Modellierende Spezifikation, umgangssprachlich

- ADT Polynom

  mit ganzzahligen Koeffizienten

```
interface PolyIF {
 // model:  polynominals from mathematics
 //         with integer constants
 //         e.g.   c0 + c1*x1 + ...cn*xn
 //PolyIF ();
// effect: returns zero polynominal

 //PolyIF (int c, int n) throws NegativeExpException;
 // requires:  n >= 0
 //            else NegativeExpException
 // effect: returns monominal c*xn
```


Modellierende Spezifikation, umgangssprachlich

```
 //methods
 int degree();
   // effect: returns the degree of the largest
   // exponent with a non-zero coefficient of this,
   // 0 if this is the zero Poly
 int coeff(int e);
   // effect: returns coefficient of the term of
   // this, whose exponent is e
 Object add (Object p) throws NullPointerException;
   // requires: p != null , else NullPointerException
   // effect: returns this + p
   ...
 // mul, sub, minus(unary) like add
 }
```

Modellierende Spezifikation, umgangssprachlich

- Bemerkungen
  - Umgangssprachliche Spezifikation akzeptabel, wenn klar und eindeutig.
    Angabe eines Beispiels (" e.g. ....) sinnvoll
  - Polynome nicht veränderbar (nach dieser Spezifikation!)
    => keine Invariante nötig, nichts ändert sich
  - Mehr als ein Konstruktor

Implementierung von ADT

2.3.3 Implementierung von Abstrakten Datentypen und Korrektheitskriterien

Spezifikation	◄ ? ►	**Implementierung**
abstraktes Modell		konkrete Repräsentation
abstrakte Invariante		konkrete Invariante
abstrakte/konkrete Signatur		konkrete Signatur
abstrakte Vorauss./Effekte		konkrete Vorauss./Effekte

Implementierung von ADT

- Zentrales Problem: genügt Implementierung der Spezifikation?
  - Gibt es zu jeder Operation des Modells Implementierung?
    ... Triviale Forderung
  - Gibt es zu jedem abstrakten Wert (des Modells) eine konkreten?
  - Entspricht jeder konkrete Wert eindeutig einem abstrakten?
  - Kann es mehrere konkrete Werte geben, die einen abstrakten repräsentieren?
  - Welche konkreten Invarianten müssen gelten?
    Berücksichtigen: Invarianten aus Modell und Implem.
  - Konkrete Voraussetzungen / Effekte?

Implementierung von ADT: Beispiel Boolsche Schlange

```
ADT BQueue  {
    model: q = e | q = (x_i)_{i=1..k} && type(x_i) = Bool
    inv:    0 < k <=  MAX
init :: -> BQueue
   requires:    True
   effect :     result e

enq:: Bool -> BQueue -> BQueue
  requires:    k < MAX
          -- else error Overflow
  effect:    result of enq(x)
                | Q == e    = (x),
               | otherwise =(x_i)_{i=1..k+1}, x_1==x ,
                            (x_i)_{i=2..k+1}== Q
              -- Q queue before enq
         -- x attached at front of sequence
```

Beispiel Boolesche Schlange

```
deq :: BQueue -> BQueue;
  requires:    q != e
          -- else error Underflow
  effect:
      result
               |k==1   = e
               |k > 1  = (x_i)_{i=1..k-1} , (x_i)_{i=1..k}==Q
      -- x dequeued from rear
next :: BQueue -> Bool
  requires:    q !=e
  effect:      result x_k, q == Q == (x_i)_{i=1..k}
     -- next taken from the rear of sequence
}
```


Beispiel Boolesche Schlange: Repräsentation

- Folge von Wahrheitswerten kann als Dualzahl interpretiert werden
- Eine Alternative:
  Präfix 1 verwenden:

```
1: [1,0,0,1] <-  25
1:[1,0]      <-   6
1:[0,1]      <-   5
1:[1]        <-   3
```

- Eindeutig, da $2^k + ?_{i=0..k-1}$ bi $*2^i$ eindeutig

```
1,0,0,1    <-  9
1,0        <-  2
0,1        <-  1
  1     ?? <-  1
```

Konkreter Werte repräsentiert verschiedene abstrakte Werte: ~~unbrauchbar~~!

Beispiel Boolesche Schlange: Repräsentation

- Repräsentation

  (val, k) where val = $2^k + \sum_{i=0..k-1} bi * 2^i$

  -- k list length

	Konkrete Werte		Abstrakte Werte
1:	`[1,0,0,1]` <-	`(25,4)` ->	`1,0,0,1`
	`1:[1,0]` <-	`(6,2)` ->	`1,0`
	`1:[1]` <-	`(3,1)` ->	`1`
	`1:[0]` <-	`(2,1)` ->	`0`
	`1:[]` <-	`(1,0)` ->	`e`

Entsprechen alle Paare `(n,m)` konkreten Werten? Welche?

Beispiel Boolesche Schlange: Repräsentation

Haskell-Implementierung:

```
type Val    = Int
type Length = Int
data BQueue = BQue (Val, Length)
              --- BQue  Konstruktor

-- inv (of representation (v,k)) :
--      2^k <= v < 2^(k+1) && 0 <= k < MAX = 30
```

Beispiel Boolsche Schlange: Implementierung

- Operationen

```
deq :: BQueue -> BQueue;
requires:    q != e
        -- else error Underflow

deq bq
-- requires: leng bq > 0
--  effects: returns (x 'div'2, k-1), bq=BQ=(x,k)
  | leng bq == 0   = error "underflow"
  | otherwise      = Bque (v',leng  bq –1)
     where v' = div ( val bq) 2
 -- remove at rear!

leng (a,b) = b
val  (a,b) = a
```


Beispiel Boolsche Schlange: Implementierung

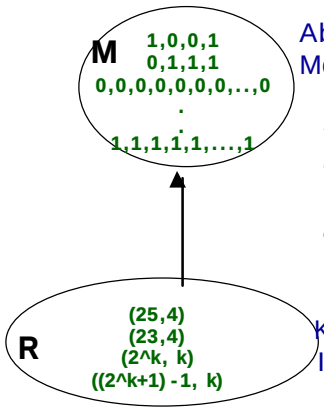
```
enq:: Bool -> BQueue -> BQueue

enq  b bq

-- requires: k < MAX
-- effects:  result ...
--           See code
--           bq == BQ
--           (nothing changed)
  | b == True
      =  BinQ (v +2^(k+1), k+1)
  | b == False
      =  BinQ (v +2^k , k+1)
    where v = val bq
          k = leng bq
```

```
enq 1  1[001] = 1[1001]
       1[001]
     +10[000]
   =  1[1001]
enq 0  1[001] = 1[0001]
       1[001]
      +1[000]
   =  1[0001]
```

Analog für andere Operationen von **BQueue**

Abstraktionsfunktion

- Wie hängen konkrete und abstrakte Werte zusammen?



Abstrakte Werte, wie im Modell spezifiziert

Abstraktionsfunktion:

abstr :: R -> M

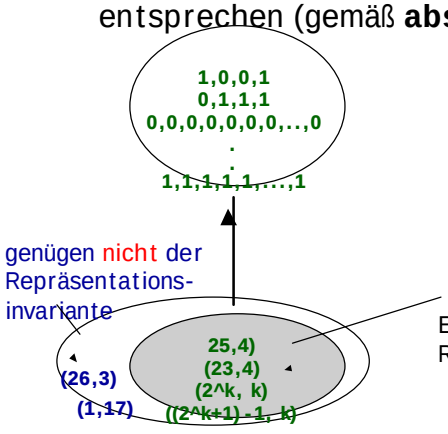
abstr ((v, k)) = $(b_{k-1},...,b_0)$

mit $v = 2^k + ?_{\ i=0..k-1}\ 2^i * b_i$

Konkrete Werte, wie in Implementierung spezifiziert

Repräsentationsinvariante

- Welche konkreten Werte sind erlaubt, d.h. entsprechen (gemäß **`abstr`**) abstrakten Werten?



Repräsentationsinvariante (Konkrete Invariante):

2^k <= v < 2^{k+1} &&

0 <=k < MAX = 30

Erlaubte Werte, genügen der Repräsentationsinvariante

Abstraktionsfunktion / Repräsentationsinvariante

- Abstraktionsfunktion legt explizit fest, welchem abstrakten Wert ein konkreter, erlaubter Wert der gewählten Repräsentation entspricht.
  Unerlässlich für jede Implementierung !
- Repräsentationsinvariante macht alle Annahmen, die der Implementierung der Operationen des ADT zugrunde liegen explizit und definiert die erlaubten Werte.
- Voraussetzungen und Effekte
  Zu {P} op {Q} im abstrakten Modell äquivalente konkrete Voraussetzungen / Effekte $\{P_k\}\ op_k\ \{Q_k\}$ angeben.

Abstraktionsfunktion / Repräsentationsinvariante

- Implementierung der Repräsentationsinvariante eignet sich als Zusicherung

```
... if not (invR rep) then error
                          "invalid value"
        where rep = enq x bq

invR :: bq -> Bool
invR  bq  = 2k <= v && (v < 2^(k+1)) &&
        (0 <=k) && (k < MAX)
    where bq = (v,k)
```

Implementierung von Poly

Repräsentation

Koeffizienten: **`int [] coeff // coeff[i] i-ter Koeffizient`**

Grad des Poly: **`int degree  // redundant, aber nützlich`**

Abstraktionsfunktion:

`// abstr(Poly c) = c`$_0$ **`+ c`$_1$**`*x`**^{1} **`+ ...`**

`//  where c`$_i$ **`= c.coeff[i], if  0 <= i < c.coeff.size`**

`//        c`$_i$ **`= 0 otherwise`**

```
public class Poly implements PolyIF {
// overview.....
 private int[] coeff;
 private int degree;

Poly () {
// effect: returns zero polynominal
coeff = new int[1]; degree = 0;
}
```


Implementierung von Poly(2)

```
Poly (int c, int n) throws NegativeExpException {
// requires:  n >= 0
//            else NegativeExpException
// effect: initializes Polynominal c*x^n
 if (n<0) throw new NegativeExpException();
 coeff = new int[n+1];
 if (n==0) {coeff[0]= 0; return;}
 for (int i=0; i < n; i++) coeff[i] = 0;
 coeff[n] = c;
 degree = n;
}
```

Repräsentationsinvariante?

```
// boolean invP(c){ c.coeff != null && c.coeff.length >= 1
// &&c.degree == c.coeff.length-1 && c.degree>0
// => c.coeff[n] != 0}
```

```
public Object add(Object q) throws NullPointerException {
// requires: p != null , elseNullPointerException
// effect: returns this + p
Poly large, small;
Poly p = (Poly)q;
if (degree > p.degree) {large = this; small = p;}
   else {large = p; small = this;}
int newDegree = large.degree;
if (degree == p.degree)
   for (int j=degree; j>0; j--)   //remove leading 0s
      if (coeff[j]+p.coeff[j] != 0) break;
         else newDegree--;       // establish invariant
Poly s = new Poly(newDegree);
 int i;
for (i=0; i<= small.degree && i<=newDegree; i++)
    s.coeff[i]=small.coeff[i]+large.coeff[i];
  for (int j=i; j<= newDegree; j++)
    s.coeff[j]= large.coeff[j];
  return s;
}
```

Implementierung von Poly(3)

```
public Poly sub (Poly p) throws
   NullPointerException {
   // effects: if null(p) throws
   //           NullPointerException
   //           else returns this – q
   return add(minus (q));
}
public minus() {
   // effects: returns the Poly object r with
   //          r.coeff[i]==-this.coeff &&
   //          r.degree ==this.degree
  Poly s = new Poly;
  s.degree = this.degree;
  for (int=0;i < degree; i++)
   s.coeff[i] = -this.coeff;
  return s;
}
```