Keller

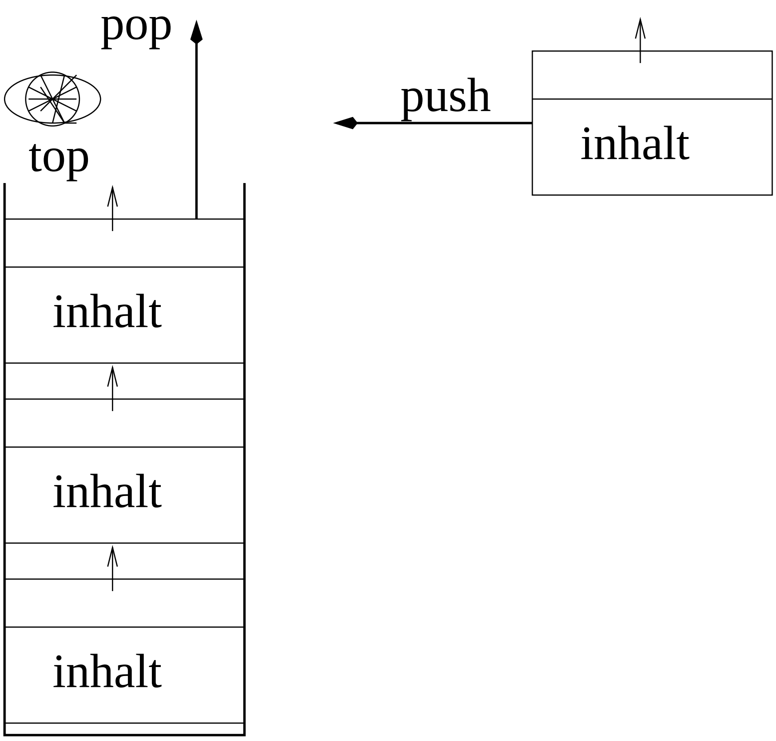
Mathematik: Folge

abstrakter Datentyp (was): Ein **Keller** ist ein abstrakter Datentyp, der eine Folge von Elementen darstellt.

- Die Folge hat ein möglicherweise undefiniertes oberstes Element *top* oder auch *peek* genannt.
- Die Operationen sind
  - das Ablegen eines Elementes auf den Keller (*push*),
  - das Entfernen des obersten Elementes (*pop*),
  - das Anzeigen des obersten Elementes und
  - der Test, ob der Keller leer ist.
- Wird nacheinander *push* und *pop* angewandt, ist der Keller unverändert. Nach *push* mit dem Element x liefert *top* das Element x.

Wie? Feld, ..., oder einfach verkettete Liste

Keller



```
import AlgoTools.IO;

public class Keller {

    private class KellerEintrag {
        Object inhalt;                  // Inhalt des KellerEintrags
        KellerEintrag next;             // zeigt auf naechsten
                                        //KellerEintrag
    }

    private KellerEintrag top;          // zeigt auf obersten
                                        //KellerEintrag

    public Keller () {                  // legt leeren Keller an
        top = null;
    }

    public boolean empty () {           // liefert true,
        return top == null;             // falls Keller leer
    }
```

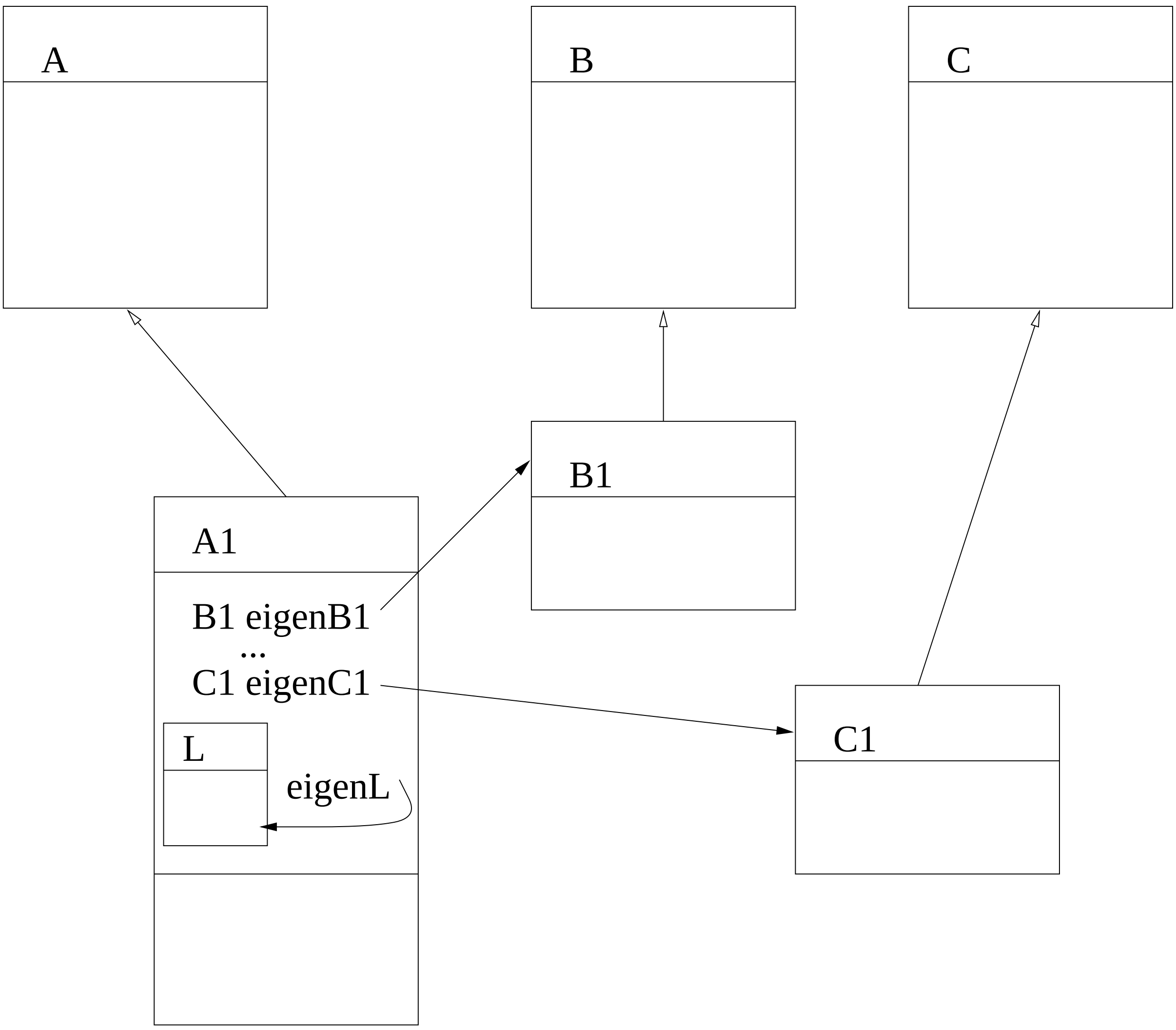
```
public void push (Object x)  {                    // legt Objekt x
        KellerEintrag hilf = new KellerEintrag (); // auf
                                                   //den Keller
        hilf.inhalt = x;
        hilf.next = top;
        top = hilf;
    }

    public Object top ()  {                        // liefert oberstes
        if (empty ()) IO.error ("in top: Keller leer");
        return top.inhalt;
    }

    public void pop ()  {                          // entfernt oberstes
        if (empty ()) IO.error ("in pop: Keller leer");
        top = top.next;
    }
}
```

Eingebettete Klasse

Eigenschaften von Objekten:
Variablen eines Typs
Dieser Typ ist eine andere Klasse.
Diese Klasse kann auch in die definierende Klasse eingebettet sein, so daß sie nur lokal sichtbar/verfügbar ist.



Keller in der JAVA-Bibliothek util

Stack ist als Unterklasse von **Vector** implementiert: die Kellereinträge sind die Elemente, die der Keller übereinander stapelt. Während bei **Keller** von der Methode **pop** nichts zurückgeliefert wird, ist der Rückgabewert der Methode **pop** bei der JAVA-Implementierung von **Stack** das entfernte Element.

Vector ist ein Feld mit flexibler Länge.

Planungsproblem

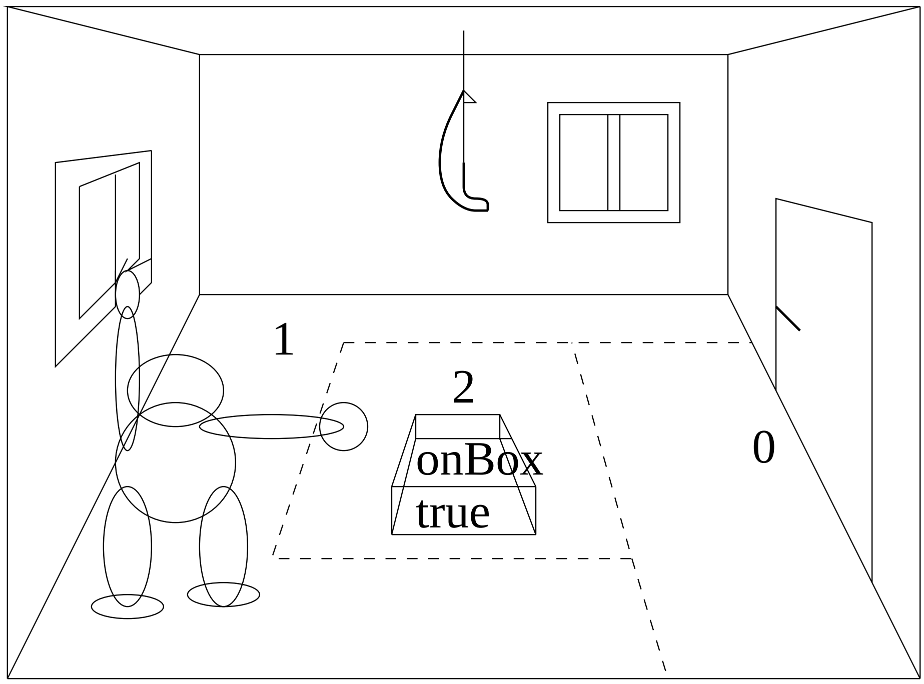
Gegeben: ein Anfangszustand, ein Zielzustand und eine Menge von Handlungen mit Vorbedingungen und einem Nachfolgezustand.

Finde: eine Folge von Handlungen, die vom Anfangs- in den Zielzustand führt.

Keller von Zuständen:
oberstes Element ist aktueller Zustand;
Nachfolgezustand wird obenauf gelegt (push);
bei Erreichen eines Zielzustands werden Zustände abgeräumt (pop).

Gibt es Zyklen? Erreicht man bestimmt das Ziel? Hält der Planungsalgorithmus unbedingt einmal an?

Affe und Banane



Ziel: Greifen der Banane

Handlungen:
Ausgangszustand → Nachfolgezustand

tryGrasp:
Box in der Mitte, Affe in der Mitte, Affe auf der Box
→ Greifen der Banane

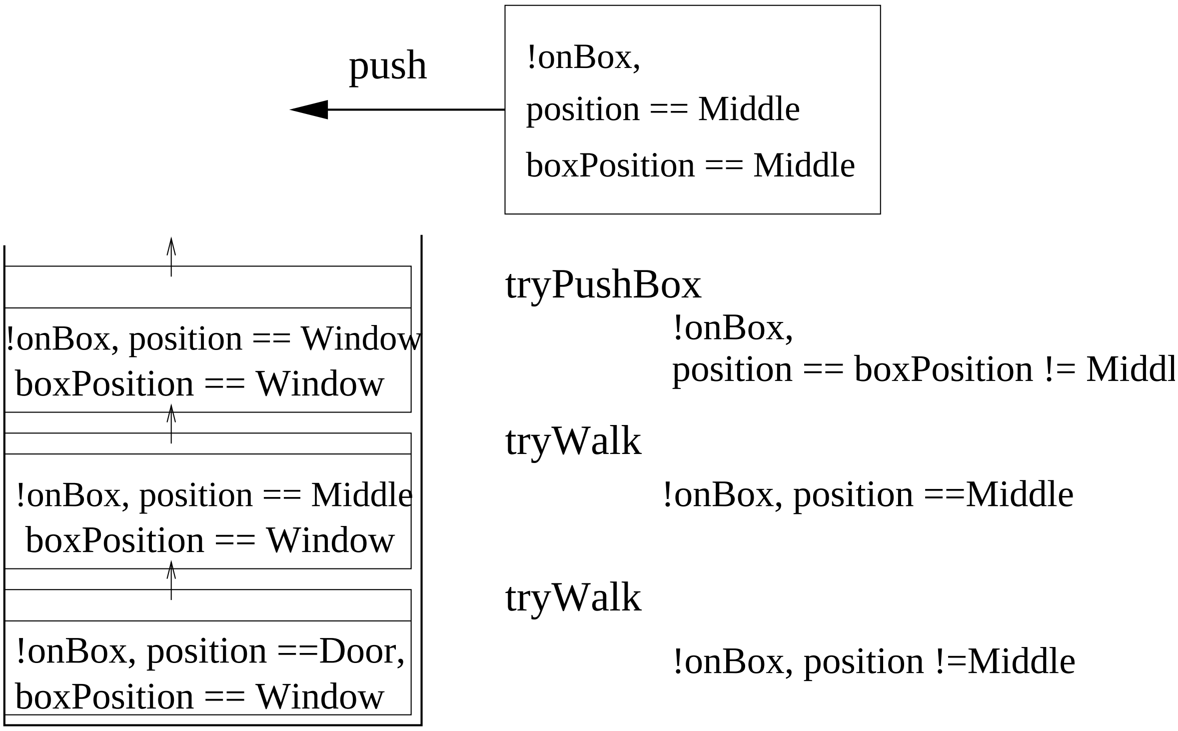
tryClimbBox:
Affe bei der Box, nicht auf der Box → auf die Box steigen

tryPushBox:
Affe bei der Box, nicht auf der Box, Position!= Mitte → Schieben der Box zur Mitte;
Affe bei d. Box, nicht auf d. Box, Position==Mitte → Schieben der Box zum Fenster, neu planen, Schieben zur Tür, neu planen;

tryWalk:
Affe nicht auf der Box, Position !=Mitte →
zur Zimmermitte gehen;
Affe nicht auf der Box, Position == Mitte →
zum Fenster gehen, neu planen, zur Tür gehen, neu planen.

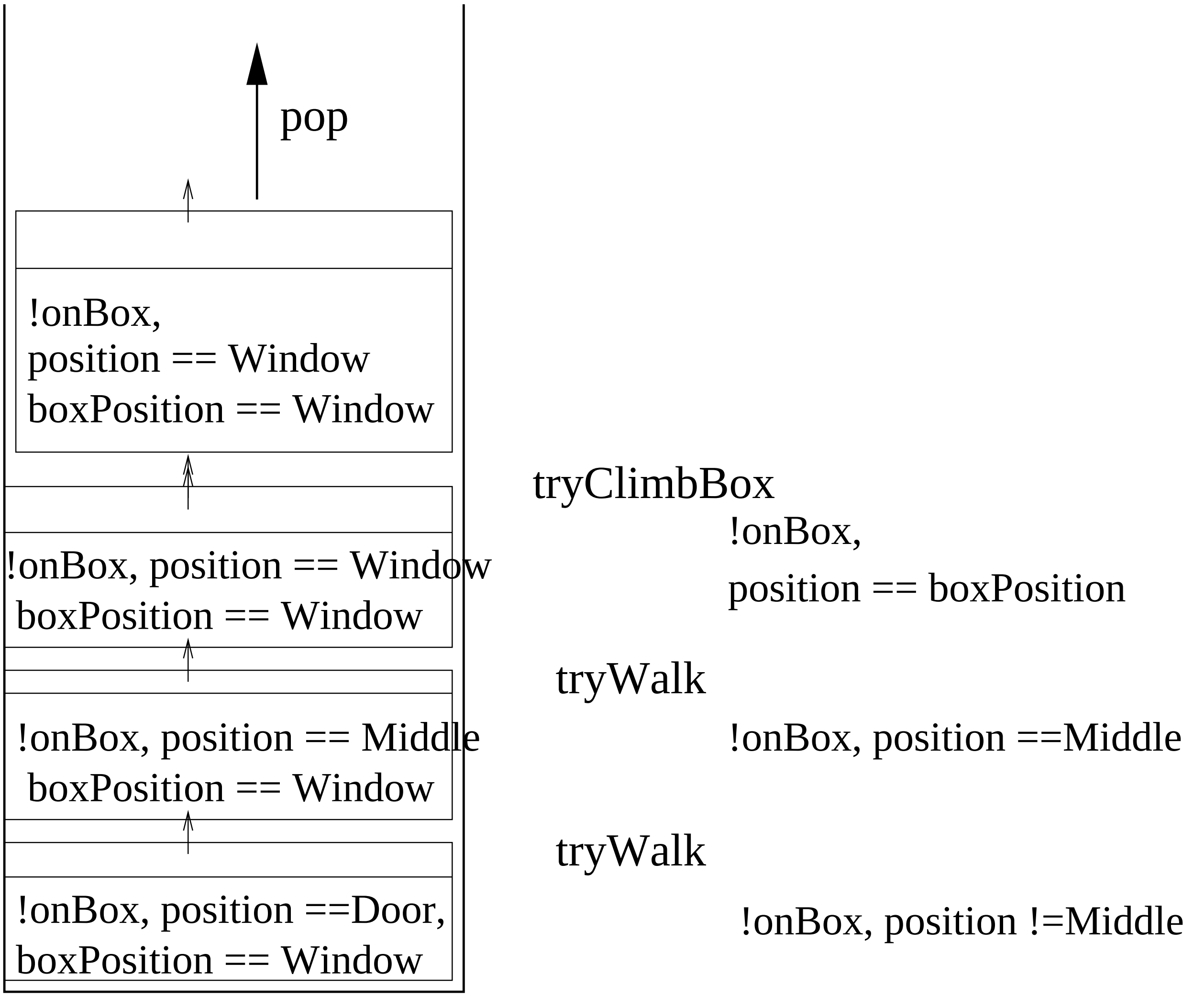
Planen mit einem Keller für Zustände

Handlung vergleicht Vorbedingung mit aktuellem Zustand (top des Kellers) und erzeugt Nachfolgezustand (push).



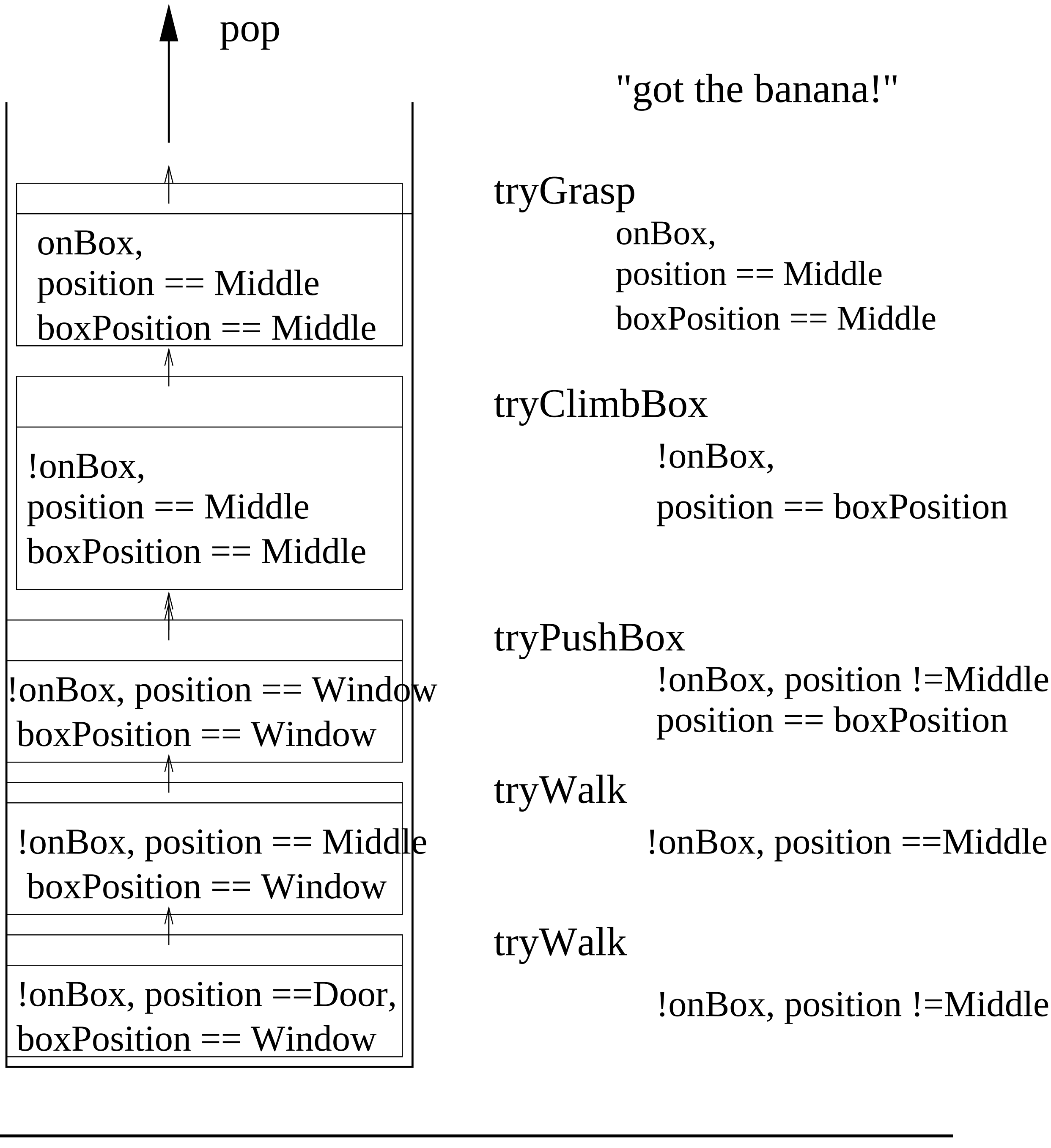
Rücksetzen bei 'Sackgassen' im Plan

Wenn in **tryAll** alle Handlungen versucht wurden, wird der aktuelle Zustand vom Keller entfernt (pop).



Ziel erreicht

Die Zustände, die zum Ziel geführt haben, werden vom Ziel bis zum Start vom Keller genommen und ausgedruckt.



```
import java.util.*;
 class State {
    static final int AT_DOOR = 0;
    static final int AT_WINDOW = 1;
    static final int MIDDLE = 2;
    boolean onBox;
    int position, boxPosition;
    String action;
    State (String _action, int _position, int _boxPosition, boolean _onBox) {
        action = _action;
        position = _position;
        boxPosition = _boxPosition;
        onBox = _onBox;
    }
```

```
void tryAll (Stack plan) {
        plan.push (this );
        tryGrasp (plan);
        tryClimbBox (plan);
        tryPushBox (plan);
        tryWalk (plan);
        plan.pop ();
    }

    void tryGrasp (Stack plan) {
        if (position == MIDDLE && onBox) {
            System.out.println ("got the banana!");
            while (!plan.empty ())
            System.out.println (plan.pop ());
            System.exit (0);
      }}
```

```
void tryClimbBox (Stack plan) {
        if (position == boxPosition && !onBox)
        new State ("ClimbBox", position, boxPosition, true).tryAll(plan);
    }

    void tryPushBox (Stack plan) {
        if (position == boxPosition && !onBox) {
            if (position != MIDDLE)
            (new State ("PushBox", MIDDLE, MIDDLE, onBox)).tryAll (plan);
            else {
                (new State ("PushBox", AT_WINDOW, AT_WINDOW,
                                        onBox)).tryAll (plan);
                (new State ("PushBox", AT_DOOR, AT_DOOR, onBox)).tryAll (plan);
            }
        }
    }
```

```
void tryWalk (Stack plan) {
    if (!onBox) {
        if (position != MIDDLE)
        (new State ("Walk", MIDDLE, boxPosition, onBox)).tryAll (plan);
        else {
            (new State ("Walk", AT_WINDOW, boxPosition, onBox)).tryAll (plan);
            (new State ("Walk", AT_DOOR, boxPosition, onBox)).tryAll (plan);
        }
    }
}
```

```
static public String posToString (int pos) {
        switch (pos) {
            case AT_DOOR: return "at door";
            case AT_WINDOW: return "at window";
            case MIDDLE: return "in the middle";
        }
        throw new RuntimeException ("Illegal Position: "+pos);
    }
```

```
public String toString () {
        return action + ": "+ "monkey "+ posToString (position) + ", "
                + (onBox ? "on box; ": "not on box; ")
                + "box "+ posToString (boxPosition);
    }
}


class Monkey {
    public static void main (String argv []) {
        new State ("Start", State.AT_DOOR, State.AT_WINDOW, false).tryAll (new Stack());
    }
}
```

Ergebnis

```
got the banana!

ClimbBox: monkey in the middle, on box;
box in the middle

PushBox: monkey in the middle, not on box;
box in the middle

Walk: monkey at window, not on box; box
at window

Walk: monkey in the middle, not on box;
box at window

Start: monkey at door, not on box; box at
window
```