vergleichbare Laufzeitverwaltungsstrukturen (**Sched**-Klasse statt **Puck**-Klasse)
  aber **1:1-Zuordnung** Puck-Instanz ←→ Active-Class-Objekt-Instanz
  Ausführung eines einzelnen Objektes kann nicht weiter parallelisiert werden

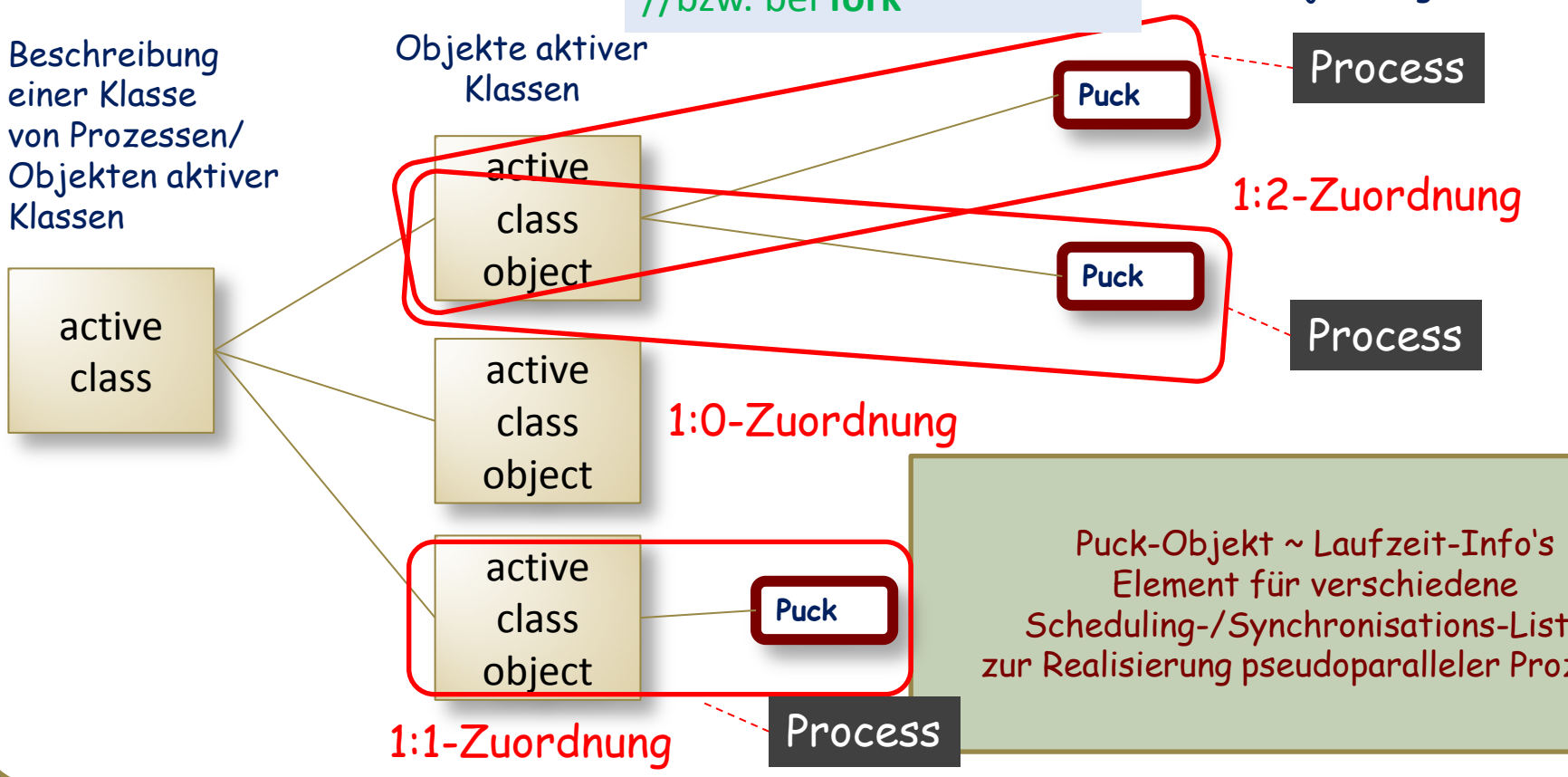
Zusammenhang: Klasse – Objekt – Prozess - Puck

```
pointer (CLASS) o;
o= new CLASS();
```

```
activate o;
//bei Erstaufruf für o
//bzw. bei fork
```

pseudoparallel ablaufende Prozesse, die den Attribut-Bereich des Objektes geteilt nutzen

Beschreibung einer Klasse von Prozessen/ Objekten aktiver Klassen



Puck-Objekt ~ Laufzeit-Info's
Element für verschiedene
Scheduling-/Synchronisations-Listen
zur Realisierung pseudoparalleler Prozesse

Klasse Puck

{passive} Puck	
frozen_successor	**pointer** (Puck)
mark_time	**double**
move_time	**double**
priority	**int**
puck_object	**pointer** (*)
state	**enum** PuckStatus
wait_incurred	**boolean**

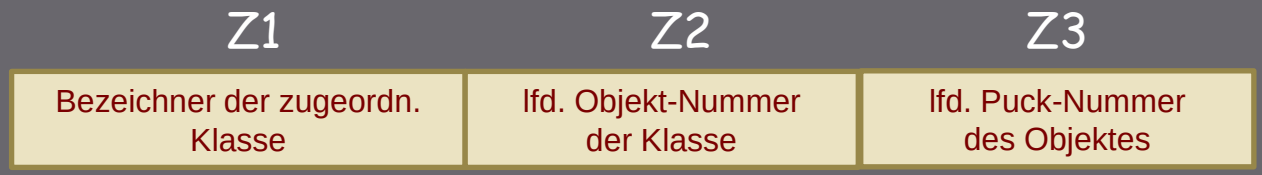
Identifikations-Zeichenkette -
beliebiger Objekte
nutzerdefinierter Klassen=
Z1 || Z2
- von Puck-Objekten=
Z1||' '|| Z2 || '/'||Z3

Puck-Instanzen

- **Erzeugung**
  erfolgt implizit

- **Verwaltung**
  Eine Puck-Instanz (falls existent) befindet sich zu einem Zeitpunkt stets
  (1) in **genau einer** von 4 verschiedenen Puck-Listen des SLX-Laufzeitsystems und u.U. **gleichzeitig**
  (2) in (beliebig vielen) **weiteren Listen** des jeweiligen Anwendungsprogramms

für Ausgaben verschiedener SLX-Tools (Report, Debugger)



Beispiele: Zellenbrand 103/2 | main 1/1 | ship 21

Klasse Puck

{passive} Puck	
frozen_successor	**pointer** (Puck)
mark_time	**double**
move_time	**double**
priority	**int**
puck_object	**pointer** (*)
state	**enum** PuckStatus
wait_incurred	**boolean**

Puck-Instanzen

- **Erzeugung**
  erfolgt implizit

- **Verwaltung**
  Eine Puck-Instanz (falls existent) befindet sich zu einem Zeitpunkt stets
  (1) in **genau einer** von 4 verschiedenen Puck-Listen des SLX-Laufzeitsystems und u.U. **gleichzeitig**
  (2) in (beliebig vielen) **weiteren Listen** des jeweiligen Anwendungsprogramms

1. Moving Pucks (MP)
2. Scheduled Pucks (SP)
3. Waiting Pucks (WP)
4. Interrupted Pucks (IP)

Zuordnung erfolgt in Abhängigkeit des Zustandes des Prozesses, für das Puck-Objekt Zuständig ist

Zum Umgang von Puck-Zeigern

- **wissen bereits:**
  elementare Coroutinen-Komponente einer quasiparallelen Ausführung wird durch Puck-Instanz erfasst

- **Operand einer SLX-Scheduling-Operation**
  ist damit zwangsläufig ein Puck-Zeiger
  (Ausnahme: activate, hier aktives Objekt)

- **Verwaltung von Puck-Instanzen**
  (1) werden implizit erzeugt, per activate

  (2) in den Besitz des Zeiger-Wertes zu dieser Puck-Instanz kann nur die aktivierte Coroutine selbst kommen, per ACTIVE

  (3) dieser Wert kann dann bei Bedarf an andere Coroutinen weitergereicht werden
  (benötigt zum Scheduling)

{passive} Puck	
frozen_successor	**pointer** (Puck)
mark_time	**double**
move_time	**double**
priority	**int**
puck_object	**pointer** (*)
state	**enum** PuckStatus
wait_incurred	boolean

```
procedure main () {
        pointer (X) xPtr= new X();
        activate xPtr;
```

```
class X {
  actions {
     pointer(puck) p;
     p = ACTIVE;
     …
  }
}
```

Puck-Listen des SLX-Laufzeitsystems

Ereigniskalender (strukturiert in zwei separate Puck-Listen)

1) erfasst Ereignisse zum **aktuellen** Modellzeitpunkt

Moving Pucks (MP): Einträge sind Puck-Objekte

- mit Verweis zum zugehörigen Objekt einer aktiven Klasse
- mit Ereigniszeitpunkt (= aktuelle System-Modellzeit)
- mit Puck-Zustand**: running (Debugger: moving/movable)**
- sortiert nach **Priorität** (alle haben die gleiche Ereigniszeit)

2) Liste **zukünftiger** Ereignisse

Scheduled Pucks (SP): Einträge sind Puck-Objekte

- mit Verweis zum zugehörigen Objekt einer aktiven Klasse
- mit Ereigniszeitpunkt (> aktuelle System-Modellzeit)
- mit Puck-Zustand: **scheduled**
- sortiert nach **Ereigniszeit**

Puck-Listen des SLX-Laufzeitsystems

Blockierungslisten (zwei separate Listen, **unabhängig** von Anwendung)

3) Liste **wartender** Pucks

Waiting Pucks (WP): Einträge sind Puck-Objekte mit

- mit Verweis zum zugehörigen Objekt einer aktiven Klasse
- mit Ereigniszeitpunkt (ohne Belang)
- mit Puck-Zustand: **waiting**
- **sortiert** nach FIFO

4) Liste **unterbrochener** Pucks

Interrupted Pucks (IP): Einträge sind Puck-Objekte mit

- Verweis zum zugehörigen Objekt einer aktiven Klasse
  die während ihrer Scheduling-Phase unterbrochen worden sind
- mit Ereigniszeitpunkt (ohne Belang)
- Puck-Zustand: **interrupted**
- **sortiert** nach FIFO

Weitere Puck-Listen

Blockierungslisten, von Anwendung **abhängig**e Listen

(3) Liste zustandsbedingter blockierter Pucks
je control-Variable

(4) Listen, die vom Anwendungsprogramm für Blockierungen /Deblockierungen
per: **wait** list= name;
oder: **reactivate** list= name;

Verkettung erfolgt mittels …

{passive} Puck		
frozen_successor	**pointer** (Puck)	NULL
mark_time	**double**	
move_time	**double**	
priority	**int**	
puck_object	**pointer** (*)	
state	**enum** PuckStatus	
wait_incurred	**boolean**	

Puck-Attribute

puckPointer->frozen_successor
Zur Verkettung in SLX-Laufzeitstrukturen

puckPointer->mark_Time
gespeicherte Modellzeitmarkierung (initial: Erzeugungszeitpunkt des zugehörigen Active-Class-Objektes)

puckPointer->move_Time
Zeitpunkt (Modellzeit) der Aufnahme der Abarbeitung (geplante Ereigniszeit)

puckPointer->puck_object
Zeiger zum zugehörigen Active-Class-Objekt

puckPointer->state
Status des Pucks, der den Abarbeitungsstand des zugehörigen Prozesses/Objektes beschreibt

puckPointer->wait_incurred
gesetzt, wenn zugeordnetes Objekt zustands-blockiert (wait until ...)

{passive} Puck		
frozen_successor	**pointer** (Puck)	NULL
mark_time	**double**	
move_time	**double**	
priority	**int**	
puck_object	**pointer** (*)	
state	**enum** PuckStatus	
wait_incurred	**boolean**	

state	
running	Puck ist in Liste MP; Prozess wartet auf Ausführung oder ist aktiv
scheduled	Puck ist in Liste SP; Prozess ist um ein Δt verzögert
waiting	Puck ist in Liste WP; Prozess ist blockiert
interrupted	Puck ist in Liste IP; Prozess ist unterbrochen/blockiert
terminated	Prozess ist beendet, es gibt aber noch Verweise auf den Puck

Erzeugung und Vernichtung von Pucks

- **Erzeugung**

  ein Puck
  - wird mit dem Start der Prozedur main() oder
  - mit dem Aktivieren eines aktiven Objekt per activate oder
  - per Aufsplittung der Programmsteuerung mittels fork

  generiert

- **Vernichtung**
  - der Puck zur Prozedur main() wird mit Beendigung dieser Prozedur vernichtet
  - der Puck eines aktiven Objektes wird vernichtet, wenn
    a) die letzte Anweisung der **actions**-Property erreicht, bzw.
    b) wenn die **terminate**-Anweisung ausgeführt wurde.
  - Die Pucks können noch weiterhin physisch existieren, wenn noch Referenzen auf diesen Puck existieren (use count >0).