ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 17: Ereignisorientierte Programmierung und Agenten

Unser Ziel in dieser Lektion

Unsere Kontrollstrukturen um einen flexibleren Mechanismus erweitern, der unter anderem interaktives und graphisches Programmieren (GUI) unterstützt.

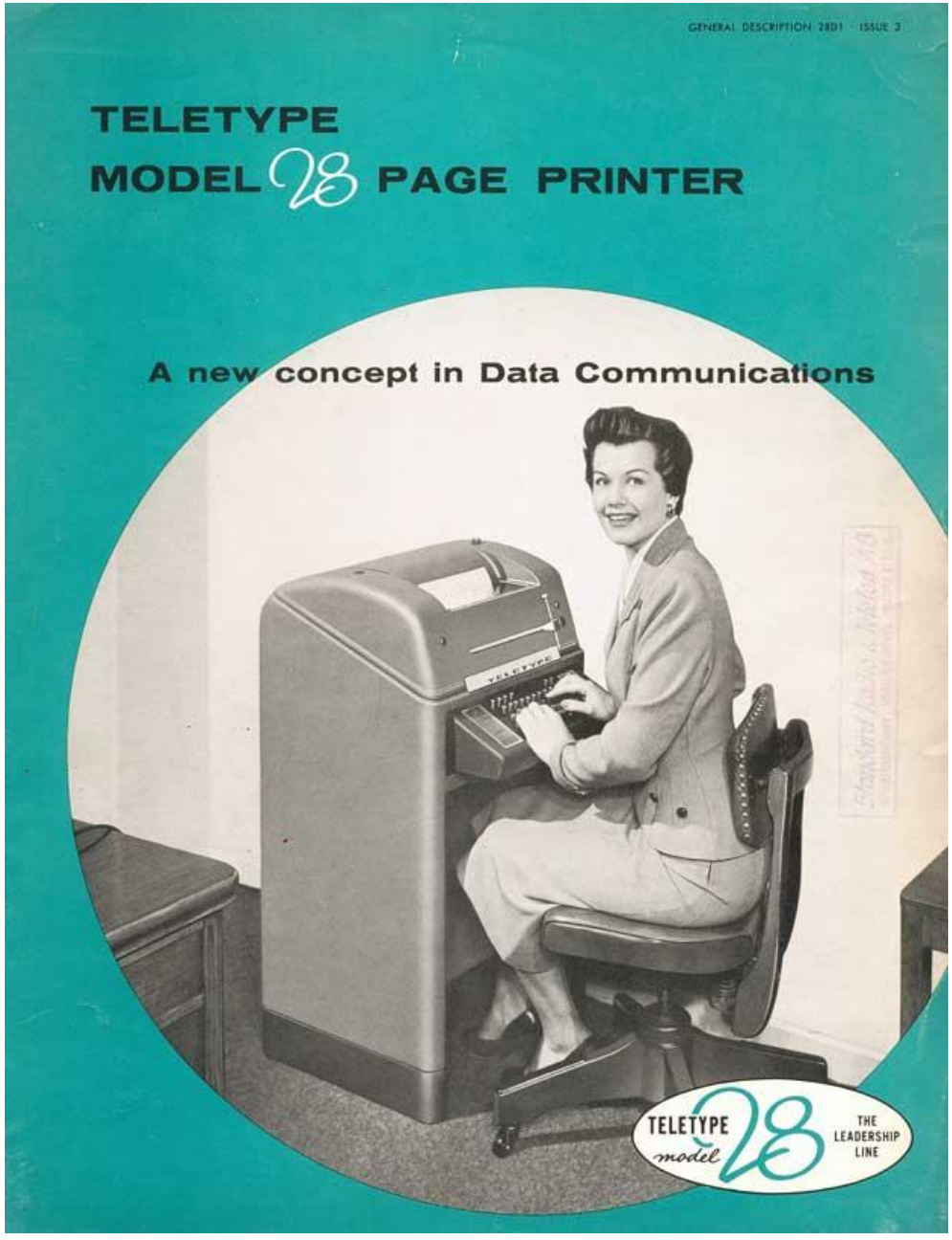
Der resultierende Mechanismus, Agenten, hat viele andere spannende Anwendungen.

Andere Sprachen haben Mechanismen wie z.B. Delegaten (*delegates*) (C#), *closures* (funktionale Sprachen) .

Input verarbeiten: traditionelle Techniken

Das Programm führt den Benutzer:

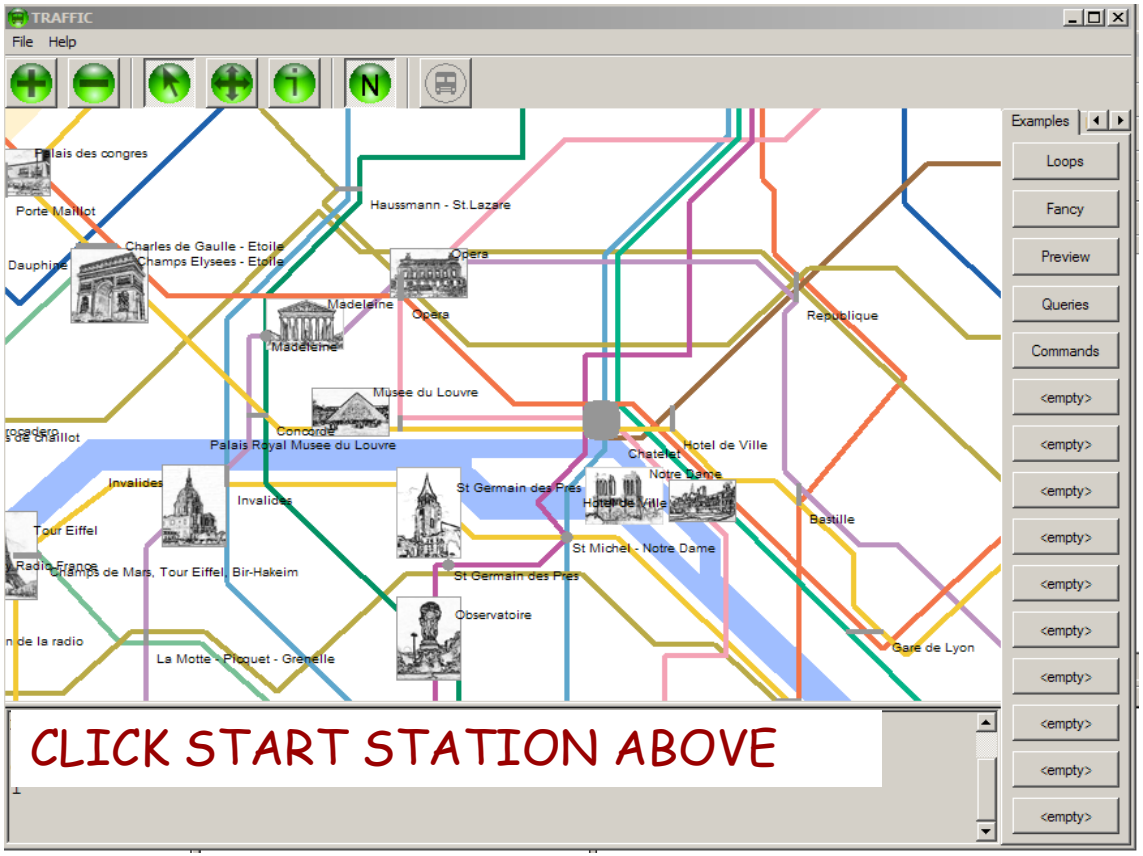
```
from
    i := 0
    read_line
until end_of_file loop
    i := i + 1
    Result [i] := last_line
    read_line
end
```



Input verarbeiten mit modernen GUIs

Der Benutzer führt das Programm:

" Wenn ein Benutzer diesen Knopf drückt, führe diese Aktion in meinem Programm aus."

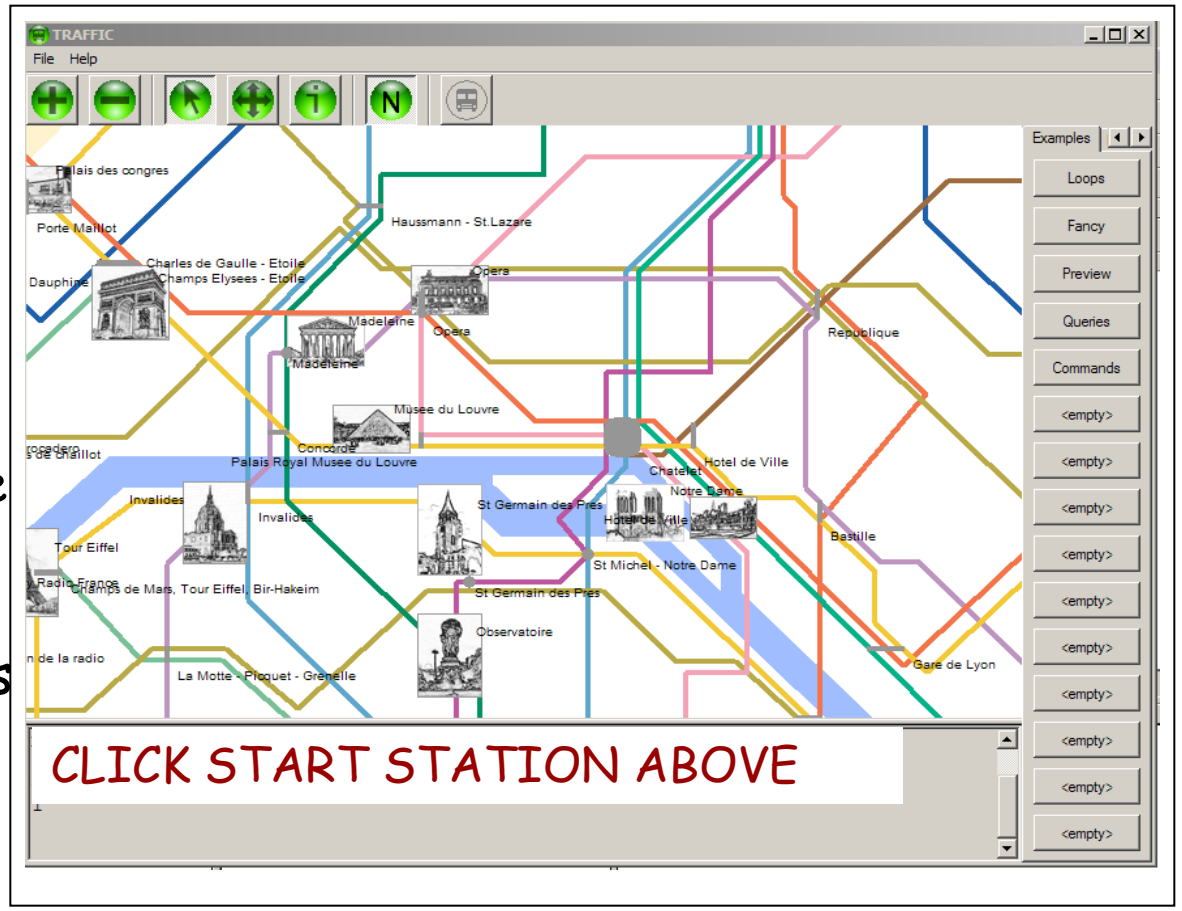


Ereignisorientierte Programmierung: Beispiel

Spezifizieren Sie, dass, wenn ein Benutzer diesen Knopf drückt, das System

find_station (*x*, *y*)

ausführt, wobei *x* und *y* die Mauskoordinaten sind und *find_station* eine spezifische Prozedur Ihres Systems ist.



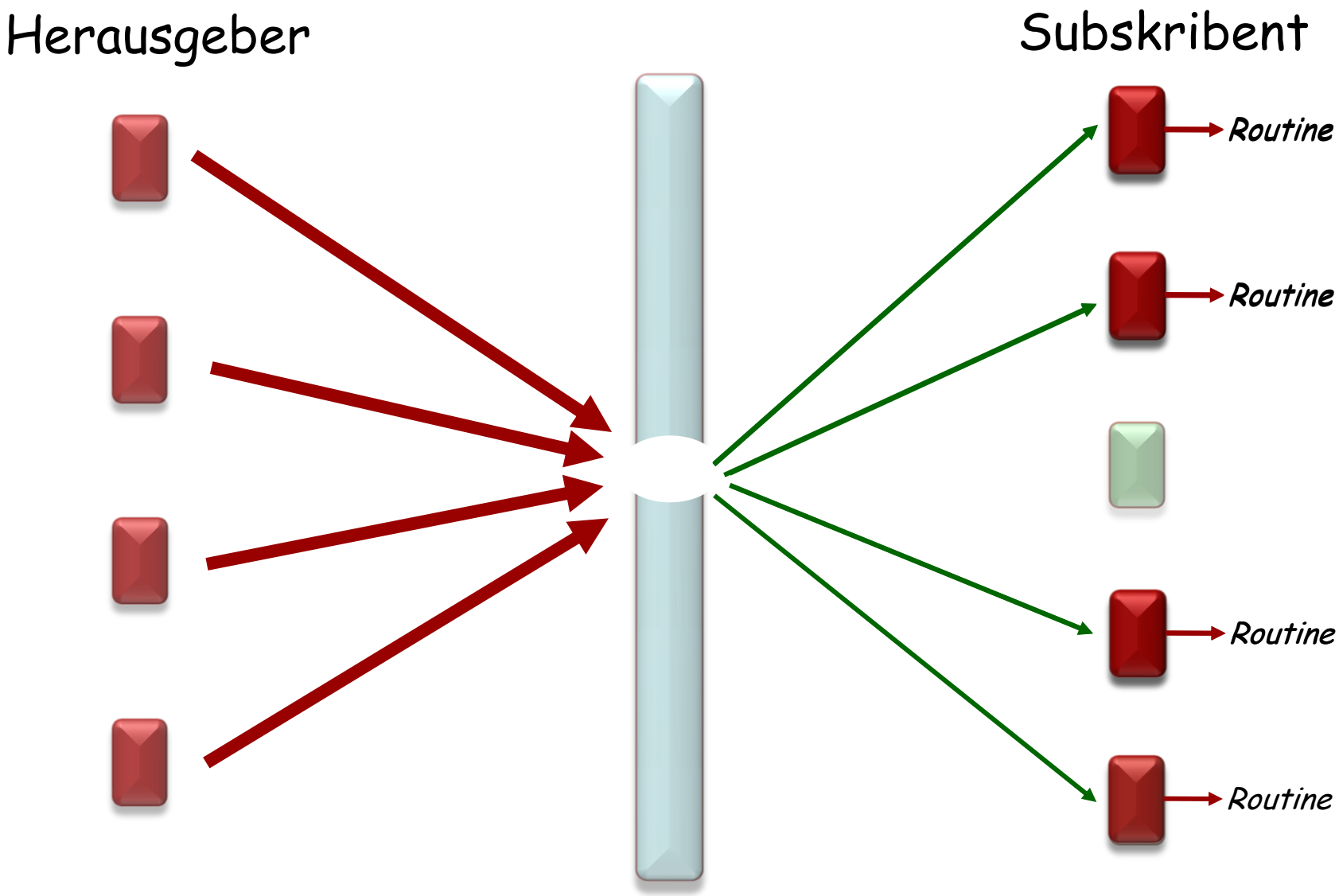
Einige Schwierigkeiten

1. Das „Geschäftsmodell" und das GUI getrennt halten.
   - Geschäftsmodell (oder einfach nur *Modell*): Kernfunktionalitäten der Applikation
   - GUI: Interaktion mit Benutzern

2. Den „Verbindungscode" zwischen den beiden minimieren.

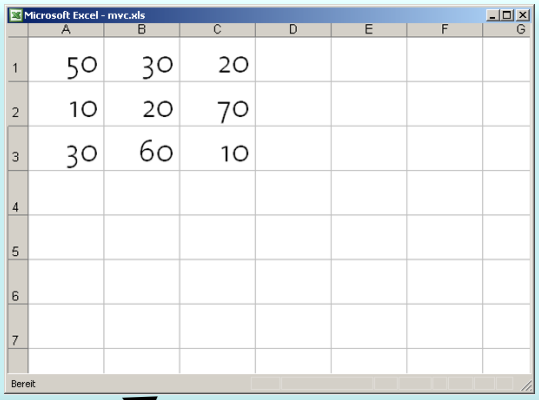
3. Sicherstellen, dass wir mitbekommen, was passiert.

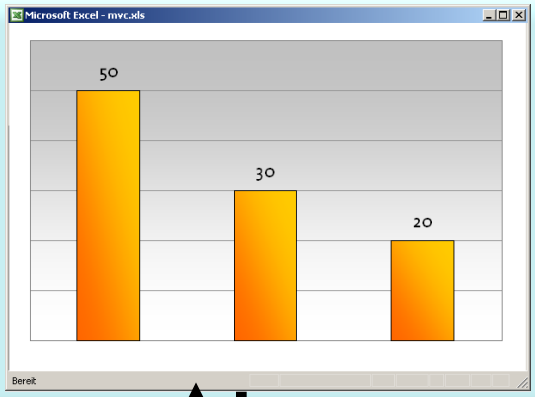
Ereignisorientierte Programmierung: Metapher

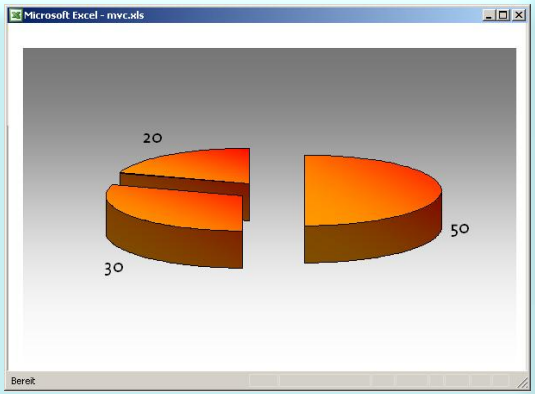


Einen Wert beobachten

Beobachter





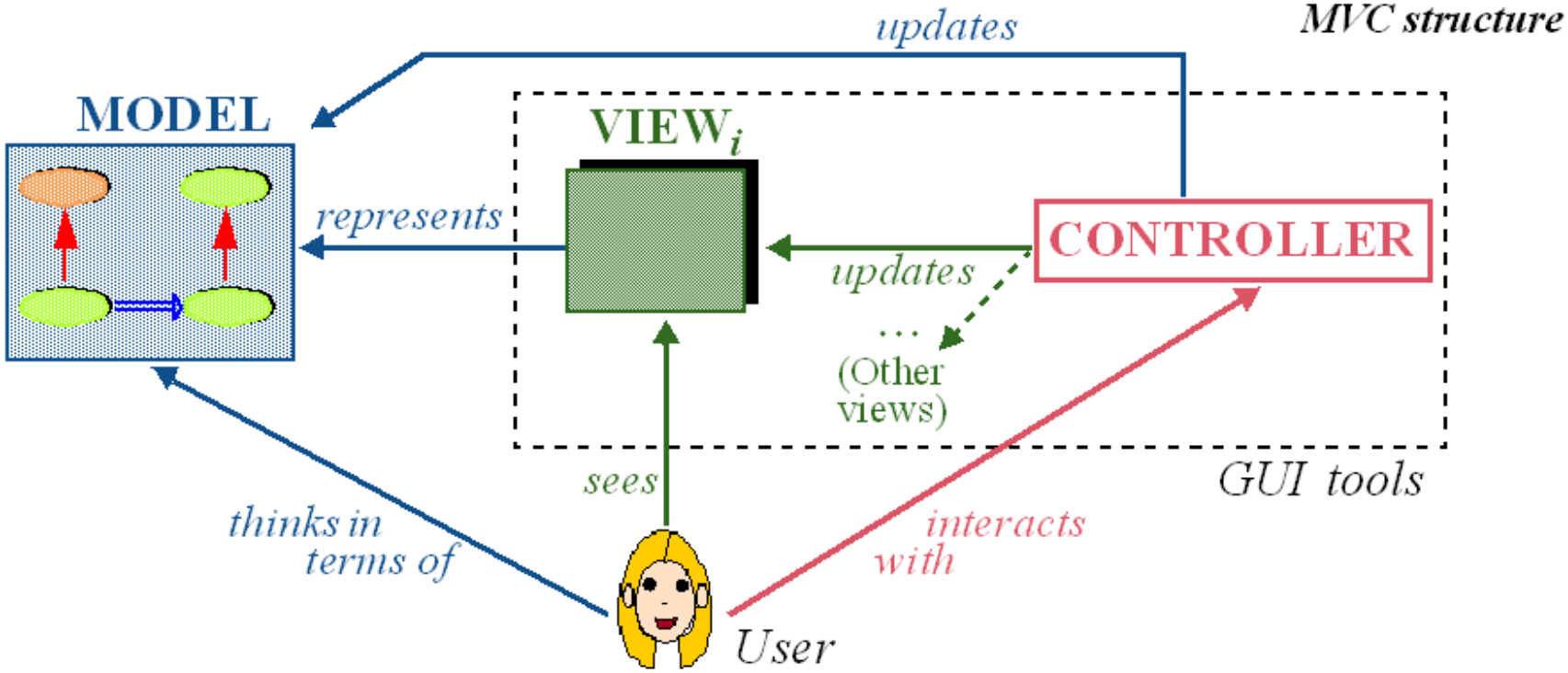


Subjekt

A = 50%
B = 30%
C = 20%

Model-View-Controller (Modell/Präsentation/Steuerung)

(Trygve Reenskaug, 1979)

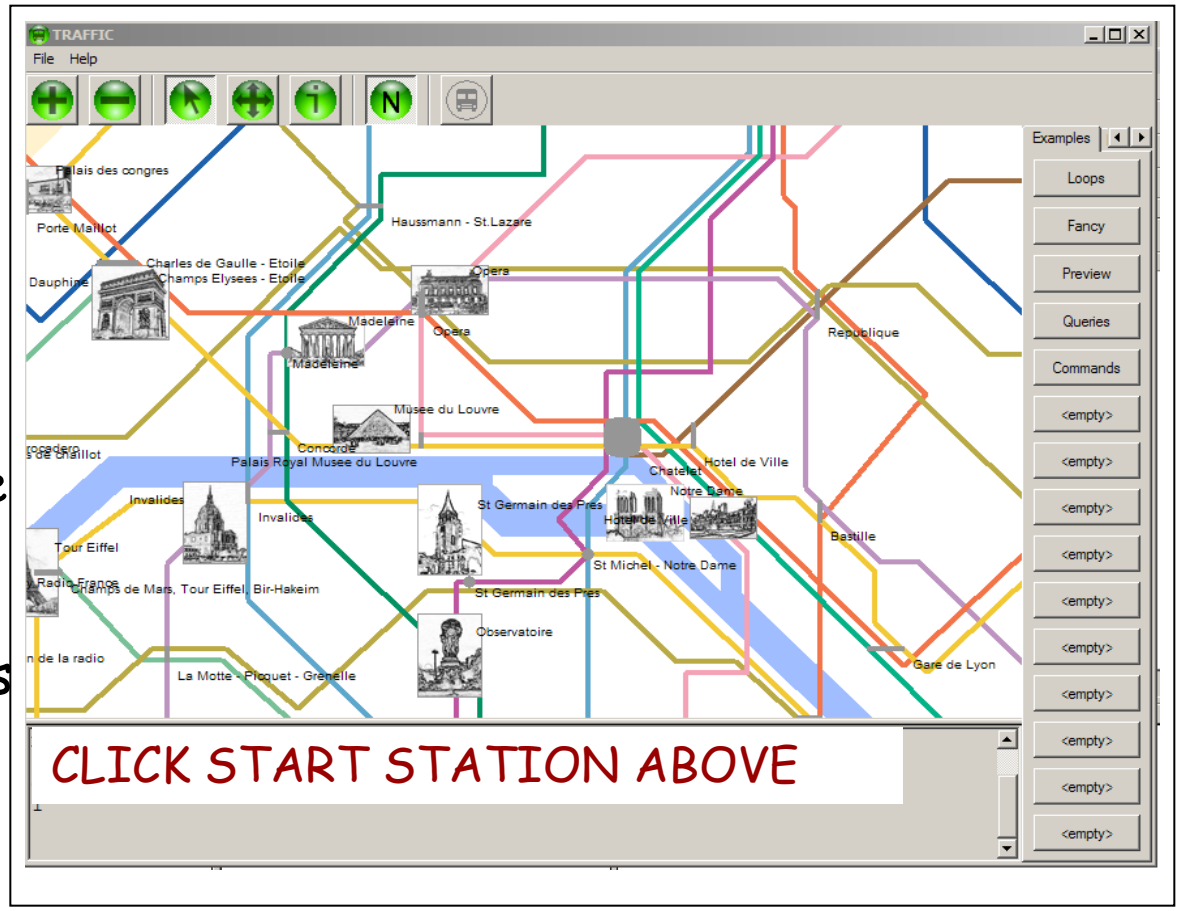


Unser Beispiel

Spezifizieren Sie, dass, wenn ein Benutzer diesen Knopf drückt, das System

find_station (*x*, *y*)

ausführt, wobei *x* und *y* die Mauskoordinaten sind und *find_station* eine spezifische Prozedur Ihres Systems ist.



Events Overview

Events have the following properties:

1. The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
2. An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
3. Events that have no subscribers are never called.
4. Events are commonly used to signal user actions such as button clicks or menu selections in graphical user interfaces.
5. When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see [another section].
6. Events can be used to synchronize threads.
7. In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class.

Ereignisse: Übersicht

Ereignisse haben folgende Eigenschaften:

1. Der Herausgeber bestimmt, wann ein Ereignis ausgelöst wird; die Subskribenten bestimmen, welche Aktion als Antwort darauf ausgeführt wird.
2. Ein Ereignis kann mehrere Subskribenten haben. Ein Subskribent kann mehrere Ereignisse von mehreren Herausgebern handhaben.
3. Ereignisse, die keinen Subskribenten haben, werden nie aufgerufen.
4. Ereignisse werden häufig benutzt, um Benutzeraktionen wie Knopfdrücke oder Menuselektionen in graphischen Benutzerschnittstellen zu signalisieren.
5. Wenn ein Ereignis mehrere Subskribenten hat, werden die Ereignishandler synchron aktiviert, wenn ein Ereignis ausgelöst wird. Um Ereignisse asynchron auszulösen, siehe [ein anderer Abschnitt]
6. Ereignisse können benutzt werden, um Threads zu synchronisieren.
7. In der .NET Framework-Klassenbibliothek basieren Ereignisse auf dem EventHandler Delegaten und der EventArgs Oberklasse.

Alternative Terminologien

In dieser Präsentation: **Herausgeber** und **Subskribent** (*Publisher & Subscriber*)

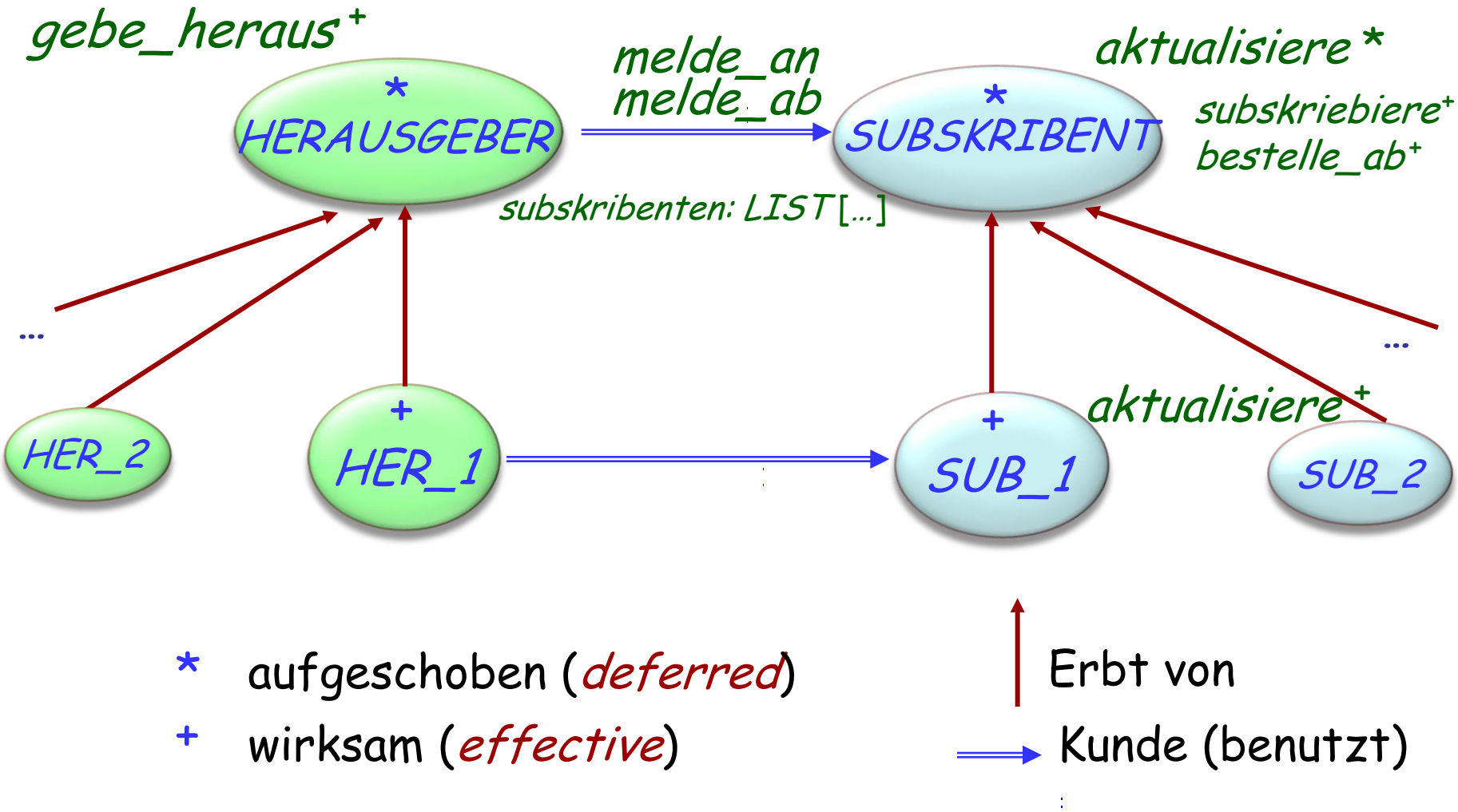
Beobachtete / Beobachter (*Observed / Observer*)

Subjekt (*Subject*) / Beobachter

Herausgeben / Subskriebieren (*Publish / Subscribe*)

Ereignisorientierters (*Event-Oriented*) Design & Programmieren

Eine Lösung: das Beobachter-Muster (Observer-Pattern)

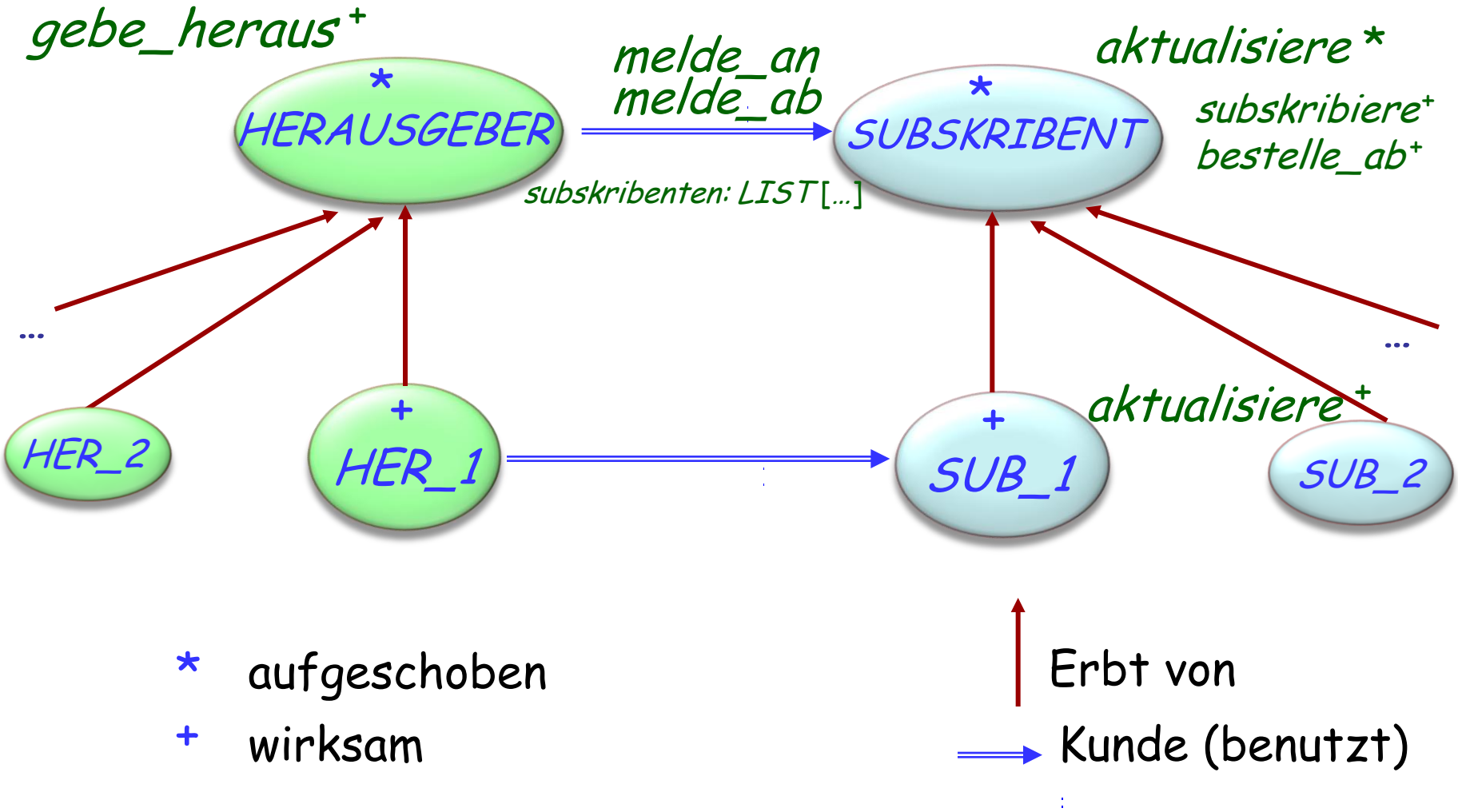


Entwurfsmuster (Design Pattern)

Ein Entwurfsmuster ist ein architektonisches Schema — eine Organisation von Klassen und Features — das Anwendungen standardisierten Lösungen für häufige Probleme bietet.

Seit 1994 haben verschiedene Bücher viele Entwurfsmuster vorgestellt. Am bekanntesten ist *Design Patterns* von Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley 1994.

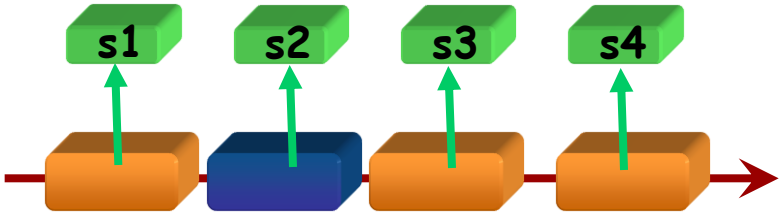
Eine Lösung: das Beobachter-Muster



Das Beobachter-Muster

Der Herausgeber unterhält eine (geheime) Liste von Beobachtern:

subskribenten : *LINKED_LIST*[*SUBSKRIBENT*]



Um sich zu registrieren, führt ein Beobachter

subskribiere (*ein_herausgeber*)

aus, wobei *subskribiere* in *SUBSKRIBENT* definiert ist:

```
subskribiere (h: HERAUSGEBER)
            -- Setze das aktuelle Objekt als Beobachter von h.
        require
            herausgeber_existiert: h /= Void
        do
            h.melde_an (Current)
        end
```

Einen Beobachter anbinden

In der Klasse *HERAUSGEBER*:

Wieso?

```
feature {SUBSKRIBENT}
    melde_an (a : SUBSKRIBENT)
            -- Registriere a als Subskribenten zu diesem Herausgeber.
        require
            subskribent_existiert: a /= Void
        do
            subskribenten.extend (a)
        end
```

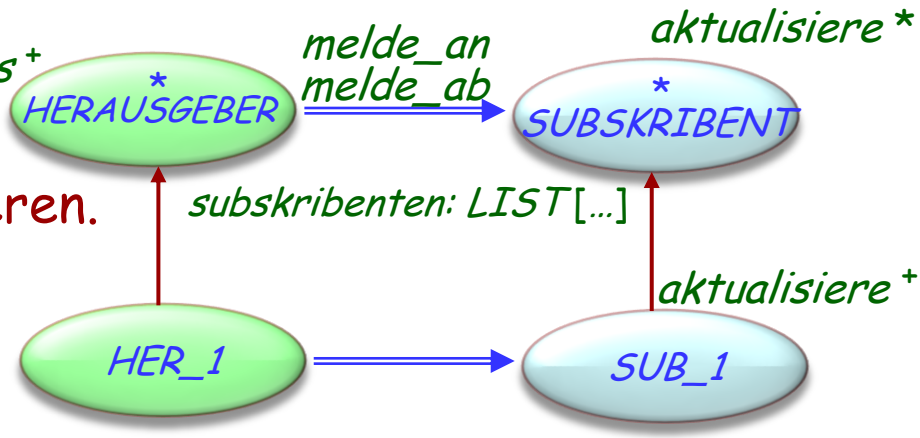
Beachten Sie, dass die *HERAUSGEBER* -Invariante die Klausel

```
subskribenten /= Void
```

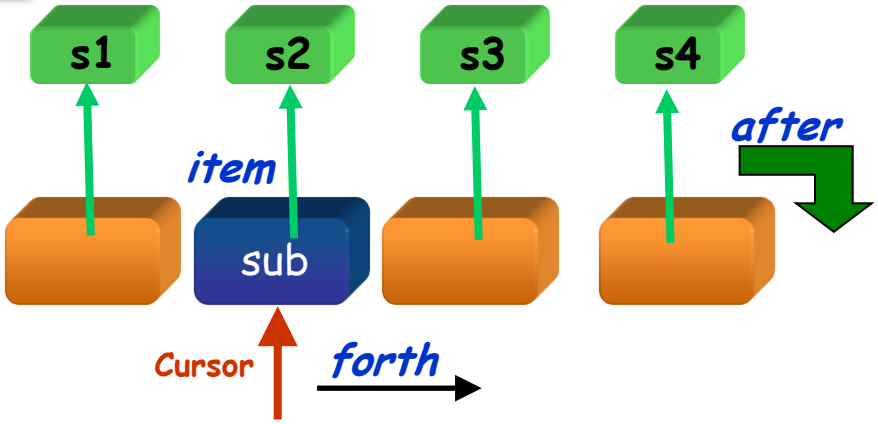
beinhaltet. (Die Liste *subskribenten* wird in den Erzeugungsprozeduren von *HERAUSGEBER* erzeugt.)

Einen Ereignis auslösen

```
gebe_heraus
        -- Lade alle Beobachter ein,
        -- auf diesem Ereignis zu reagieren.
    do
        from
            subskribenten.start
        until
            subskribenten.after
        loop
            subskribenten.item.aktualisiere
            subskribenten.forth
        end
    end
```



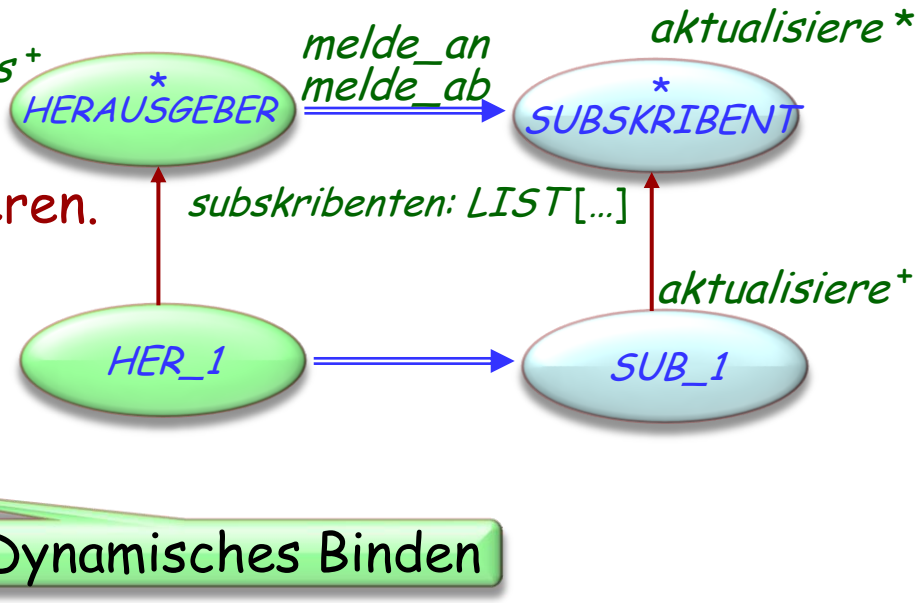
Dynamisches Binden

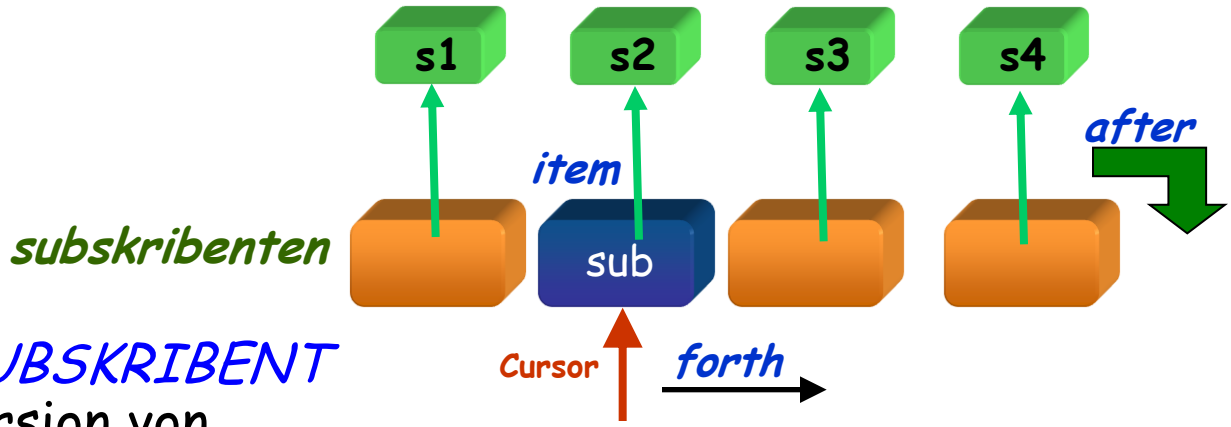


Jeder Nachkomme von *SUBSKRIBENT* definiert seine Eigene Version von *aktualisiere*

Einen Ereignis auslösen (eine Variante)

```
gebe_heraus
        -- Lade alle Beobachter ein,
        -- auf diesem Ereignis zu reagieren.
    do
        across subskribenten as a loop
            a.item.aktualisiere
        end
    end
```



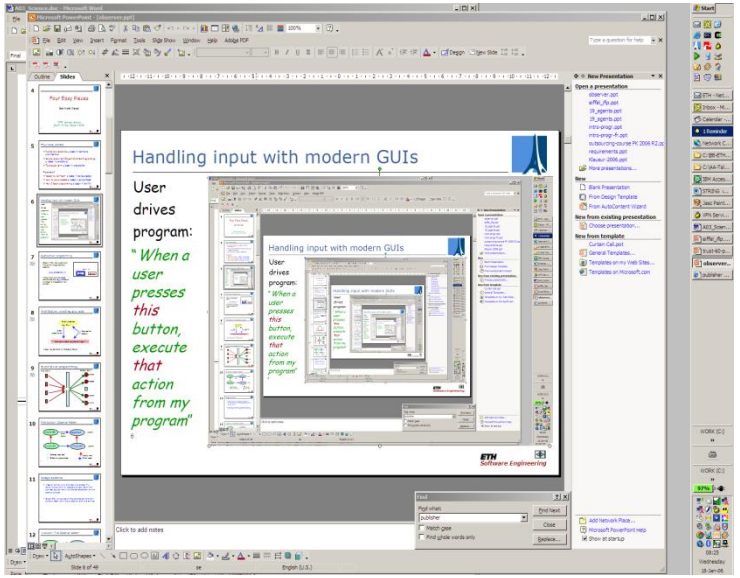


Jeder Nachkomme von *SUBSKRIBENT* definiert seine Eigene Version von *aktualisiere*

Das Beobachter-Muster (in der Grundform)

- Die Herausgeber kennen die Subskribenten.
- Jeder Subskribent kann sich nur bei maximal einem Herausgeber einschreiben.
- Können maximal eine Operation registrieren.
- Nicht wiederverwendbar — muss für jede Applikation neu programmiert werden.
- Argumente zu behandeln ist schwierig.

Anderer Ansatz: Ereignis-Kontext-Aktion-Tabelle



Eine Menge von „Trippeln“
[Ereignis-Typ, Kontext, Aktion]

Ereignis-Typ: irgendeine Art von Ereignis, an dem wir interessiert sind.
Beispiel: Linksklick

Kontext: Das Objekt, für welches diese Ereignisse interessant sind.
Beispiel: ein gewisser Knopf

Aktion: Was wir tun wollen, wenn der Ereignis im Kontext ausgelöst wird.
Beispiel: Speichern der Datei

Eine Ereignis-Kontext-Aktion-Tabelle kann z.B. mit Hilfe einer Hashtabelle implementiert werden.

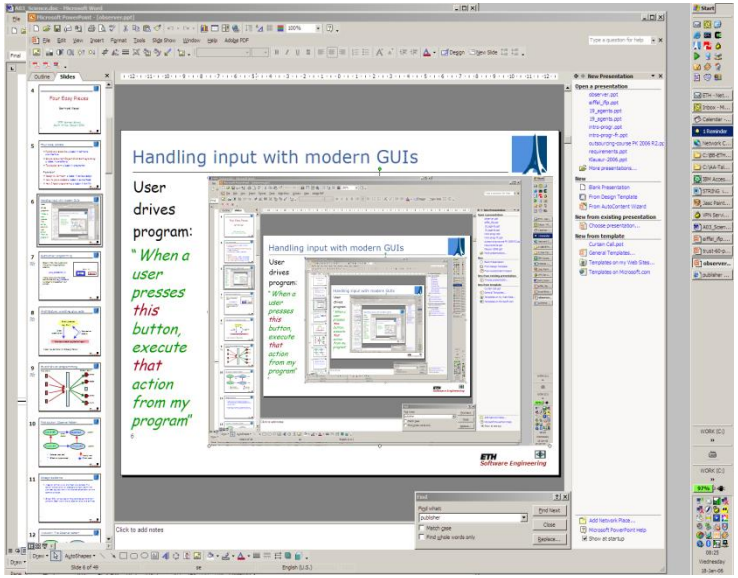
Ereignis-Aktion-Tabelle

Präziser: Ereignis_Typ – Aktion - Tabelle

Noch präziser: Ereignis_Typ - Kontext – Aktion-Tabelle

Ereignis-Typ	Kontext	Aktion
Left_click	Save_button	*Save_file*
Left_click	Cancel_button	*Reset*
Left_click	Map	*Find_station*
Left_click	…	…
Right_click	…	*Display_Menu*
…		…

Ereignis-Kontext-Aktion-Tabelle



Eine Menge von „Trippeln“
[Ereignis-Typ, Kontext, Aktion]

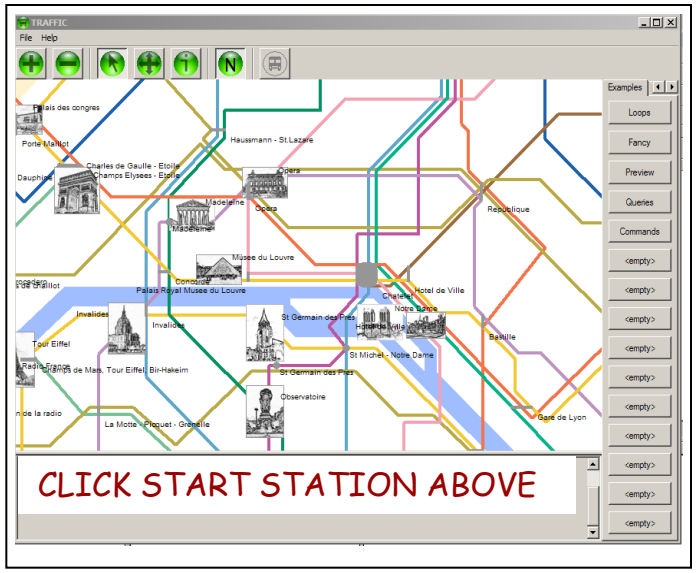
Ereignis-Typ: irgendeine Art von Ereignis, an dem wir interessiert sind.
Beispiel: Linksklick

Kontext: Das Objekt, für welches diese Ereignisse interessant sind.
Beispiel: ein gewisser Knopf

Aktion: Was wir tun wollen, wenn der Ereignis im Kontext ausgelöst wird.
Beispiel: Speichern der Datei

Eine Ereignis-Kontext-Aktion-Tabelle kann z.B. mit Hilfe einer Hashtabelle implementiert werden.

In EiffelVision



Paris_map.click.action_list.extend (**agent** *find_station*)

Mechanismen in anderen Sprachen

C und C++: “Funktionszeiger”

C#: Delegaten (eine limitiertere Form von Agenten)

Bemerkung zu Sprachen

In nicht-O-O Sprachen wie z.B. C und Matlab gibt es den Begriff der Agenten nicht, aber man kann eine Routine als Argument an eine andere Routine übergeben, z.B.

integral (&*f*, *a*, *b*)

wobei *f* eine zu integrierende Funktion ist. &*f* (C Notation) ist eine Art, sich auf die Funktion *f* zu beziehen.

- (Wir brauchen eine solche Syntax, da nur `*f*' auch ein Funktionsaufruf sein könnte.)

Agenten (oder C# Delegaten) bieten eine Typ-sichere Technik auf hoher Ebene, indem sie die Routine in ein Objekt verpacken.

Mit .NET-Delegates: Herausgeber (1)

P1. Einführung einer neuen Klasse *ClickArgs* , die von *EventArgs* erbt und die Argument-Typen von *yourProcedure* wiederholt:

public class *Clickargs* {... *int x, y*; ...}

P2. Einführung eines neuen Typs *ClickDelegate* (Delegate-Typ), basierend auf dieser Klasse.

public void *delegate ClickDelegate* (*Object sender, Clickargs e*);

P3. Deklarieren eines neuen Typs *Click* (Ereignis-Typ), basierend auf dem Typ *ClickDelegate*:

public event *ClickDelegate Click*;

Mit .NET-Delegates: Herausgeber (2)

P4. Schreiben einer neuen Prozedur *OnClick*, um das Handling zu verpacken:

```
protected void OnClick (Clickargs c)
    {if (Click != null) {Click (this, c);}}
```

P5. Für jedes mögliche Auftreten: Erzeuge ein neues Objekt (eine Instanz von *ClickArgs*), die die Argumente dem Konstruktor übergibt:

```
ClickArgs yourClickargs = new Clickargs (h, v);
```

P6. Für jedes Auftreten eines Ereignisses: Löse den Ereignis aus:

```
OnClick (yourClickargs);
```

Mit .NET-Delegates: Subskribent

D1. Deklarieren eines Delegaten *myDelegate* vom Typ *ClickDelegate*.
(Meist mit dem folgenden Schritt kombiniert.)

D2. Instantiieren mit *yourProcedure* als Argument:

```
myDelegate = new ClickDelegate (yourProcedure);
```

D3. Hinzufügen des Delegaten zur Liste für den Ereignis:

```
YES_button.Click += myDelegate;
```

Der Eiffel-Ansatz (Event Library)

Ereignis: Jeder Ereignis-*Typ* wird ein Objekt sein.

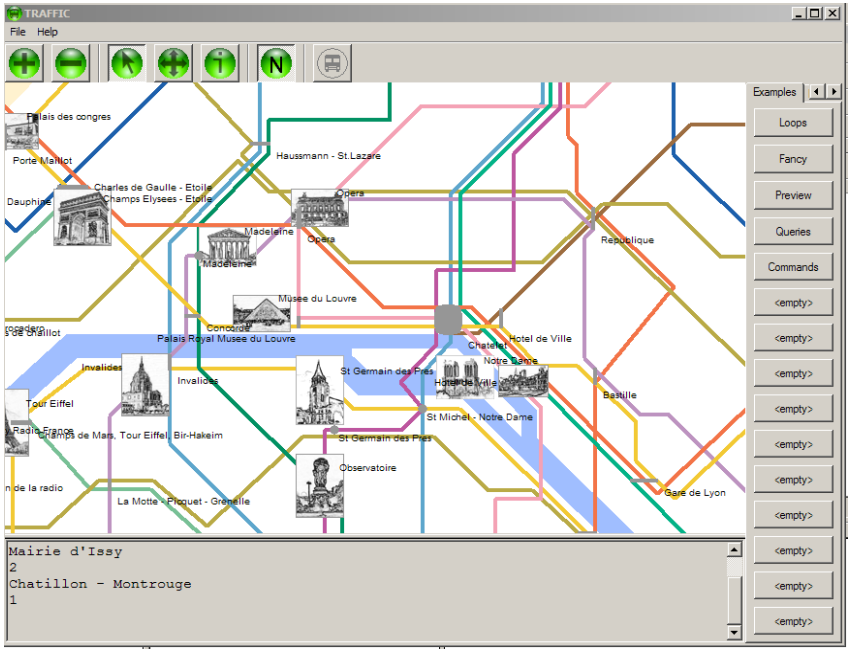
Beispiel: Linksklick

Kontext: Ein Objekt, das meistens ein Element der Benutzeroberfläche repräsentiert.

Beispiel: die Karte

Aktion: Ein Agent, der eine Routine repräsentiert.

Beispiel: *find_station*



Die Ereignis-Bibliothek

Grundsätzlich:

- Eine generische Klasse: *EVENT_TYPE*
- Zwei Features: *publish* und *subscribe*

Zum Beispiel: Ein Kartenwidget *Paris_map*, welches in einer in *find_station* definierten Art reagiert, wenn es angeklickt wird (Ereignis *left_click*).

Der Stil der Ereignis-Bibliothek

Die grundlegende Klasse ist *EVENT_TYPE*

Auf der Herausgeber-Seite, z.B. GUI-Bibliothek:

- (Einmaliges) Deklarieren eines Ereignis-Typs:

  click: *EVENT_TYPE* [*TUPLE* [*INTEGER, INTEGER*]]

- (Einmaliges) Erzeugen eines Ereignis-Typ Objektes:

  create *click*

- Um ein Auftreten des Ereignisses auszulösen:

  click.*publish* ([*x_coordinate, y_coordinate*])

Auf der Subskribent-Seite, z.B. eine Applikation:

click.*subscribe* (**agent** *find_station*)

Beispiel mit Hilfe der Ereignis-Bibliothek

Die Subskribenten (Beobachter) registrieren sich bei Ereignissen:

```
Paris_map.left_click.subscribe (agent find_station)
```

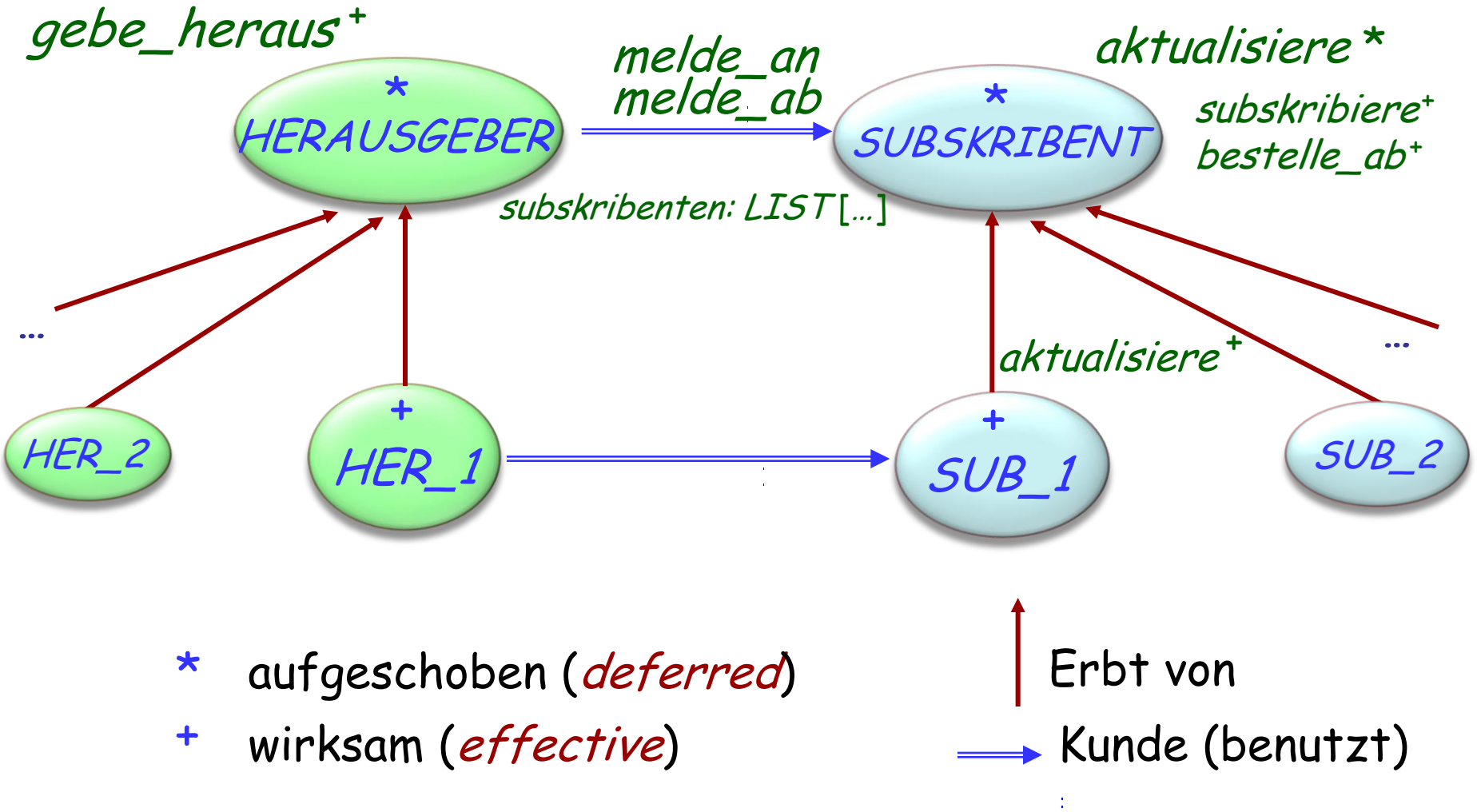
Der Herausgeber (Subjekt) löst einen Ereignis aus:

```
left_click.publish ([x_positition, y_position])
```

Jemand (normalerweise der Herausgeber) definiert den Ereignis-Typ:

```
left_click: EVENT_TYPE [TUPLE [INTEGER, INTEGER]]
            -- „Linker Mausklick"-Ereignisse
    once
            create Result
    ensure
            exists: Result /= Void
    end
```

Erinnerung: Observer-Pattern



Vergleich: Beobachter-Muster und Ereignis-Bibliothek

Im Falle einer existierenden Klasse *MEINE_KLASSE*:

- Mit dem Beobachter-Muster:
  - Bedingt das Schreiben von Nachkommen von *Subskribent* und *MEINE_KLASSE*
  - Unnötige Vervielfachung von Klassen

- Mit der Ereignis-Bibliothek:
  - Direkte Wiederverwendung von existierenden Routinen als Agenten

Varianten des Registrierens

click.subscribe (**agent** *find_station*)

Paris_map.click.subscribe (**agent** *find_station*)

click.subscribe (**agent** *your_procedure* (*a*, *?*, *?*, *b*))

click.subscribe (**agent** *other_object.other_procedure*)

Tupel

Tupel-Typen (für irgendwelche Typen *A*, *B*, *C*, ...):

TUPLE
TUPLE [A]
TUPLE [A, B]
TUPLE [A, B, C]
...

Ein Tupel des Typs *TUPLE [A, B, C]* ist eine Sequenz von mindestens drei Werten, der Erste von Typ *A*, der Zweite von Typ *B*, der Dritte von Typ *C*.

Tupelwerte: z.B.

[a1, b1, c1, d1]

Benannte Tupel-Typen

TUPLE [*autor*: *STRING*; *jahr*: *INTEGER*; *titel*: *STRING*]

Eine beschränkte Form einer Klasse

Ein benannter Tupel-Typ bezeichnet den gleichen Typ wie eine unbenannte Form, hier

TUPLE [*STRING*, *INTEGER*, *STRING*]

aber er vereinfacht den Zugriff auf einzelne Elemente.

Um ein bestimmtes Tupel (benannt oder unbenannt) zu bezeichnen:

["*Tolstoj*", *1865*, "*Krieg und Frieden*"]

Um ein Tupelelement zu erreichen: z.B. *t.jahr*

Tupel: Prozeduren und Funktionen

Auf einen Agenten *a* anwendbare Features:

- Falls *a* eine Prozedur repräsentiert, ruft die Prozedur auf:

```
a.call(argument_tupel)
```

z.B. ["Tolstoj", 1865, "Krieg und Frieden"]

- Falls *a* eine Funktion repräsentiert, ruft die Funktion auf und gibt ihr Resultat zurück:

```
a.item(argument_tupel)
```

Was Sie mit einem Agenten *a* tun können

Aufrufen der assoziierten Routine durch das Feature *call*,
dessen Argument ein einfaches Tupel ist:

Ein manifestes Tupel

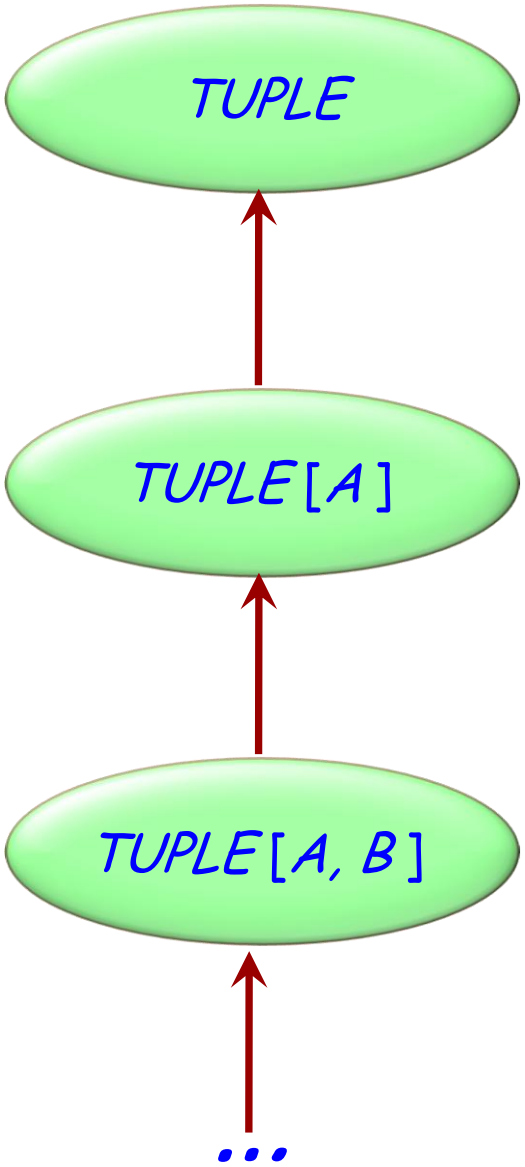
```
a.call ([horizontal_position, vertical_position])
```

Falls *a* mit einer Funktion assoziiert ist, gibt

```
a.item ([ ..., ...])
```

das Resultat der Anwendung der Funktion zurück.

Die Vererbungshierarchie von Tupeln



Die Ereignis-Bibliothek benutzen

Die grundlegende Klasse ist *TRAFFIC_EVENT_CHANNEL*

Auf der Herausgeber-Seite, z.B. GUI-Bibliothek:

- (Einmaliges) Deklarieren eines Ereignis-Typs:

  click: TRAFFIC_EVENT_CHANNEL

  [TUPLE [INTEGER, INTEGER]]

- (Einmaliges) Erzeugen eines Ereignis-Typ Objektes:

  create *click*

- Um ein Auftreten des Ereignisses auszulösen:

  click.publish ([x_coordinate, y_coordinate])

Auf der Subskribent-Seite, z.B. eine Applikation:

click.subscribe (**agent** *find_station*)

Argumente offen lassen

Ein Agent kann sowohl „geschlossene" als auch „offene" Argumente haben.

Geschlossene Argumente werden zur Zeit der Definition des Agenten gesetzt, offene Argumente zur Zeit des Aufrufs.

Um ein Argument offen zu lassen, ersetzt man es durch ein Fragezeichen:

```
u := agent a0.f(a1, a2, a3) -- Alle geschlossen (bereits gesehen)
w := agent a0.f(a1, a2, ?)
x := agent a0.f(a1, ?, a3)
y := agent a0.f(a1, ?, ?)
z := agent a0.f(?, ?, ?)
```

Den Agenten aufrufen

```
f(x1: T1; x2: T2; x3: T3)
a0: C; a1: T1; a2: T2; a3: T3
```

`u := agent a0.f(a1, a2, a3)`	`u.call([])`
`v := agent a0.f(a1, a2, ?)`	`v.call([a3])`
`w := agent a0.f(a1, ?, a3)`	`w.call([a2])`
`x := agent a0.f(a1, ?, ?)`	`x.call([a2, a3])`
`y := agent a0.f(?, ?, ?)`	`y.call([a1, a2, a3])`

Ein weiteres Beispiel zum Aufruf von Agenten

$$\int_a^b meine\_funktion(x)\, dx$$

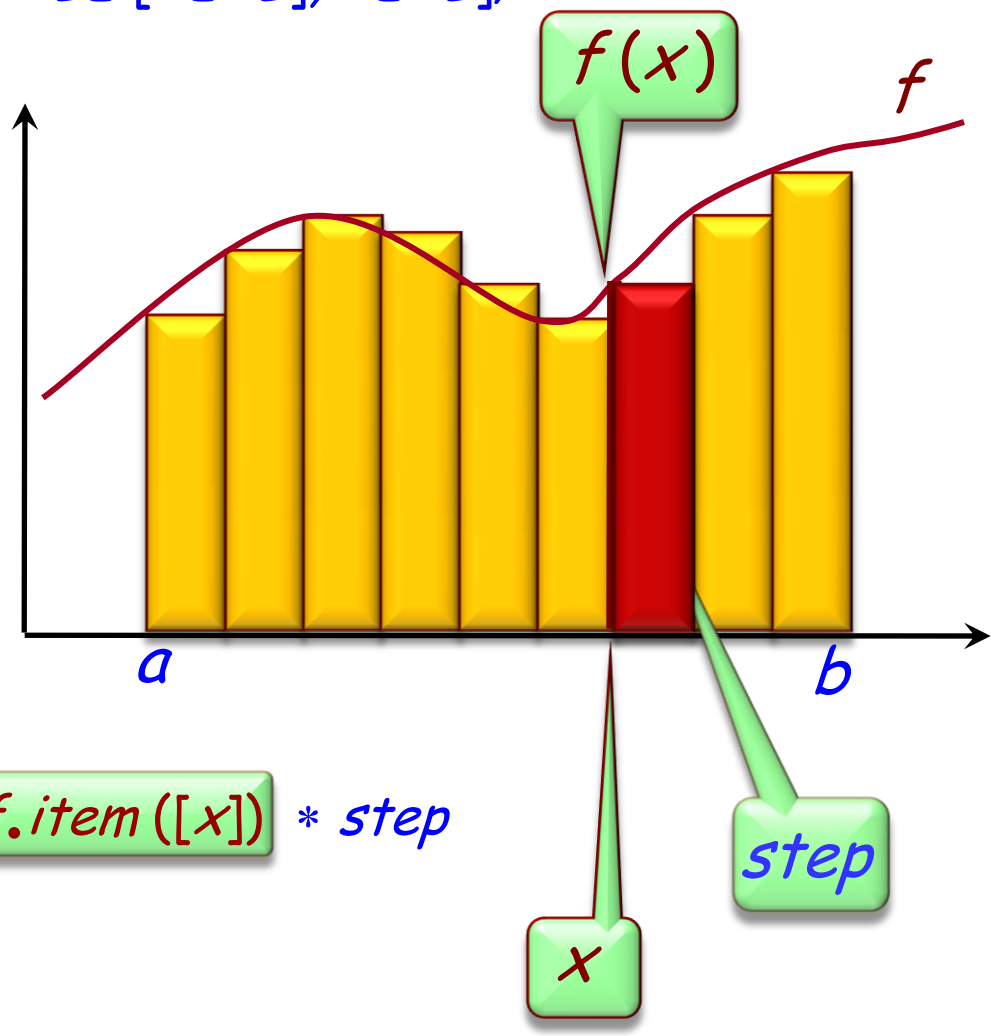
$$\int_a^b ihre\_funktion(x, u, v)\, dx$$

my_integrator.*integral*(**agent** *meine_funktion* , *a*, *b*)

my_integrator.*integral*(**agent** *ihre_funktion*(? , *u*, *v*), *a*, *b*)

Die Integrationsfunktion

```
integral (f : FUNCTION [ANY, TUPLE [REAL], REAL];
          a, b : REAL): REAL

        -- Integral von f
        -- über Intervall [a, b]

    local
        x: REAL; i : INTEGER

do

        from x := a until x > b loop

                Result := Result + f.item ([x]) * step
                i := i + 1
                x := a + i * step

        end

    end
```



Weitere Anwendung: Benutzen eines Interators

```
class C feature
    all_positive, all_married: BOOLEAN

    is_positive (n: INTEGER) : BOOLEAN
            -- Ist n grösser als null?
        do Result := (n > 0) end

    intlist: LIST[INTEGER]
    emplist: LIST[EMPLOYEE]

    r
        do
            all_positive := intlist.for_all (agent is_positive (?) )
            all_married := emplist.for_all (agent {EMPLOYEE} is_married )
        end
end
```

```
class EMPLOYEE feature
        is_married: BOOLEAN
        ...
end
```

Iteratoren

In der Klasse *LINEAR [G]*, dem Vorfahren aller Klassen für Listen, Sequenzen, etc., finden Sie:

- *for_all*
- *there_exists*
- *do_all*
- *do_if*
- *do_while*
- *do_until*

Anwendungen von Agenten

Entwurfsmuster: Beobachter (Beobachter), Visitor, Undo-redo (Command-Pattern)

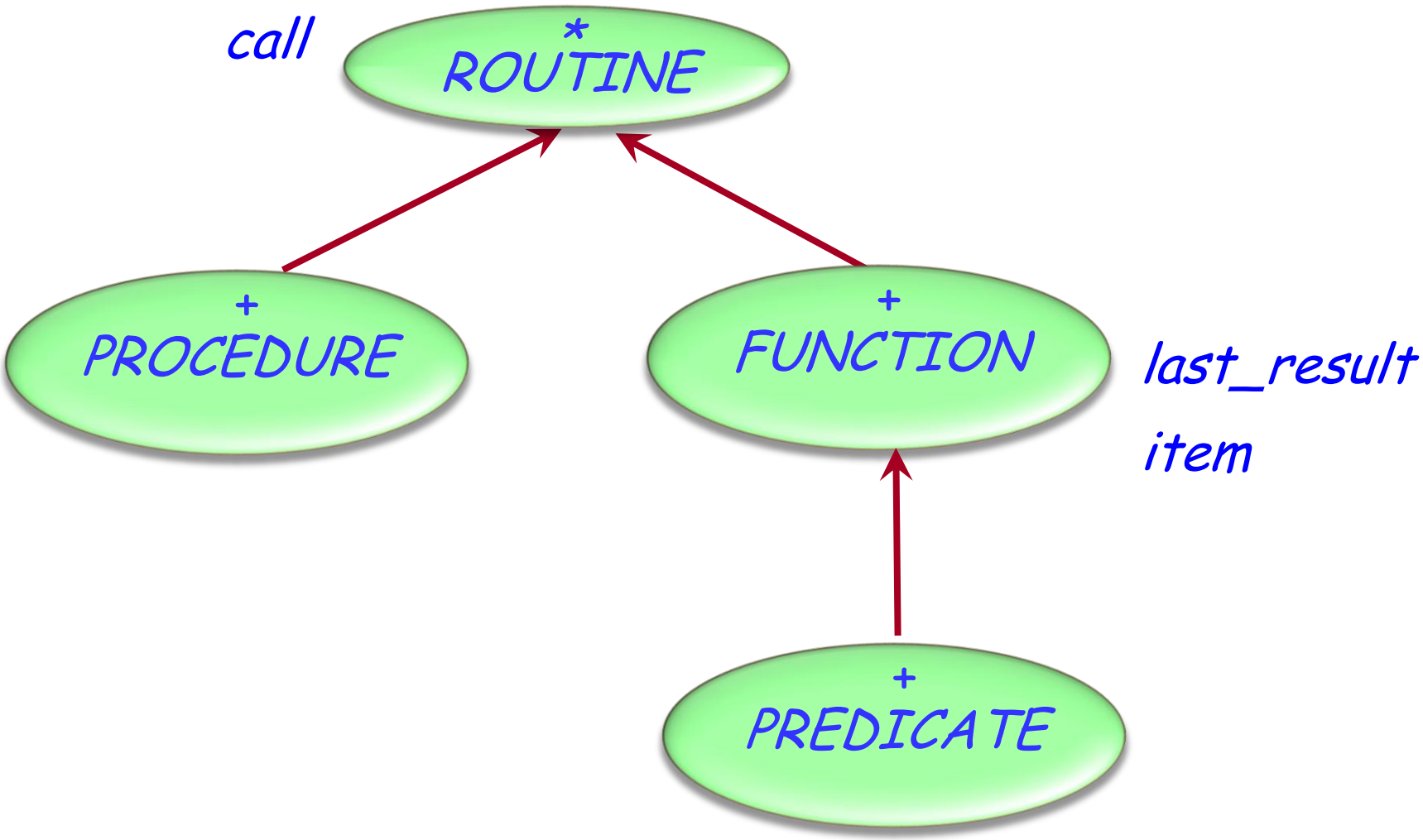
Iteration

Verträge auf einer hohen Ebene

Numerische Programmierung

Introspektion (die Eigenschaften eines Programmes selbst herausfinden)

Agenten: Klassen- Repräsentation in der Kernel-Bibliothek



Einen Agenten deklarieren

```
p: PROCEDURE [ANY, TUPLE]
    -- Ein Agent, der eine Prozedur repräsentiert,
    -- keine offenen Argumente

q: PROCEDURE [ANY, TUPLE [X, Y, Z]]
    -- Agent, der eine Prozedur repräsentiert,
    -- 3 offene Argumente

f: FUNCTION [ANY, TUPLE [X, Y, Z], RES]
    -- Agent, der eine Prozedur repräsentiert,
    -- 3 offene Argumente, Resultat vom Typ RES
```

Einen Agenten aufrufen

```
f(x1: T1; x2: T2; x3: T3)
a0: C; a1: T1; a2: T2; a3: T3
```

`u := agent a0.f(a1, a2, a3)`	`u.call([])`
`v := agent a0.f(a1, a2, ?)`	`v.call([a3])`
`w := agent a0.f(a1, ?, a3)`	`w.call([a2])`
`x := agent a0.f(a1, ?, ?)`	`x.call([a2, a3])`
`y := agent a0.f(?, ?, ?)`	`y.call([a1, a2, a3])`

Noch eine Anwendung: Undo/Redo

Bisherige Lösung: Vererbung und eine polymorphe Liste
(in den nächsten paar Folien zusammengefasst)

Referenzen:

- Kapitel 21 meines Object-Oriented Software Construction, Prentice Hall, 1997
- Erich Gamma et al., *Design Patterns,* Addison – Wesley, 1995: "Command pattern"

Die Problemstellung

Dem Benutzer eines interaktiven Systems die Möglichkeit geben, die letzte Aktion rückgängig zu machen.

Bekannt als "**Control-Z**"

Sollte mehrstufiges rückgängig Machen ("**Control-Z**") und Wiederholen ("**Control-Y**") ohne Limitierung unterstützen, ausser der Benutzer gibt eine maximale Tiefe an.

In unserem Beispiel: Ein Texteditor

Begriff der „aktuellen Zeile“ mit folgenden Befehlen:

- Löschen der aktuellen Zeile
- Ersetzen der aktuellen Zeile mit einer anderen
- Einfügen einer Zeile vor der aktuellen Position
- Vertauschen der aktuellen Zeile mit der nächsten (falls vorhanden)
- „Globales Suchen und Ersetzen“ (fortan GSE): Jedes Auftreten einer gewissen Zeichenkette durch eine andere ersetzen.
- ...

Dies ist der Einfachheit halber eine Zeilen-orientierte Ansicht, aber die Diskussion kann auch auf kompliziertere Ansichten angewendet werden.

Eine einfache Lösung

Sichern des gesamten Zustandes vor jeder Operation.

Im Beispiel: Der Text, der bearbeitet wird und die aktuelle Position im Text.

Wenn der Benutzer ein „Undo“ verlangt, stelle den zuletzt gesicherten Zustand wieder her.

Aber: Verschwendung von Resourcen, im Speziellen Speicherplatz.

Intuition: Sichere nur die Änderungen (diff) zwischen zwei Zuständen.

Entwerfen einer Software-Architektur: Schlüsselschritt

Die richtigen Abstraktionen finden

(die interessanten Objekt-Typen)

Hier:

Der Begriff eines "Befehls"

Die „Geschichte“ einer Sitzung speichern

Die Geschichte-Liste:

history : *TWO_WAY_LIST* [*COMMAND*]

Was ist ein “Command”-Objekt?

Ein Command-Objekt beinhaltet genügend Informationen über eine Ausführung eines Befehls durch den Benutzer, um

- den Befehl auszuführen
- den Befehl rückgängig zu machen

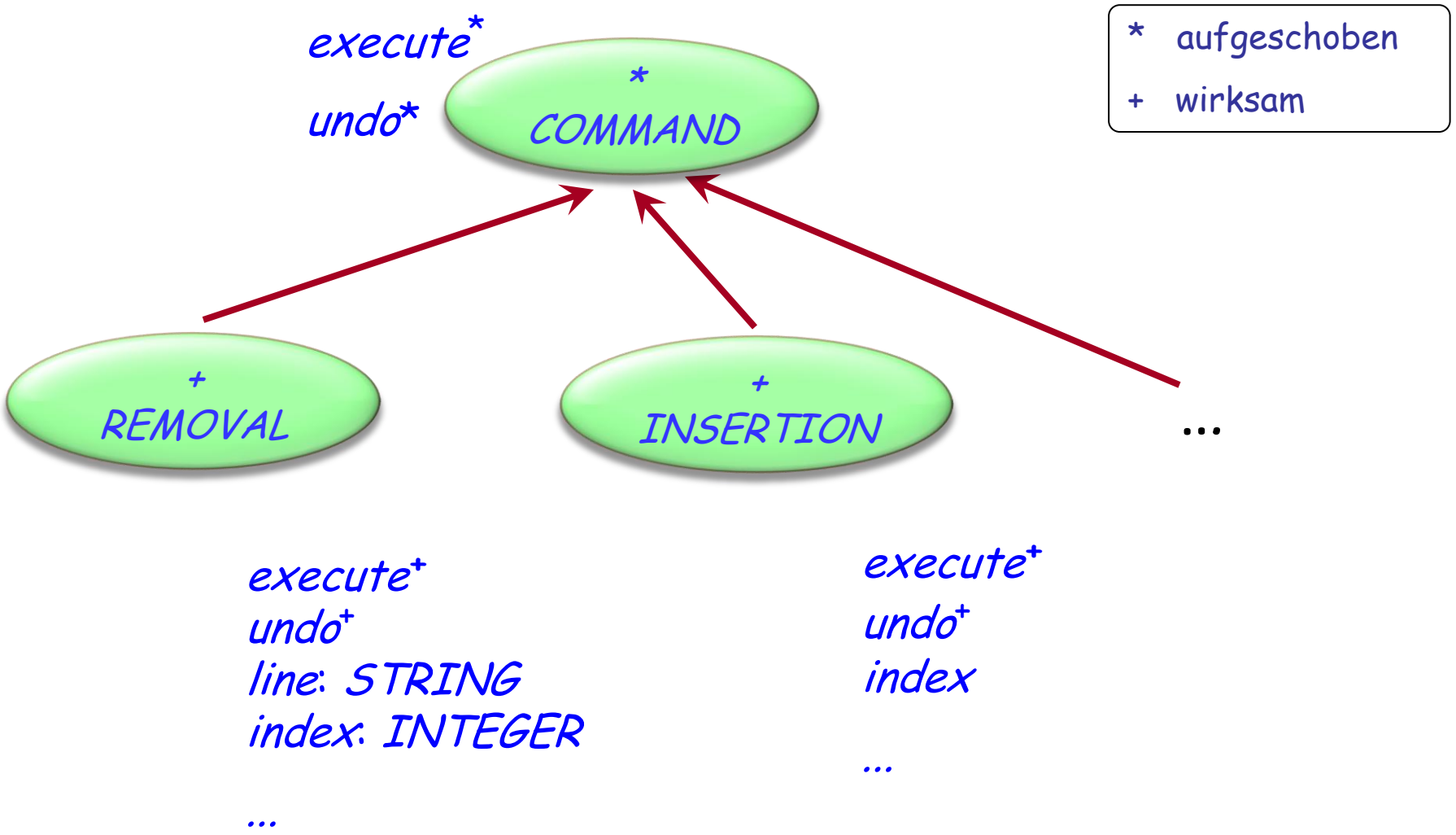
Beispiel: In einem “**Löschungs**”-Objekt brauchen wir:

- Die Position der zu löschenden Zeile
- Der Inhalt dieser Zeile!

Allgemeiner Begriff eines Befehls

```
deferred class  COMMAND  feature
    done: BOOLEAN
            -- Wurde dieser Befehl ausgeführt?
    execute
            -- Befehl einmal ausführen.
        deferred
        ensure
            already: done
        end
    undo
            -- Eine frühere Ausführung des Befehls
            --  rückgängig machen.
        require
            already: done
        deferred
        end
end
```

Die Command-Klassenhierarchie



Zugrundeliegende Klasse (Geschäftsmodell)

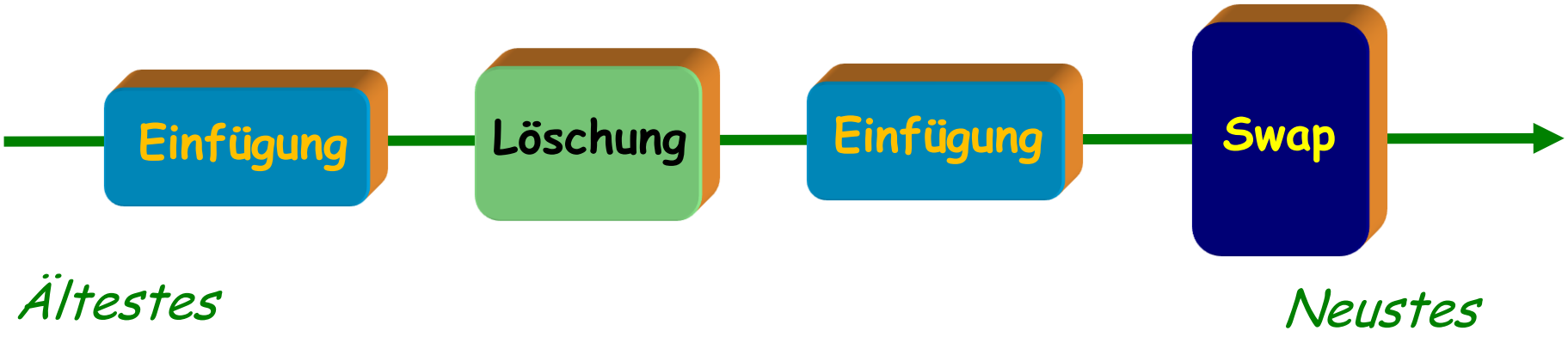
```
class EDIT_CONTROLLER feature

        text : TWO_WAY_LIST[STRING]

        remove
                        -- Linie an aktueller Position löschen.
                require
                        not off
                do
                        text.remove
                end

        put_right (line : STRING)
                        -- line  nach der aktuellen Position einfügen.
                require
                        not after
                do
                        text.put_right (line)
                end

        ... Auch: item, index, go_ith, put_left ...

end
```

Eine Befehlsklasse (Skizze, ohne Verträge)

```
class REMOVAL inherit COMMAND feature
        controller: EDIT_CONTROLLER
                        -- Zugriff aufs Geschäftsmodell
        line: STRING
                        -- Zu löschende Linie
        index: INTEGER
                        -- Position der zu löschenden Linie

        execute
                        -- Lösche aktuelle Linie und speichere sie.
                do      line := controller.item ; index := controller.index
                        controller.remove   ; done := True
                end

        undo
                        -- Wieder-Einfügung einer vorher gelöschten Linie.
                do      controller.go_i_th (index)
                        controller.put_left (line)
                end
end
```

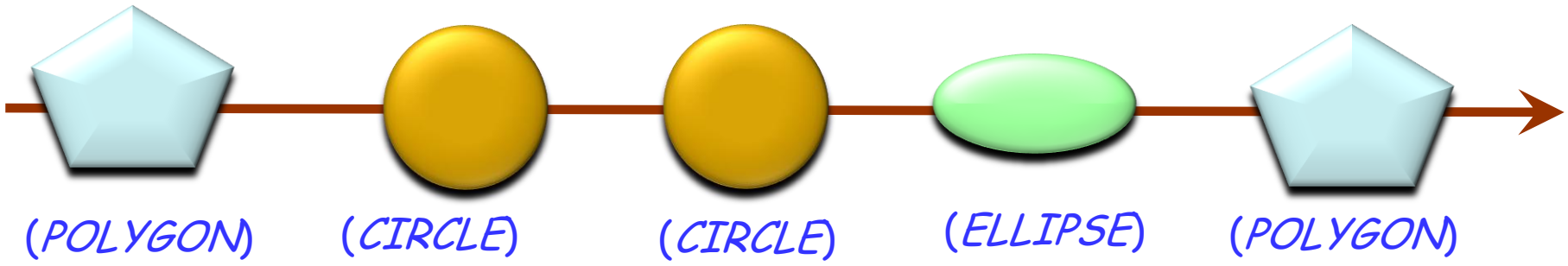
Die Geschichte-Liste

Eine polymorphe Datenstruktur:



history : *TWO_WAY_LIST* [*COMMAND*]

Erinnerung: Die Figurenliste



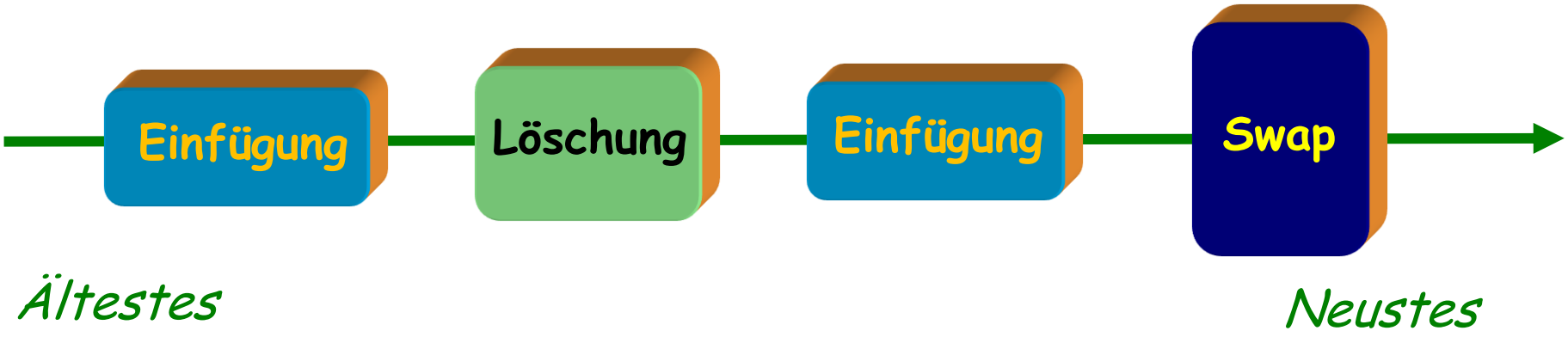

```
figs.extend(p1); figs.extend(c1); figs.extend(c2)
figs.extend(e); figs.extend(p2)
```

```
figs: LIST [FIGURE]
p1, p2: POLYGON
c1, c2: CIRCLE
e: ELLIPSE
```

```
class LIST[G] feature
        extend(v: G) do …
end
        last: G
        …
end
```

Die Geschichte-Liste

Eine polymorphe Datenstruktur:

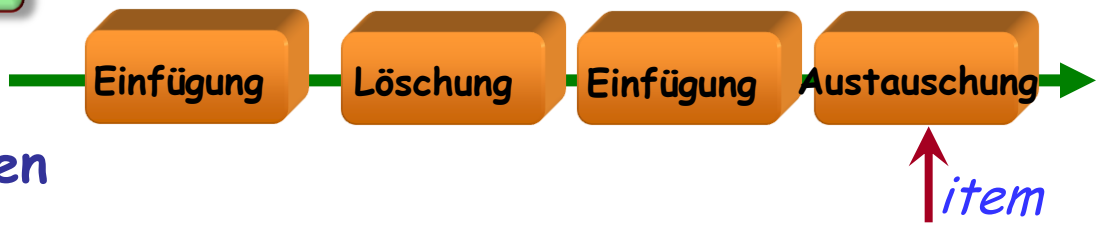


history : *TWO_WAY_LIST* [*COMMAND*]

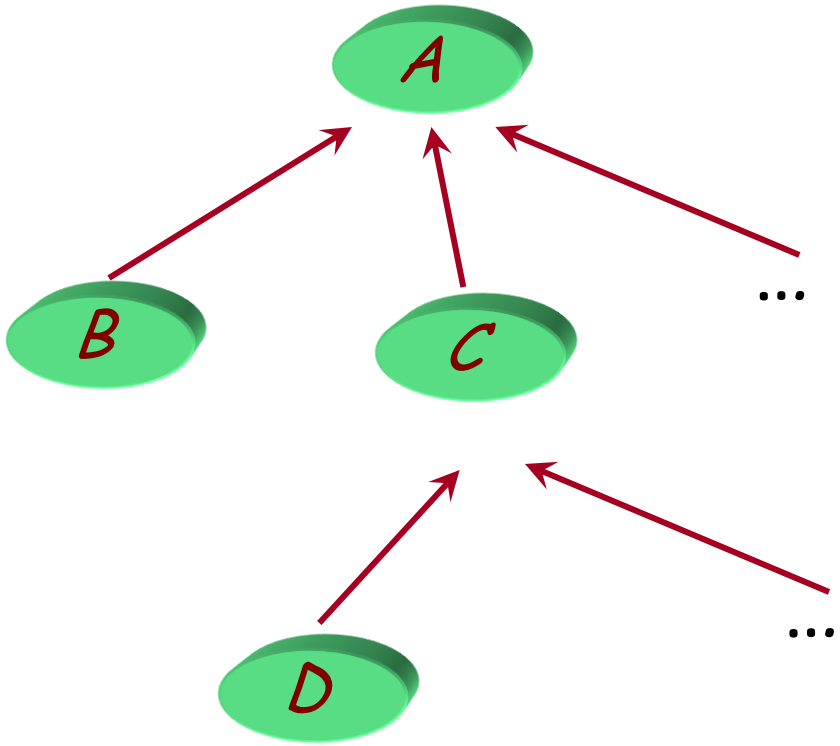
Einen Benutzerbefehl ausführen

```
decode_user_request

if "Anfrage ist normaler Befehl" then
        "Erzeuge ein Befehlsobjekt c , der Anforderung entsprechend"
        history.extend (c)
        c.execute
elseif "Anfrage ist UNDO" then
        if not history.before then -- Ignoriere überschüssige Anfragen
            history.item.undo
            history.back
        end
elseif "Anfrage ist REDO" then
        if not history.is_last then - Ignoriere überschüssige Anfragen
            history.forth
            history. item.execute
        end
end
```

Pseudocode, siehe folgende Implementation



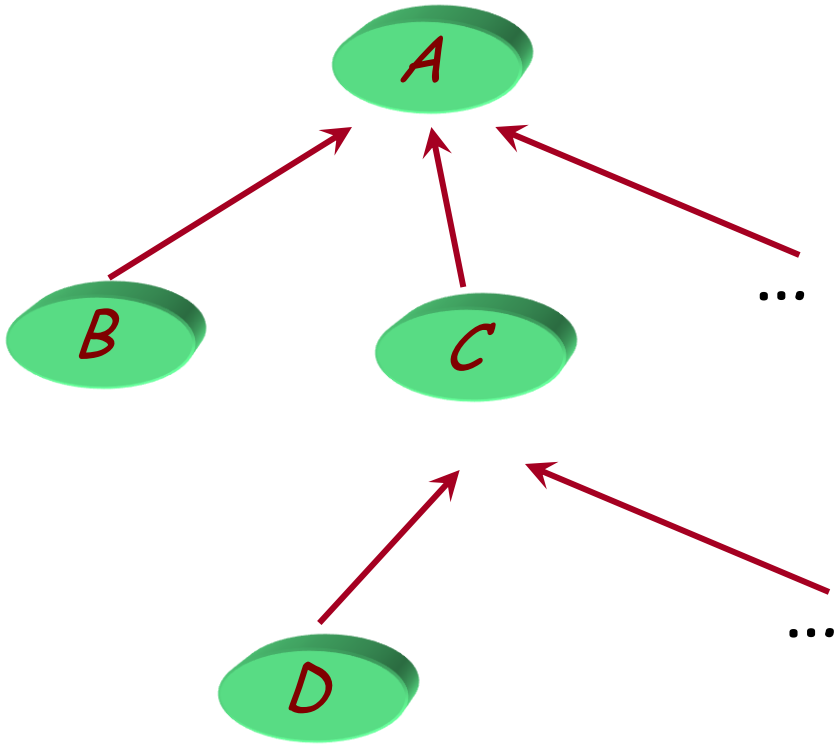
Bedingte Erzeugung (1)




```
a1: A
if condition_1 then
   -- "Erzeuge a1 als Instanz von B"
elseif condition_2 then
   -- "Erzeuge a1 als Instanz von C"
... etc.
```

```
a1: A; b1: B; c1: C; d1: D; ...
if condition_1 then
        create b1.make (...)
        a1 := b1
elseif condition_2 then
        create c1.make (...)
        a1 := c1
... etc.
```

Bedingte Erzeugung (2)

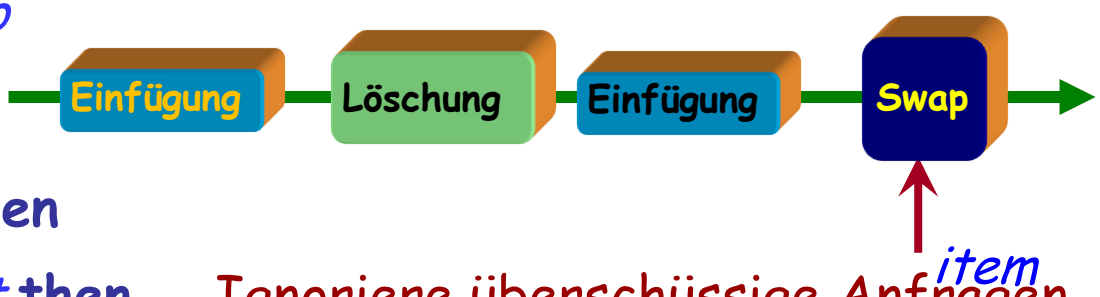



```
a1: A
if condition_1 then
   -- "Erzeuge a1 als Instanz von B"
elseif condition_2 then
   -- "Erzeuge a1 als Instanz von C"
... etc.
```

```
a1: A
if condition_1 then
        create {B} a1.make (...)
elseif condition_2 then
        create {C} a1.make (...)
... etc.
```

Einen Benutzerbefehl ausführen

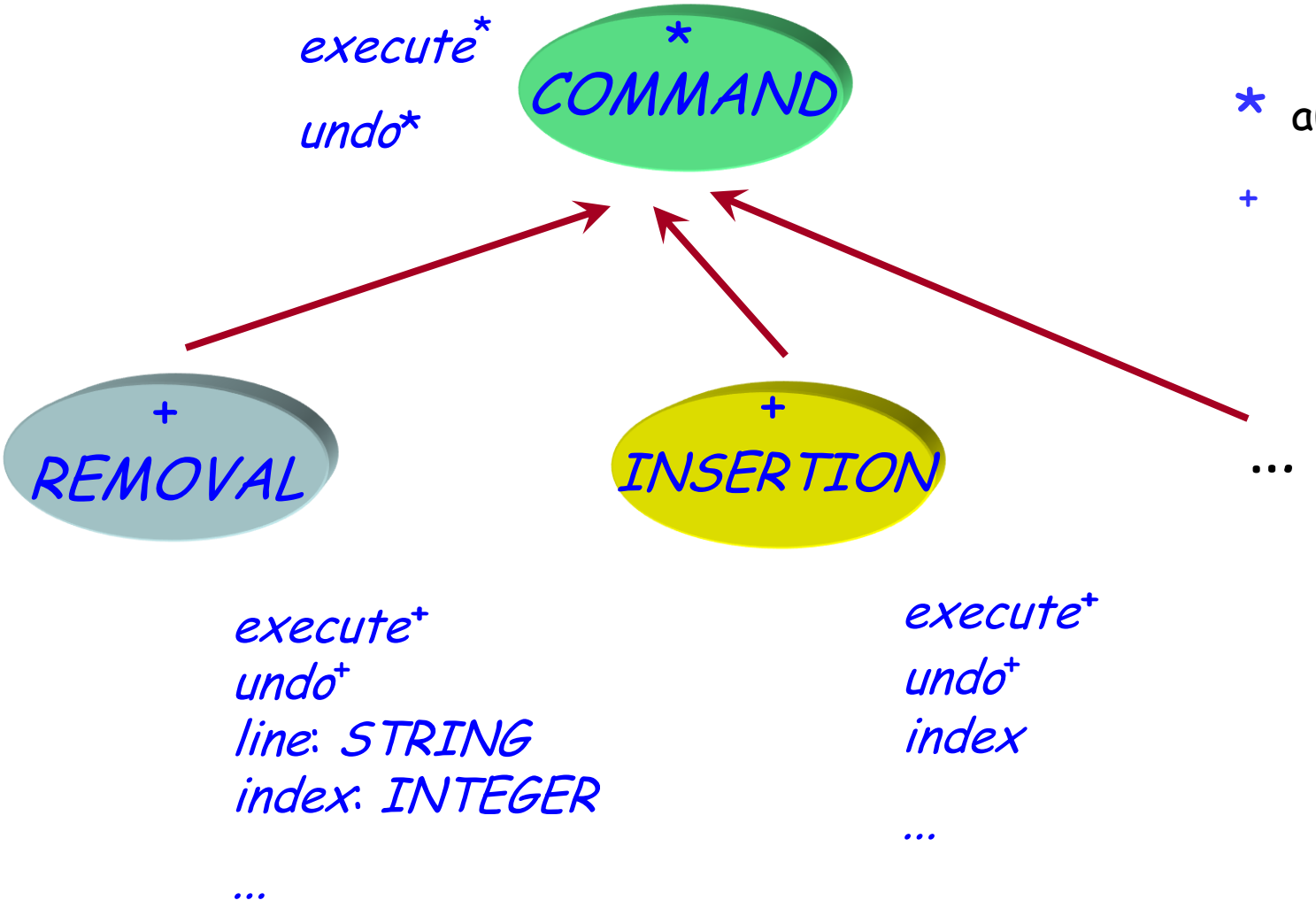
```
decode_user_request

if "Anfrage ist normaler Befehl" then
        "Erzeuge ein Befehlsobjekt c , der Anforderung entsprechend"
        history.extend(c)
        c.execute
elseif "Anfrage ist UNDO" then
        if not history.before then -- Ignoriere überschüssige Anfragen
            history.item.undo
            history.back
        end
elseif "Anfrage ist REDO" then
        if not history.is_last then -- Ignoriere überschüssige Anfragen
            history.forth
            history. item.execute
        end
end
```



Befehlsobjekte erzeugen: Erster Ansatz

```
c: COMMAND

...


decode_user_request
if "Anfrage ist remove" then
        create {REMOVAL} c
elseif "Anfrage ist insert" then
        create {INSERTION} c
else
        etc.
```

Hierarchie der Befehlsklassen



Befehlsobjekte erzeugen: Besserer Ansatz

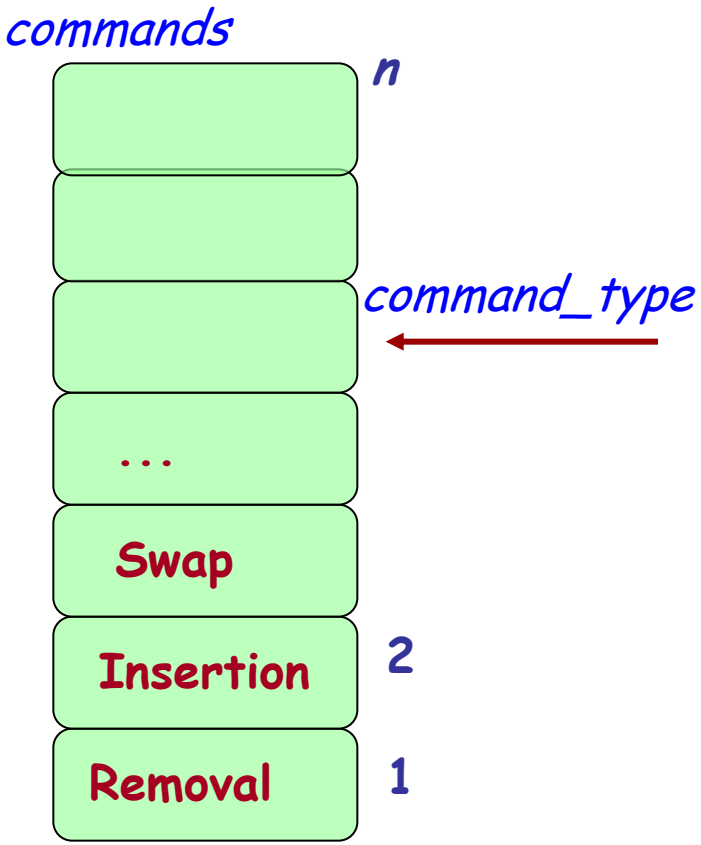
Geben Sie jedem Befehls-Typ eine Nummer

Füllen Sie zu Beginn einen Array *commands* mit je einer Instanz jedes Befehls-Typen.

Um neue Befehlsobjekte zu erhalten:

"Bestimme *command_type*"

c := (*commands* [*command_type*]).*twin*



Einen "Prototypen" duplizieren

Das Undo/Redo- (bzw. Command-) Pattern

Wurde extensiv genutzt (z.B. in EiffelStudio und anderen Eiffel-Tools).

Ziemlich einfach zu implementieren.

Details müssen genau betrachtet werden (z.B. lassen sich manche Befehle nicht rückgängig machen).

Eleganter Gebrauch von O-O-Techniken

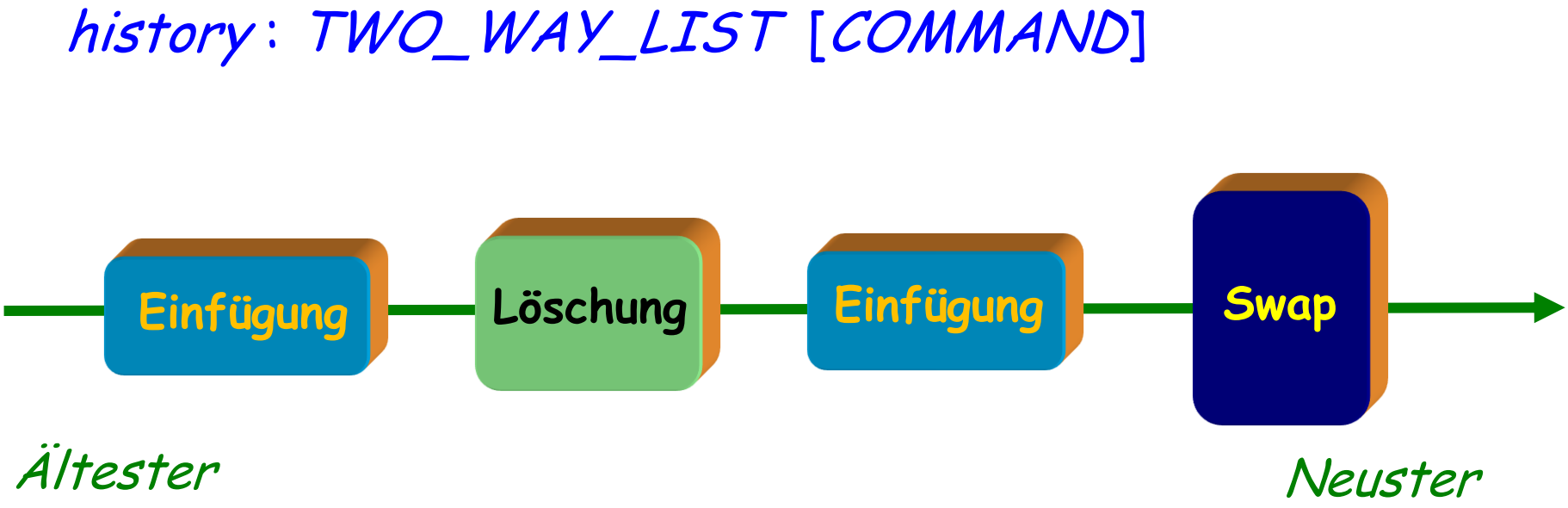
Nachteil: Explosion kleiner Klassen

Der Gebrauch von Agenten

Für jeden Benutzerbefehl haben wir zwei Routinen:

- Die Routine, um ihn auszuführen
- Die Routine, um ihn rückgängig zu machen

Die Geschichte-Liste im Undo/Redo-Pattern



Die Geschichte-Liste mit Agenten

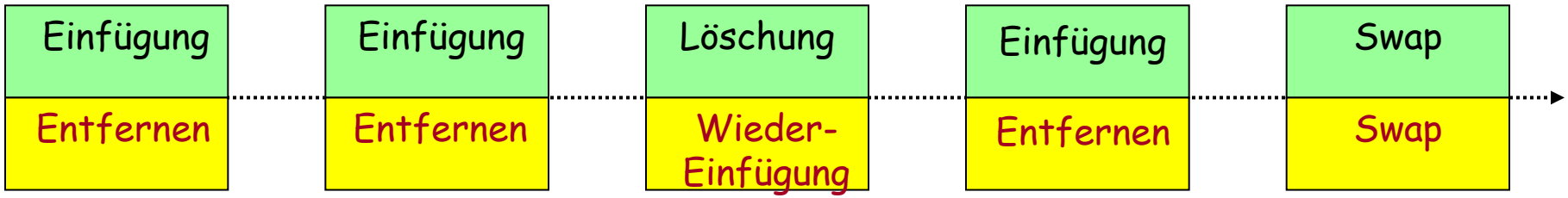
Die Geschichte-Liste wird einfach zu einer Liste von Agentenpaaren:

history : *TWO_WAY_LIST* [*TUPLE*

[*doer* : *PROCEDURE* [*ANY*, *TUPLE*],

undoer : *PROCEDURE* [*ANY*, *TUPLE*]]

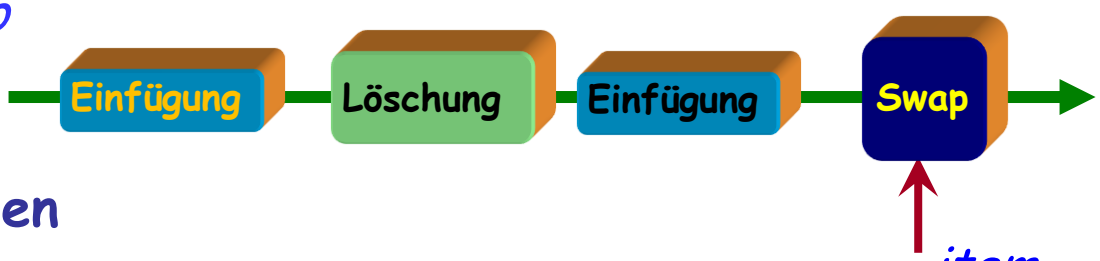
Benanntes Tupel



Das Grundschema bleibt dasselbe, aber man braucht nun keine Befehlsobjekte mehr; die Geschichte-Liste ist einfach eine Liste, die Agenten enthält.

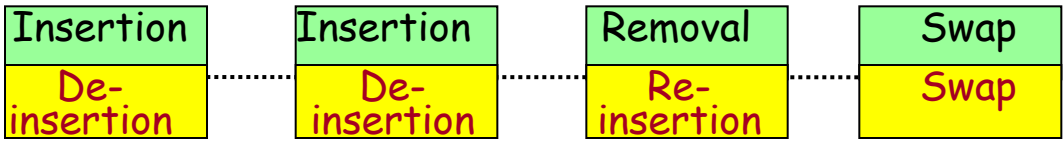
Einen Benutzerbefehl ausführen (vorher)

```
decode_user_request

if "Anfrage ist normaler Befehl" then
        "Erzeuge ein Befehlsobjekt c , der Anforderung entsprechend"
        history.extend(c)
        c.execute
elseif "Anfrage ist UNDO" then
        if not history.before then -- Ignoriere überschüssige Anfragen
            history.item.undo
            history.back
        end
elseif "Anfrage ist REDO" then
        if not history.is_last then - Ignoriere überschüssige Anfragen
            history.forth
            history. item.execute
        end
end
```



Einen Benutzerbefehl ausführen (jetzt)

```
"Dekodiere Benutzeranfrage mit zwei Agenten do_it and undo_it"
if "Anfrage ist normaler Befehl" then
        history.extend([do_it, undo_it])
        do_it.call([])
elseif "Anfrage ist UNDO" then
        if not history.before then
                history.item.undoer .call([])
                history.back
        end
elseif "Anfrage ist REDO" then
        if not history.is_last then
                history.forth
                history.item.doer .call([])
        end
end
```



Undo/Redo

Menschen machen Fehler!

Auch wenn sie keine Fehler machen: sie wollen experimentieren. Undo/Redo unterstützt einen „trial and error"-Stil.

Undo/Redo-Pattern:

- Sehr nützlich in der Praxis
- Weit verbreitet
- Ziemlich einfach zu implementieren
- Exzellentes Beispiel von eleganten O-O-Techniken
- Noch besser mit Agenten!

Was wir gesehen haben

1. Ein mächtiges Entwurfsmuster: Beobachter
   Besonders in Situationen, bei denen Änderungen eines Wertes viele Kundeen betreffen.
2. Generelles Schema für interaktive Applikationen
3. Operationen als Objekte behandeln
   Agenten, delegates, closures…
   Operationen weiterreichen
   Speichern der Operationen in Tabellen
4. Anwendungen von Agenten, z.B. numerische Analysis
5. Hilfskonzepte: Tupel, einmalige (once) Routinen
6. Rückgängig machen
   A) Mit dem Command-Pattern
   B) Mit Agenten