



Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 17: Ereignisorientierte Programmierung und Agenten



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)

Die Diskussion erlaubt uns auch, zwei wichtige **Entwurfsmuster** (*design patterns*) zu studieren: Beobachter (*observer*) und MVC



Das Programm führt
den Benutzer:

from

$i := 0$

read_line

until *end_of_file* loop

$i := i + 1$

Result [*i*] := *last_line*

read_line

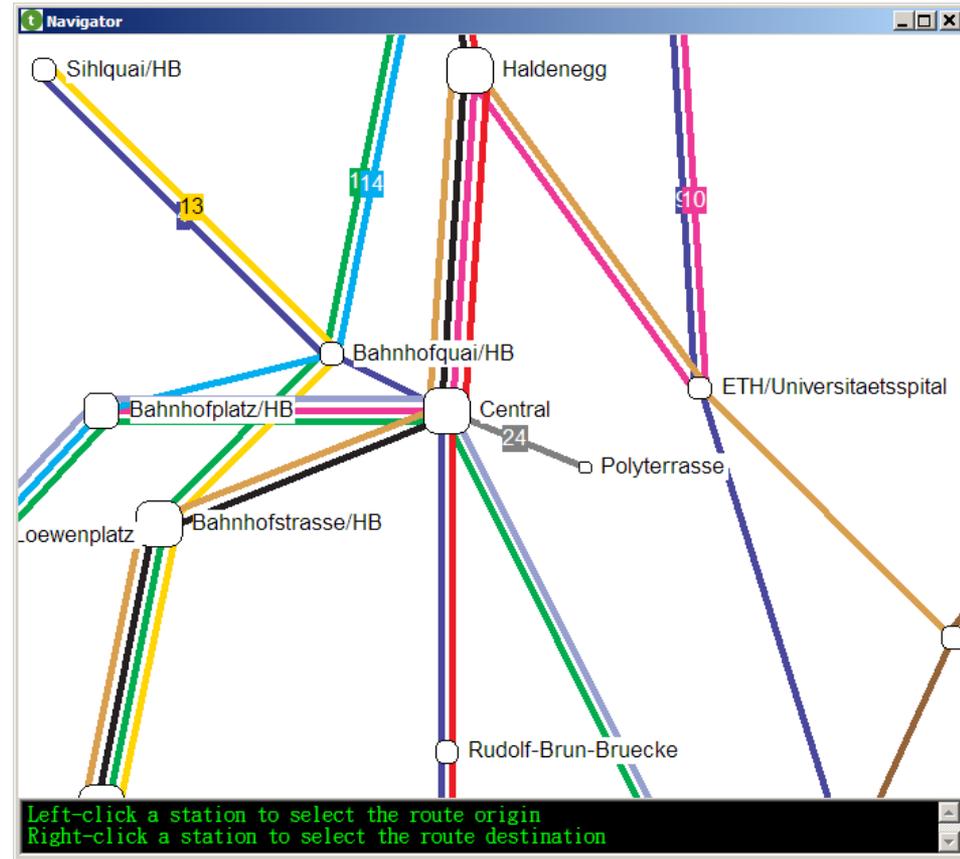
end





Der Benutzer führt das Programm:

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



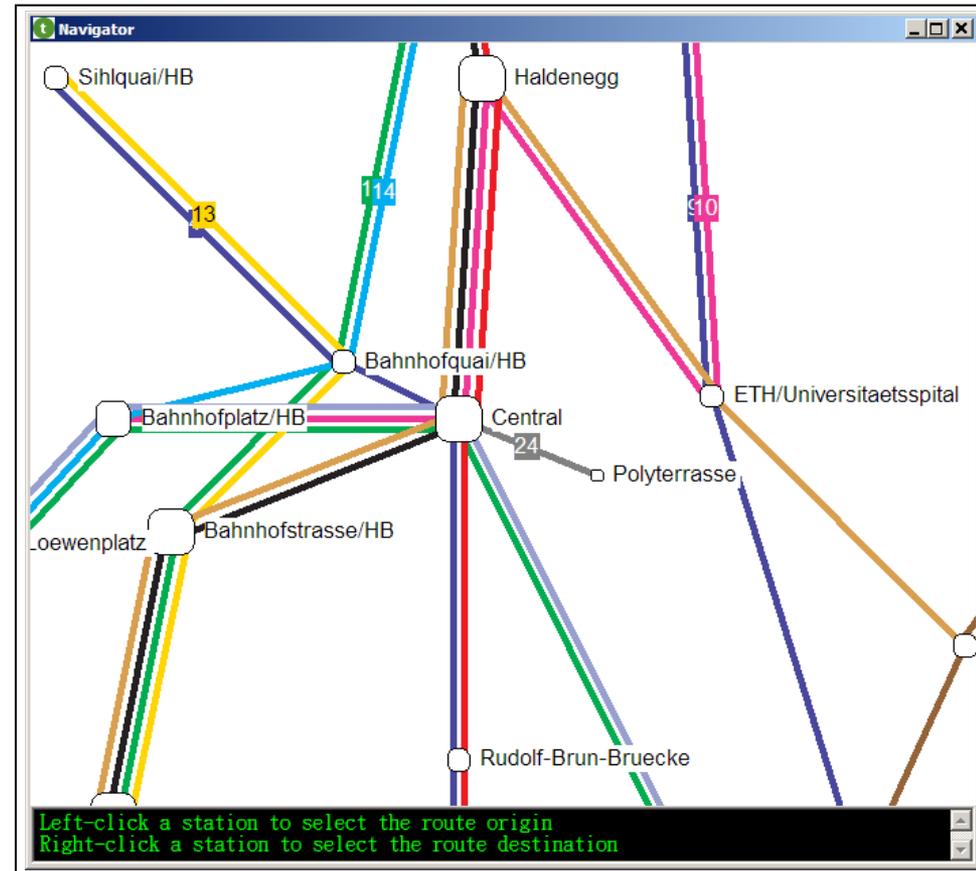
Ereignisorientierte Programmierung: Beispiel



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





1. Das „Geschäftsmodell“ und das GUI getrennt halten.
 - Geschäftsmodell (oder einfach nur *Modell*): Kernfunktionalitäten der Applikation
 - GUI: Interaktion mit Benutzern
2. Den Verbindungscode (*glue code*) zwischen den beiden minimieren
3. Sicherstellen, dass wir mitbekommen, was passiert

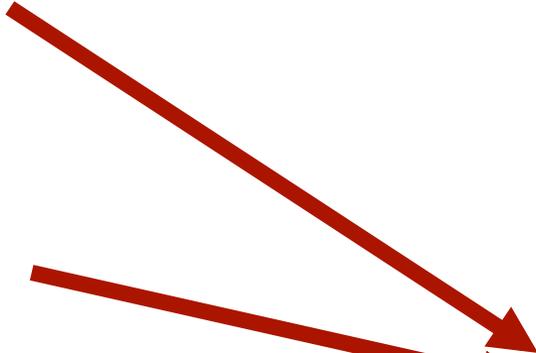
Ereignisorientierte Programmierung: Metapher



Herausgeber

Subskribent

1



2



3



4



Routine

A



Routine

B



C



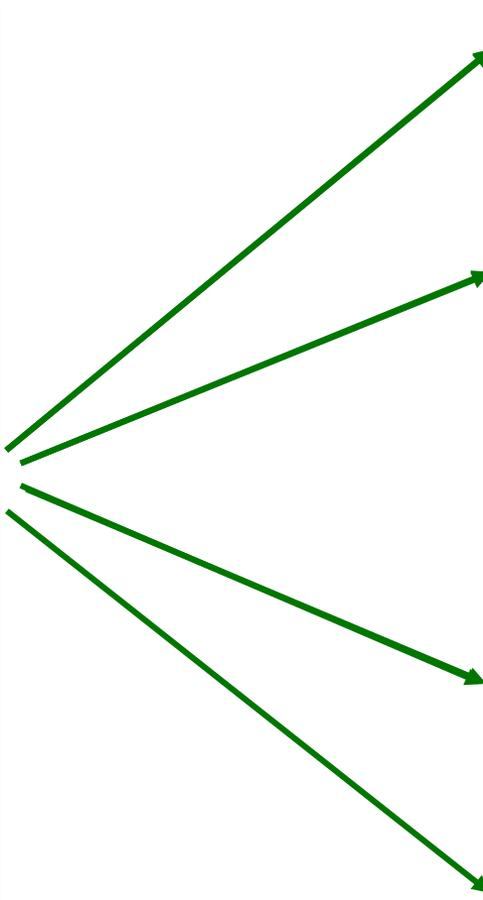
Routine

D



Routine

E

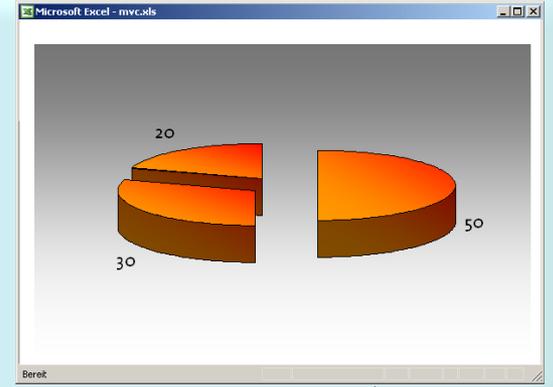
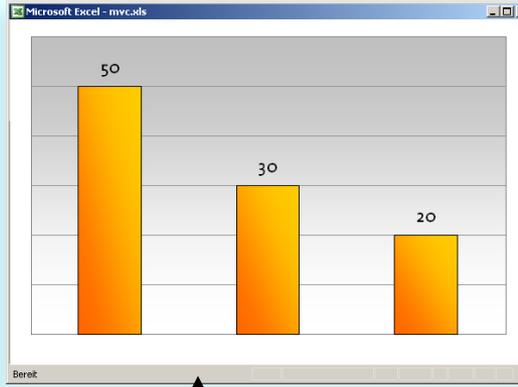


Einen Wert beobachten



Beobachter

	A	B	C	D	E	F	G
1	50	30	20				
2	10	20	70				
3	30	60	10				
4							
5							
6							
7							



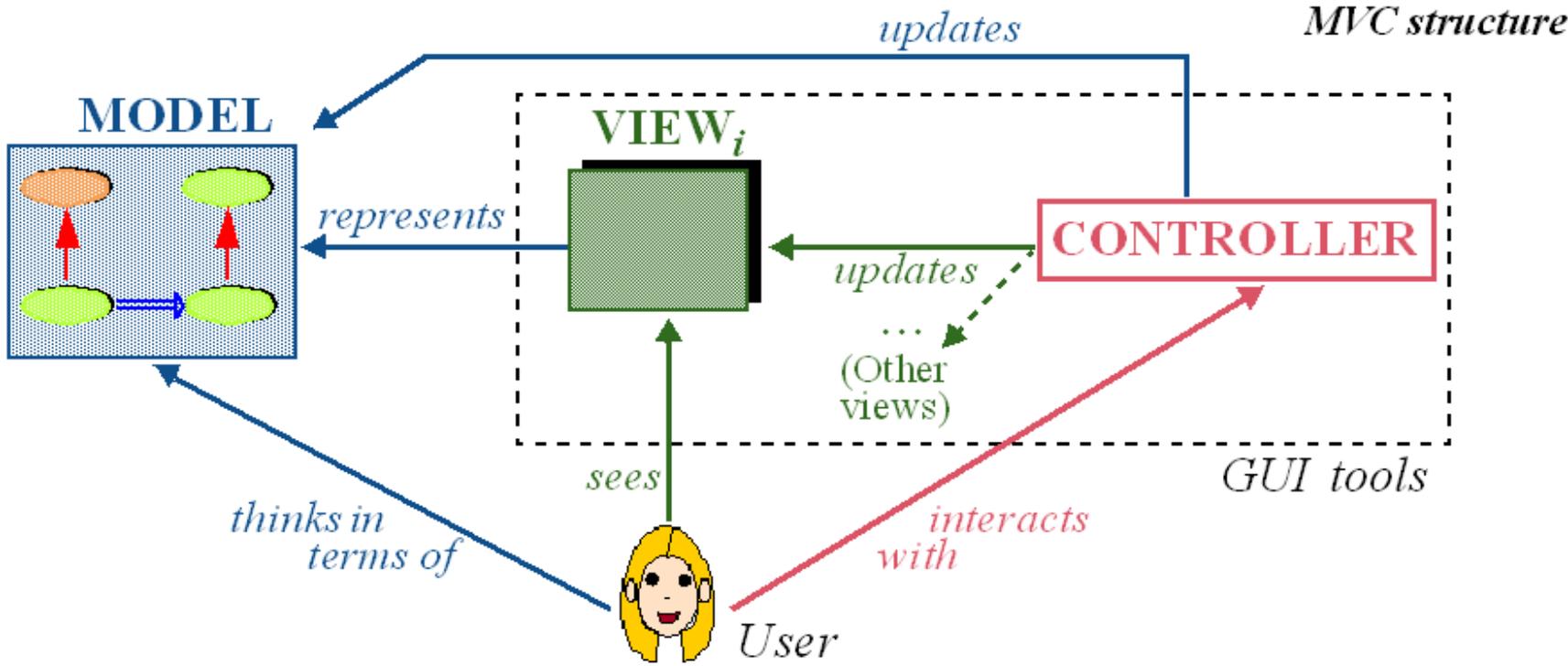
Subjekt

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

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



(Trygve Reenskaug, 1979)



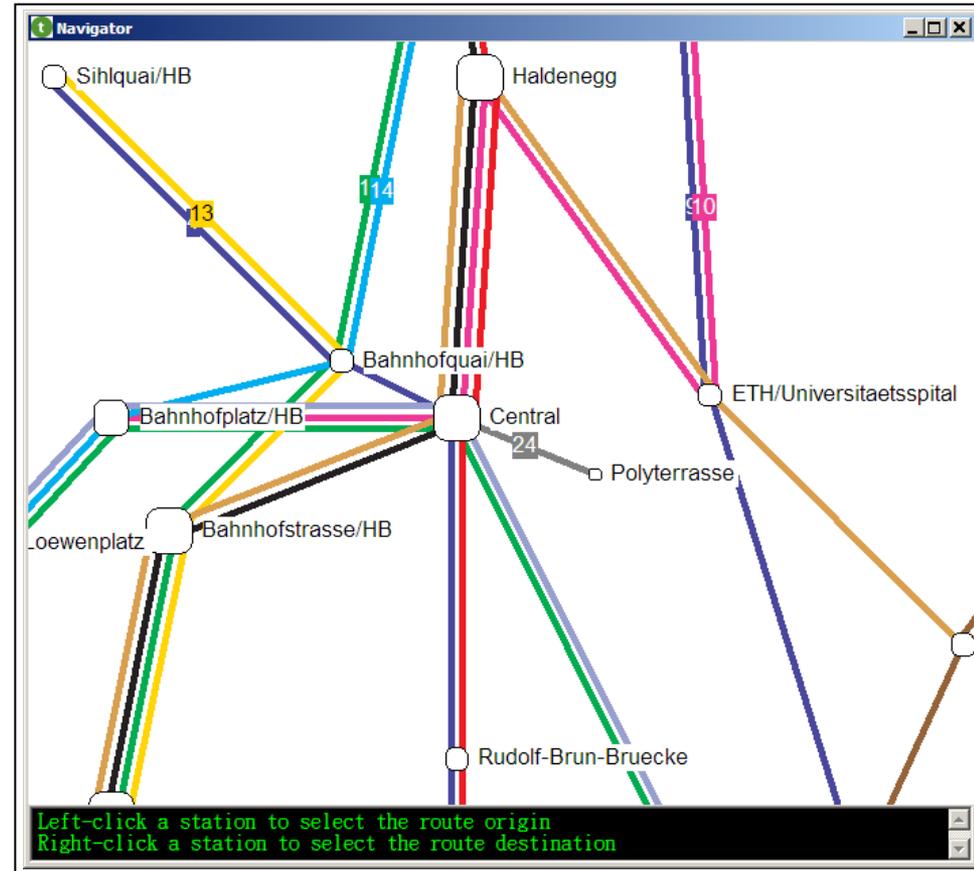
Unser Beispiel



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 für dieses **Ereignis** 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.



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

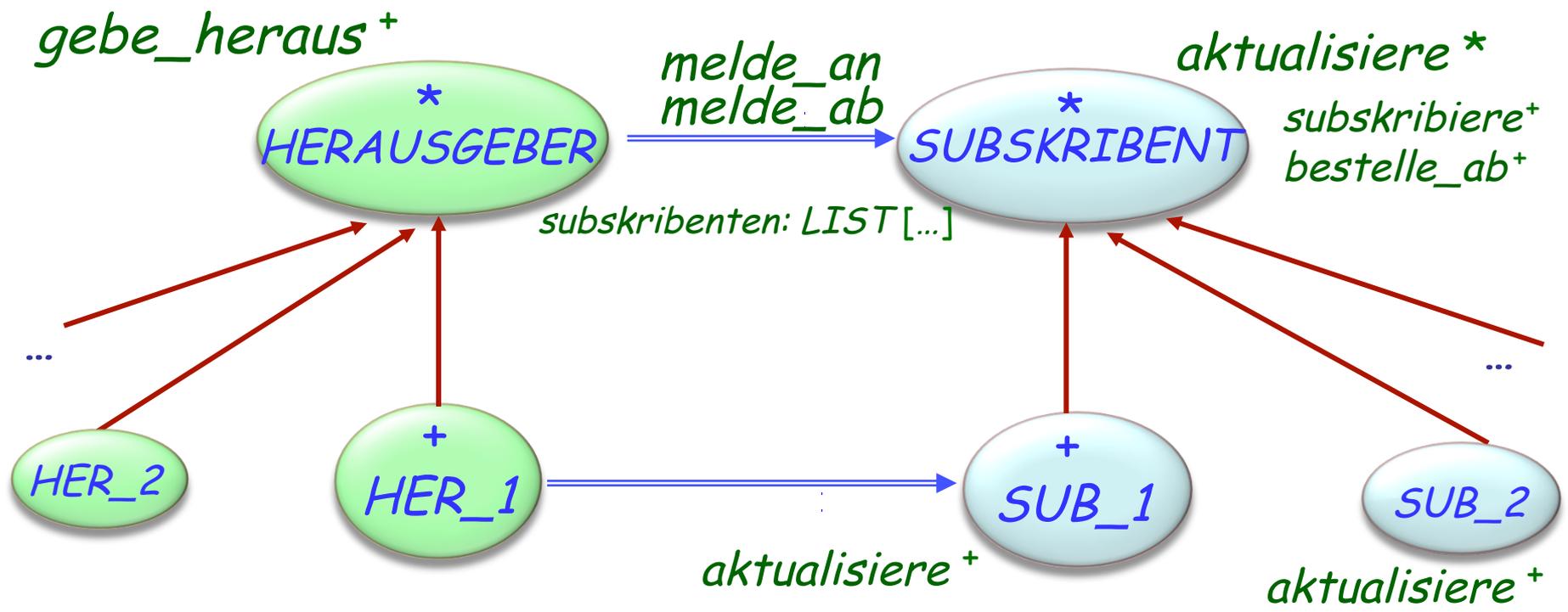
Subskribent = Beobachter = Hörer
(*Subscriber, Observer, Listener*)

Herausgeber = Subjekt = Beobachtete
(*Publisher, Subject, Observed*)

Herausgeben / Subskribieren (*Publish / Subscribe*)

Ereignisorientiertes (*Event-Oriented*) Design &
Programmieren

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



- * aufgeschoben (*deferred*)
- + wirksam (*effective*)

- ↑ Erbt von
- ==> Kunde (benutzt)

Entwurfsmuster (Design Pattern)



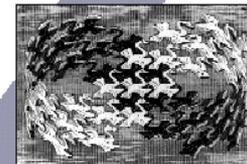
Ein Entwurfsmuster ist ein architektonisches Schema — eine Organisation von Klassen und Features — das Anwendungen standardisierte Lösungen für häufige Probleme bietet

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

Design Patterns

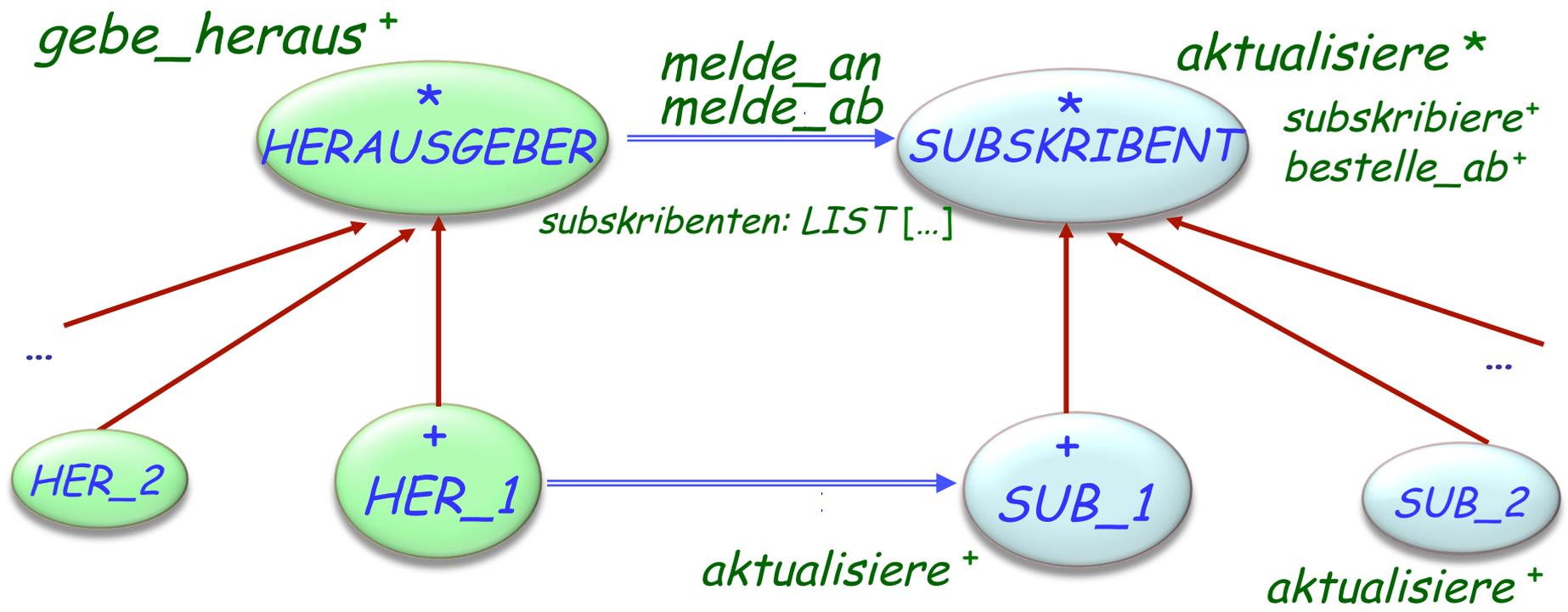
Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

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

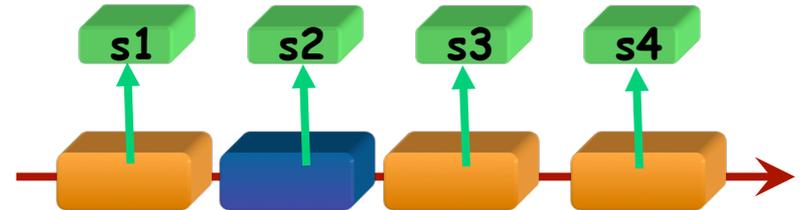


Das Beobachter-Muster



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*:

```
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)



Wieso?

Einen Ereignis auslösen



gebe_heraus

-- Lade alle Beobachter ein,
-- auf diesem Ereignis zu reagieren.

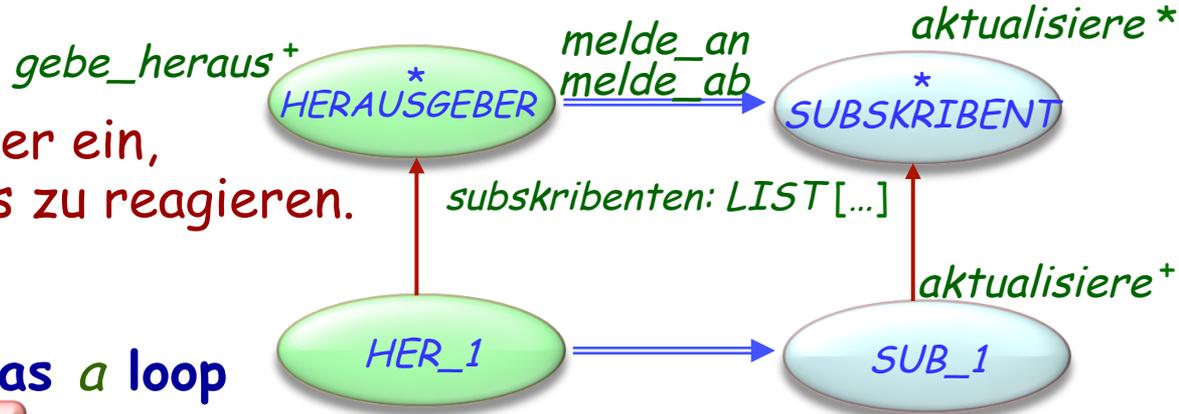
do

across *subskribenten* as a loop

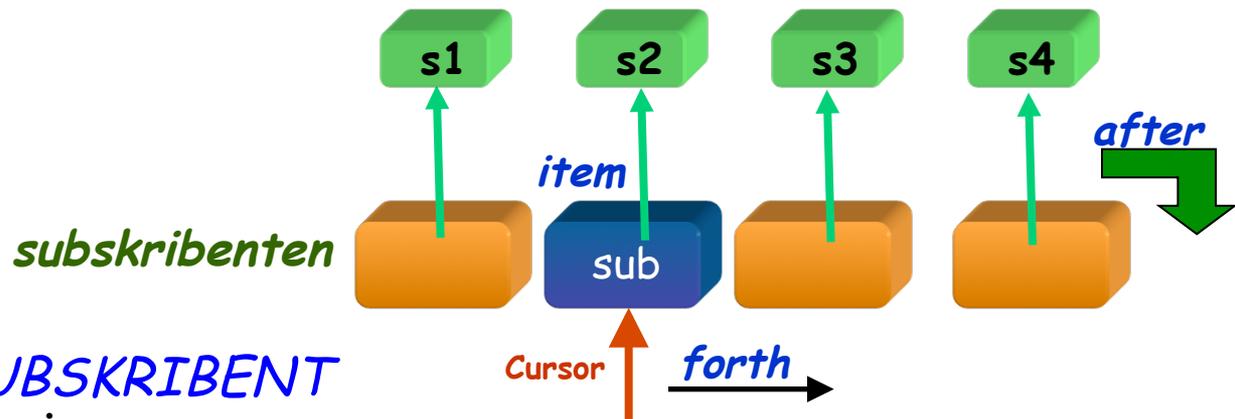
a.item. **aktualisiere**

end

end

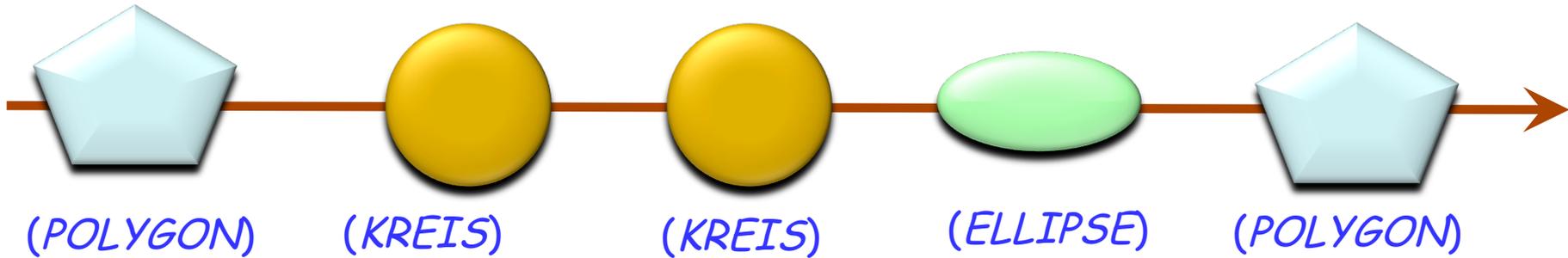


Dynamisches Binden



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

Erinnerung: Liste von Figuren



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

```
bilder: LIST [FIGUR]  
p1, p2: POLYGON  
c1, c2: KREIS  
e: ELLIPSE
```

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

Das Beobachter-Muster (in der Grundform)



- Die Herausgeber kennen (intern) die Subskribenten
- Jeder Subskribent kann sich nur bei maximal einem Herausgeber einschreiben
- Er kann maximal eine Operation registrieren
- Die Lösung is nicht wiederverwendbar — muss für jede Applikation neu programmiert werden
- Argumente zu behandeln ist schwierig

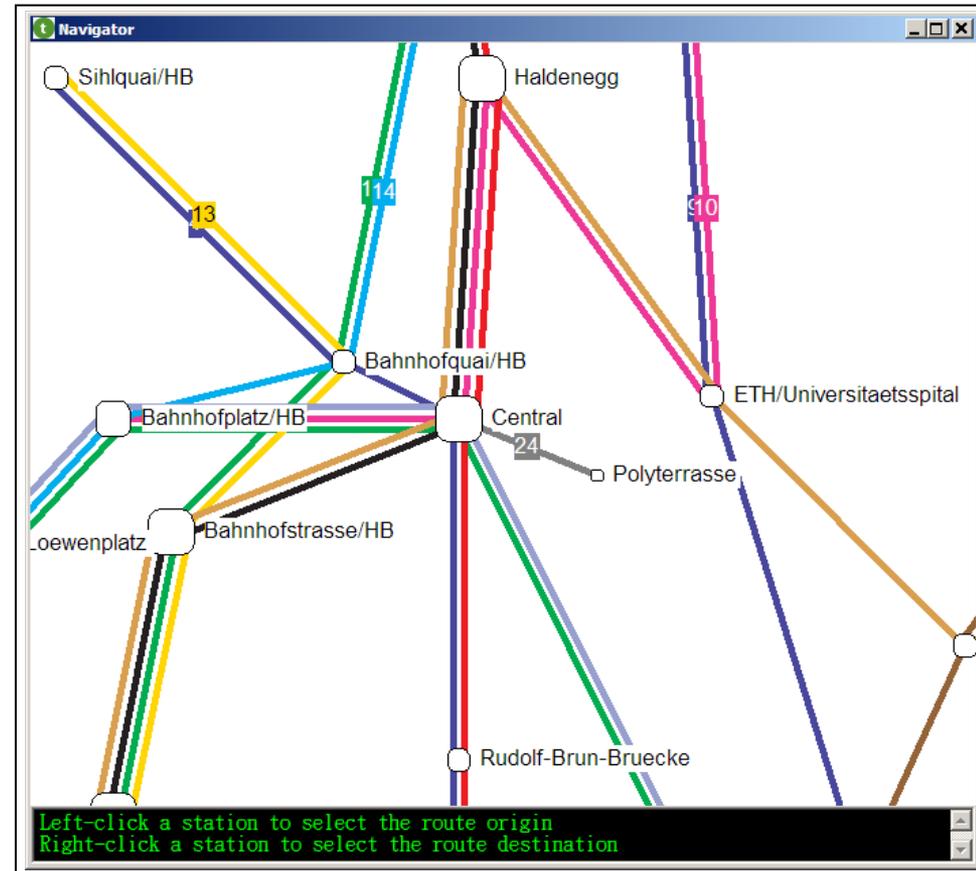
Ereignisorientierte Programmierung: Beispiel



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



Anderer Ansatz: Ereignis-Kontext-Aktion-Tabelle

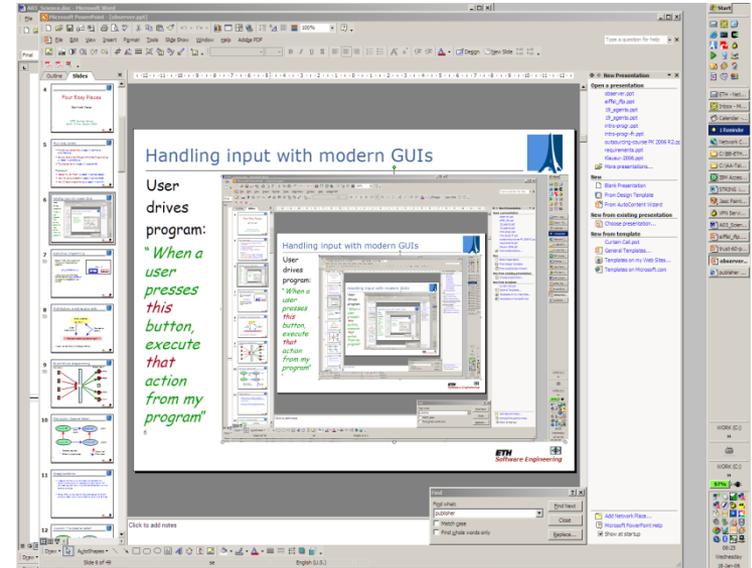
Eine Menge von „Tripeln“
[Ereignis-Typ, Kontext, Aktion]

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 das 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
Links_klick	Speichern_knopf	<i>Speichere_datei</i>
Links_klick	Abbrechen_knopf	<i>Reset</i>
Links_klick	Karte	<i>Finde_station</i>
Links_klick
Recht_klick	...	<i>Anzeige_menu</i>
...		...

Ereignis-Kontext-Aktion-Tabelle



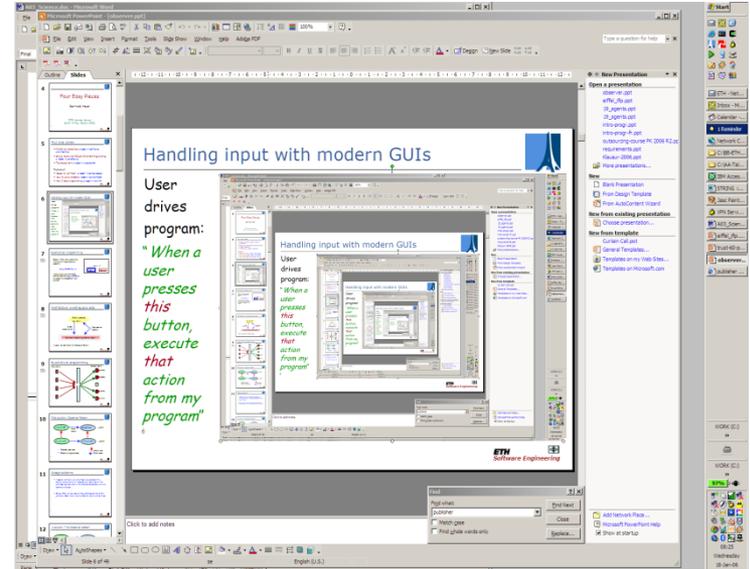
Eine Menge von „Tripeln“
[Ereignis-Typ, Kontext, Aktion]

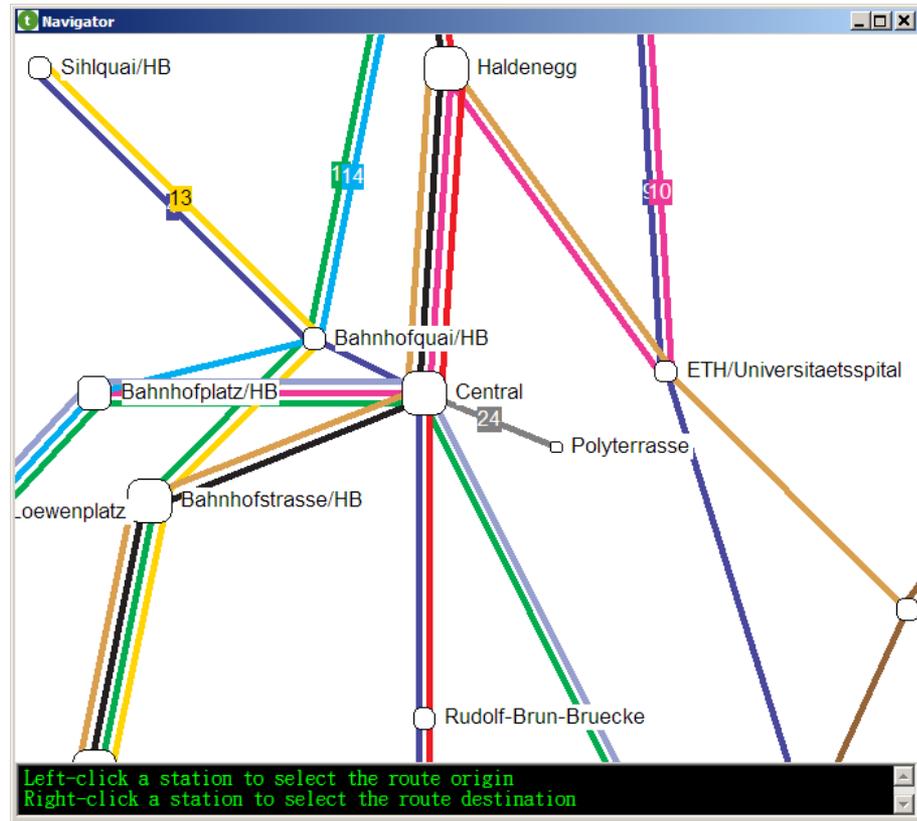
Ereignis-Typ: irgendeine Art von Ereignis, an dem wir interessiert sind.
Beispiel: Linksklick

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

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

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





Zurich_map.on_left_click.extend_back(agent find_station)



C und *C++*: "Funktionszeiger"

C#: Delegaten (eine Form von Agenten)



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.



- 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;
```



- 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 das Ereignis aus:
OnClick (*yourClickargs*);



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

D2. Instantiieren mit *yourProcedure* als Argument:

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

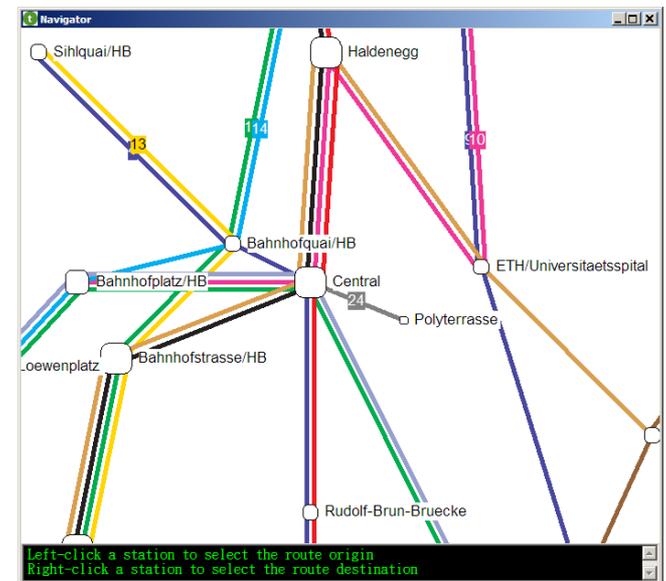
D3. Hinzufügen des Delegates zur Liste für das Ereignis:

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

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*





Grundsätzlich:

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

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



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:

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

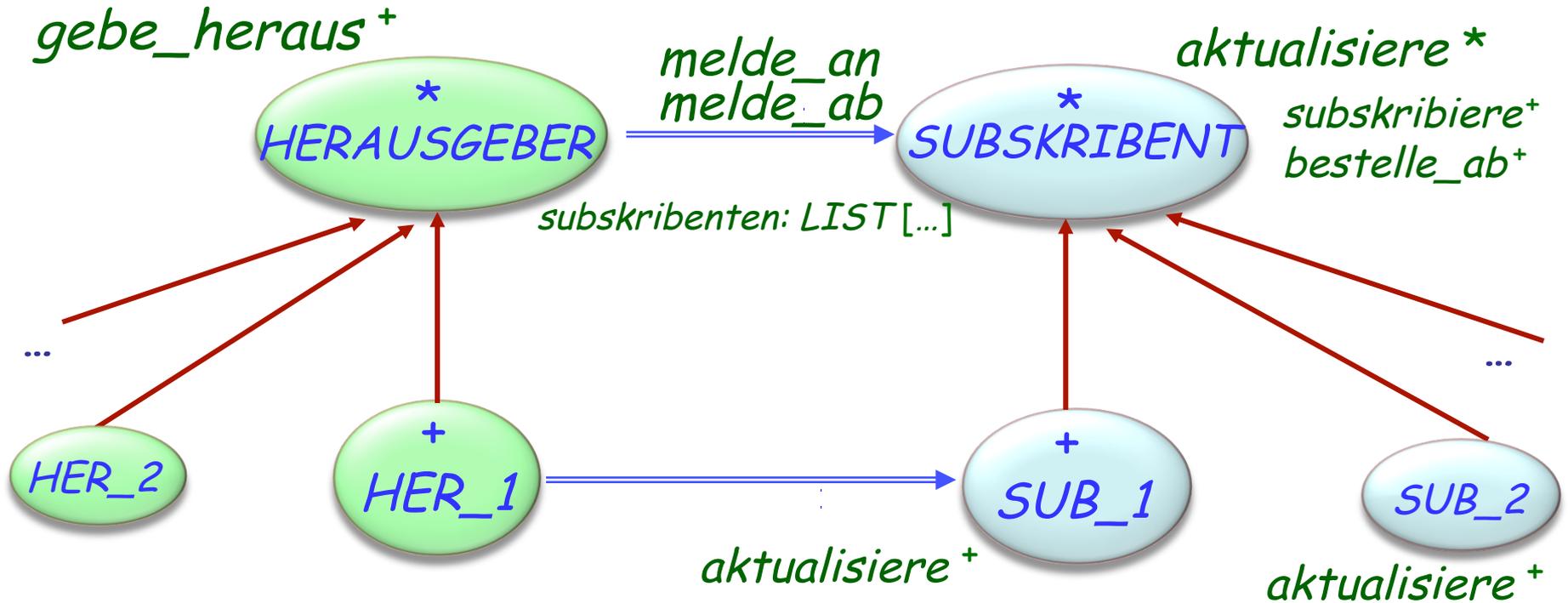
Der Herausgeber (Subjekt) löst ein Ereignis aus:

```
left_click.publish ([x_position, 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: das Beobachter-Muster



- * aufgeschoben (*deferred*)
- + wirksam (*effective*)

- ↑ Erbt von
- ==> Kunde (benutzt)

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



click.subscribe (agent find_station)

Zurich_map.click.subscribe (agent find_station)

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

click.subscribe (agent other_object.other_procedure)

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.

$[a1, b1, c1, d1]$

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]

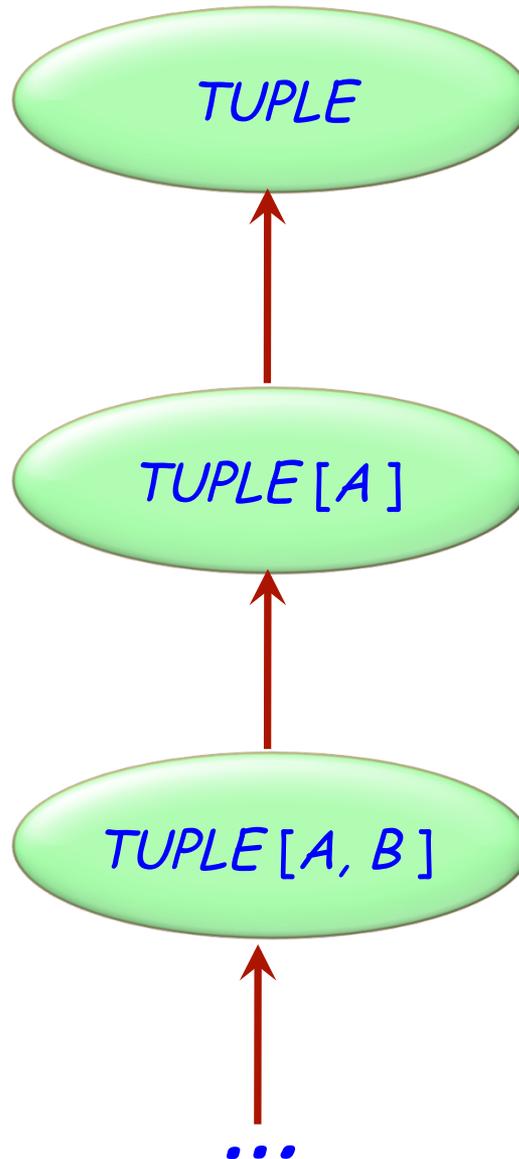
aber er vereinfacht den Zugriff auf einzelne Elemente

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

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

Um ein auf ein Tupel-element zuzugreifen: z.B. *t.jahr*

Die Vererbungshierarchie von Tupeln





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( [horizontale_position, vertikale_position] )
```

Falls *a* mit einer Funktion assoziiert ist, gibt

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

das Resultat der Anwendung der Funktion zurück

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:

$u := \text{agent } a0.f(a1, a2, a3)$ -- Alle geschlossen (bereits gesehen)

$w := \text{agent } a0.f(a1, a2, ?)$

$x := \text{agent } a0.f(a1, ?, a3)$

$y := \text{agent } a0.f(a1, ?, ?)$

$z := \text{agent } a0.f(?, ?, ?)$

Den Agenten aufrufen



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

$u := \text{agent } a0.f(a1, a2, a3)$

$u.call([])$

$v := \text{agent } a0.f(a1, a2, ?)$

$v.call([a3])$

$w := \text{agent } a0.f(a1, ?, a3)$

$w.call([a2])$

$x := \text{agent } a0.f(a1, ?, ?)$

$x.call([a2, a3])$

$y := \text{agent } a0.f(?, ?, ?)$

$y.call([a1, a2, a3])$

Ein weiteres Beispiel zum Aufruf von Agenten



$$\int_a^b \text{meine_funktion}(x) dx$$

$$\int_a^b \text{ihre_funktion}(x, u, v) dx$$

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

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

Die Integralfunktion



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

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

```
local  
  x: REAL; i: INTEGER
```

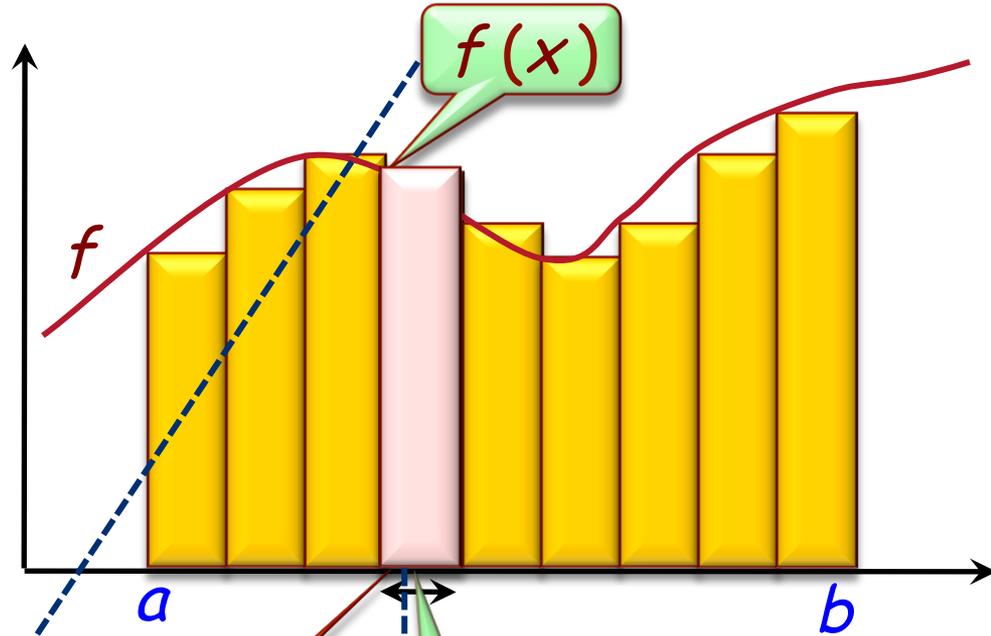
```
do
```

```
  from x := a until x > b loop
```

```
    Result := Result +  
    i := i + 1
```

```
    x := a + i * schritt
```

```
  end  
end
```



```
f.item([x]) * schritt
```

Numerische Frage: warum nicht $x := x + schritt$?

Weitere Anwendung: Benutzen eines Iterators



```
class C feature
```

```
  alle_positiv, alle_verheiratet: BOOLEAN
```

```
  ist_positiv (n: INTEGER): BOOLEAN
```

```
    -- Ist n grösser als null?
```

```
  do Result := (n > 0) end
```

```
  intlist: LIST[INTEGER]
```

```
  ang_list: LIST[ANGESTELLTER]
```

```
  r
```

```
  do
```

```
    alle_positiv := intlist.for_all (agent ist_positiv (?))
```

```
    alle_verheiratet := ang_list.for_all
```

```
      (agent {ANGESTELLTER} ist_verheiratet)
```

```
  end
```

```
end
```

```
class ANGESTELLTER feature
  ist_verheiratet: BOOLEAN
  ...
end
```



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

for_all

there_exists

do_all

do_if



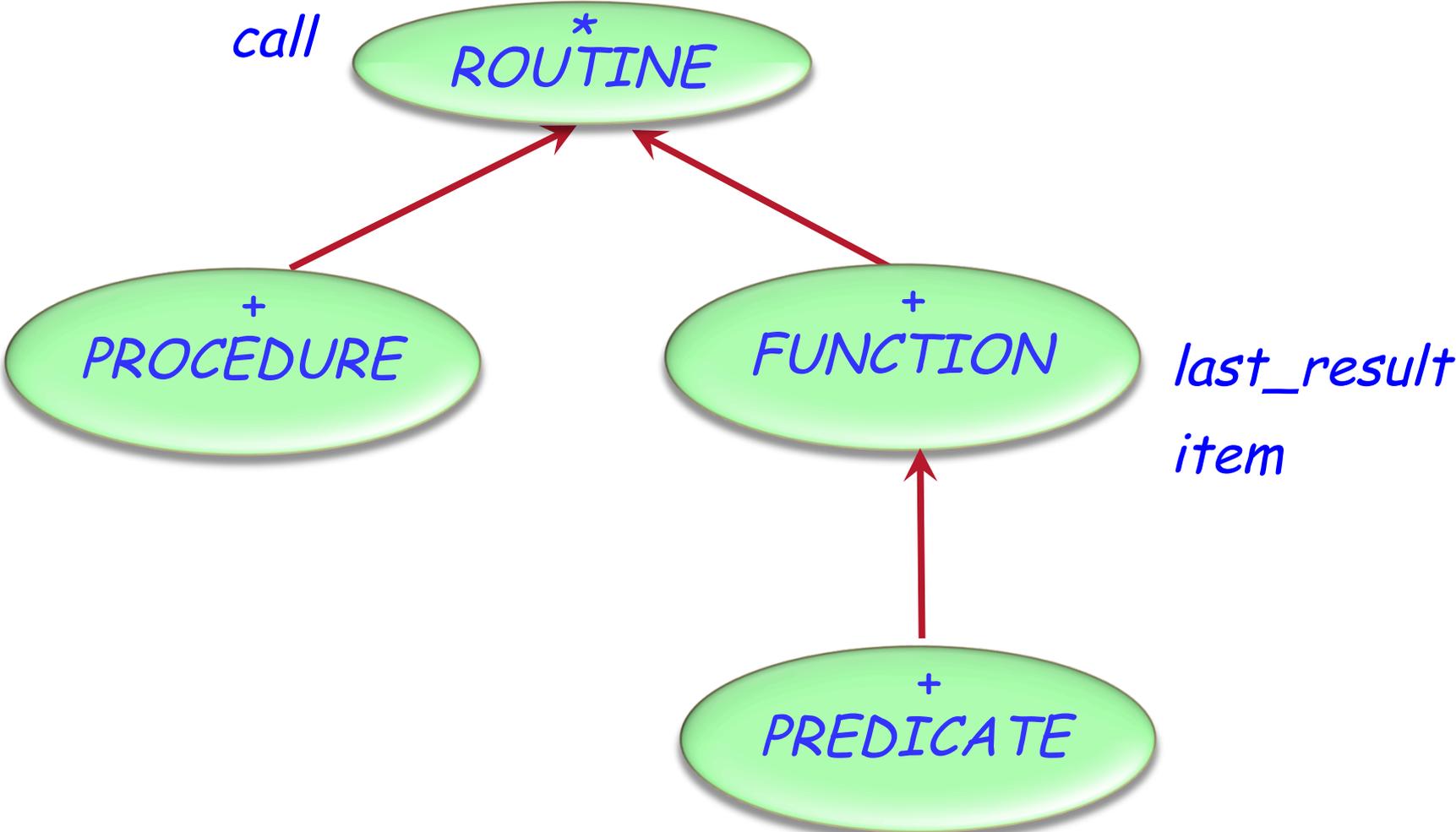
Entwurfsmuster: Beobachter (Beobachter), Visitor, Undo-redo (Command-Pattern)

Iteration

Verträge auf einer hohen Ebene

Numerische Programmierung

Introspektion (die Eigenschaften eines Programmes selbst herausfinden)





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



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

$u := \text{agent } a0.f(a1, a2, a3)$

$u.call([])$

$v := \text{agent } a0.f(a1, a2, ?)$

$v.call([a3])$

$w := \text{agent } a0.f(a1, ?, a3)$

$w.call([a2])$

$x := \text{agent } a0.f(a1, ?, ?)$

$x.call([a2, a3])$

$y := \text{agent } a0.f(?, ?, ?)$

$y.call([a1, a2, a3])$



1. **Ein mächtiges Entwurfsmuster:** Beobachter
Besonders in Situationen, bei denen
Änderungen eines Wertes viele Kunden
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