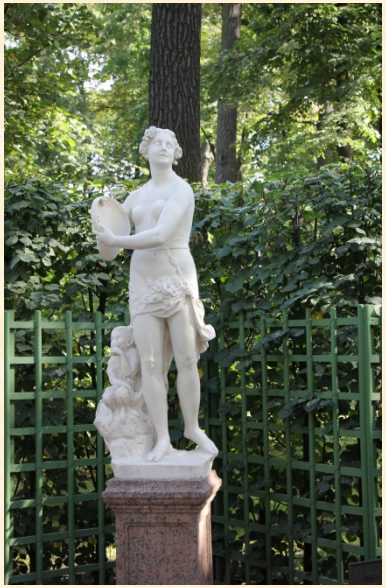
ETH zürich
Chair of Software Engineering

Einführung in die Programmierung

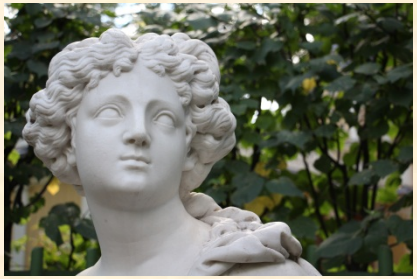
Prof. Dr. Bertrand Meyer

Lecture 18: Undo/Redo







Weiterführende Referenzen

Kapitel 21 von 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 I(“**Control-Y**”) ohne Limitierung unterstützen, ausser der Benutzer gibt eine maximale Tiefe an.

Ein Beispiel: Undo-Redo

Undo/Redo sinnvoll in jeder interaktiven Anwendung

- Entwickeln Sie keine interaktiven Anwendungen, ohne Undo/Redo zu implementieren

Nützliches Design-Pattern ("**Command**" Pattern)

Veranschaulicht die Verwendung von Algorithmen und Datastrukturen

Beispiel für O-O Techniken: Vererbung, Aufgeschobene Klassen, Polymorphe Datenstrukturen, Dynamisches Binden,…

Beispiel einer schönen und eleganten Lösung

Referenzen:

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

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.
- …

Der Einfachheit halber nutzen wir 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 Ressourcen, insbesondere Speicherplatz.

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

Schlüsselschritt im Entwerfen einer Software-Architektur

Die richtigen Abstraktionen finden

(die interessanten Objekt-Typen)

Hier:

Der Begriff eines “Befehls”

Den „Verlauf“ einer Sitzung speichern

Die Verlauf-Liste:

verlauf : *LIST* [*BEFEHL*]

Was ist ein “Befehl” (*Command*) -Objekt?

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

- den Befehl auszuführen
- den Befehl rückgängig zu machen

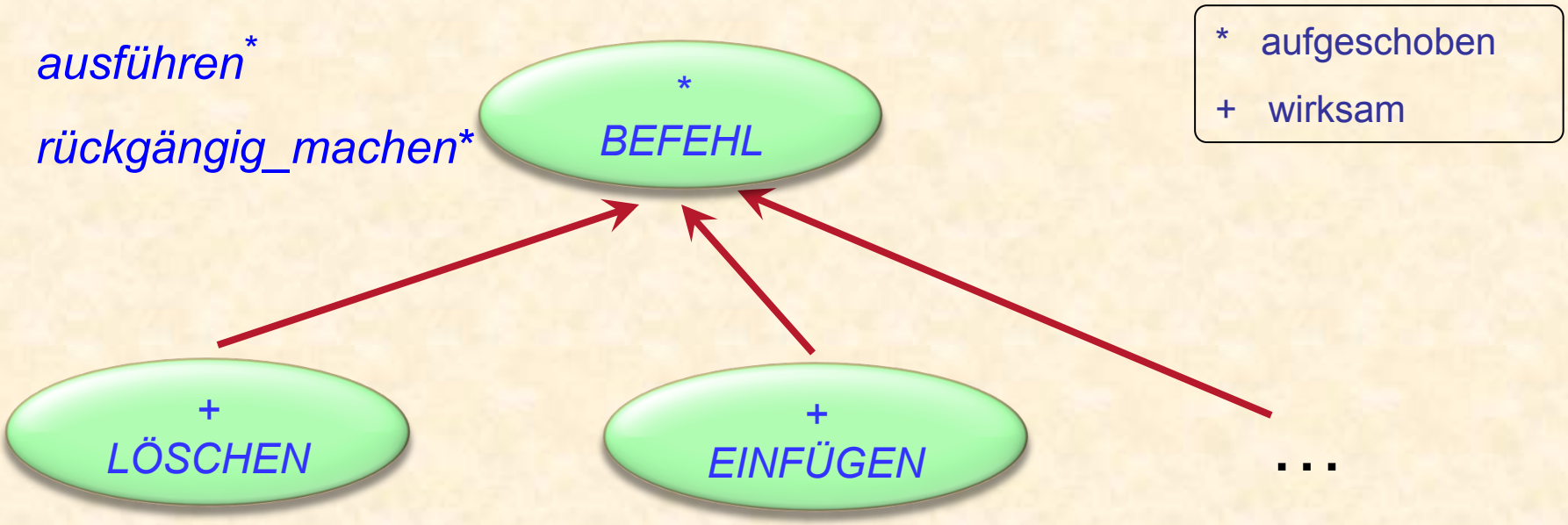
Beispiel: In einem “**Löschen**”-Objekt brauchen wir:

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

Allgemeiner Begriff eines Befehls

```
deferred class BEFEHL feature

    done: BOOLEAN
            -- Wurde dieser Befehl ausgeführt?

    ausführen
            -- Eine Ausführung des Befehls ausführen.
        deferred
        ensure
            already: done
        end

    rückgängig_machen
            -- Eine frühere Ausführung des Befehls
            -- rückgängig machen
        require
            already: done
        deferred
        end

end
```

Die Befehl-Klassenhierarchie



ausführen+
rückgängig_machen+
linie: STRING
index: INTEGER

...

ausführen+
rückgängig_machen+
index

...

Zugrundeliegende Klasse (Aus dem Geschäftsmodell)

```
class BEARBEITUNGS_CONTROLLER  feature
        text : LIST [STRING]
        position: LIST_ITERATOR [STRING]
        löschen
                -- Lösche Zeile an aktueller Position.
            require
                not position.off
            do
                position.remove
            end
        rechts_einfügen (linie : STRING)
                -- Füge linie nach der aktuellen Position ein.
            require
                not position.after
            do
                position.put_right (line)
            end
        ... Auch: item, index, go_ith, put_left ...
end
```

Eine Befehlsklasse (Skizze, ohne Verträge)

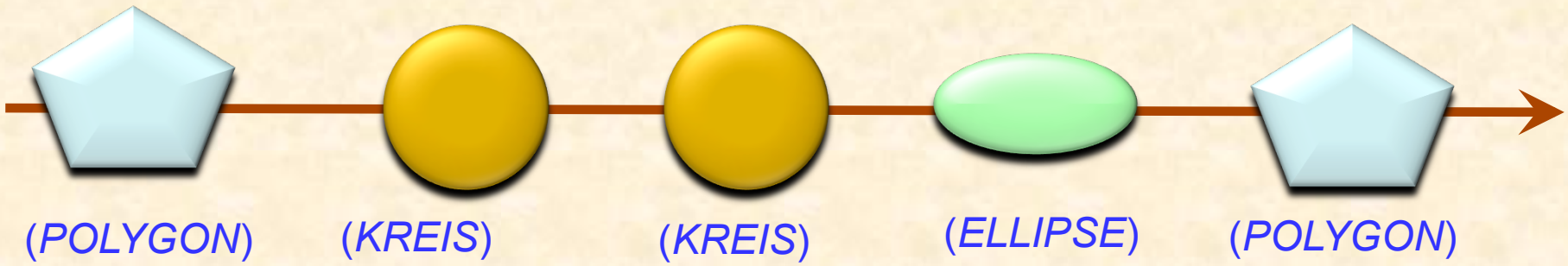
```
class LÖSCHEN  inherit BEFEHL feature
        controller : BEARBEITUNGS_CONTROLLER
                        -- Zugriff zum Geschäftsmodell.
        linie : STRING
                        -- Zu löschende Zeile.
        index : INTEGER
                        -- Position der zu löschenden Zeile.

        ausführen
                        -- Lösche aktuelle Zeile und speichere sie.
                do      linie := controller.item ; index := controller.index
                        controller.löschen ; done := True
                end

        rückgängig_machen
                        -- Füge vorher gelöschte Zeile wieder ein.
                do      controller.go_i_th (index)
                        controller.put_left (line)
                end
end
```

Die Verlauf-Liste

Eine polymorphe Datenstruktur

verlauf : *LIST* [*BEFEHL*]

Erinnerung: Liste von Figuren



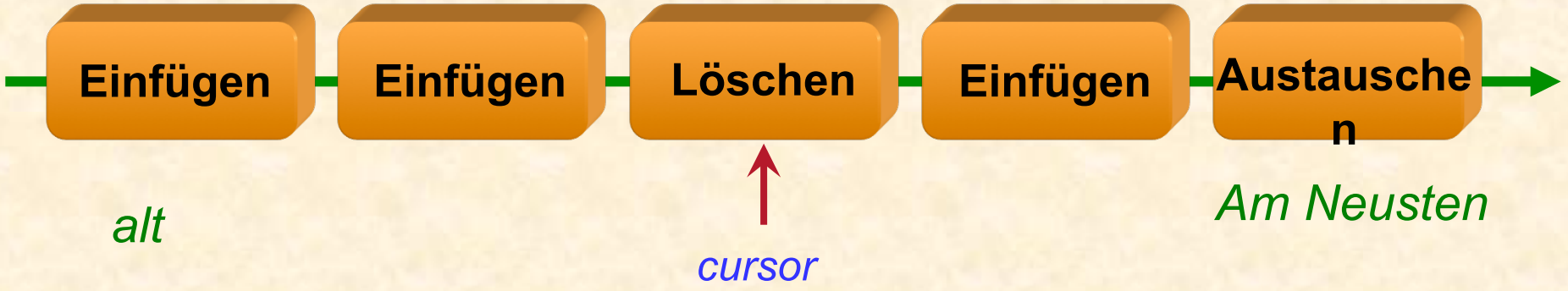

```
bilder.extend (p1 ) ; bilder.extend (c1 ) ; bilder.extend (c2 )
bilder.extend (e ) ; bilder.extend (p2 )
```

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

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

Die Verlauf-Liste

Eine polymorphe Datenstruktur



verlauf : *LIST* [*BEFEHL*]

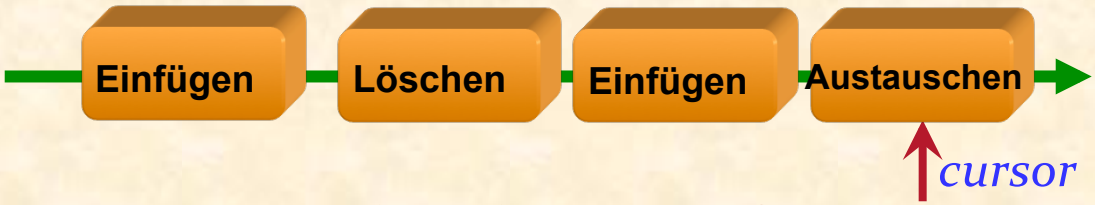
cursor: *ITERATION_CURSOR* [*BEFEHL*]

Einen Benutzerbefehl ausführen

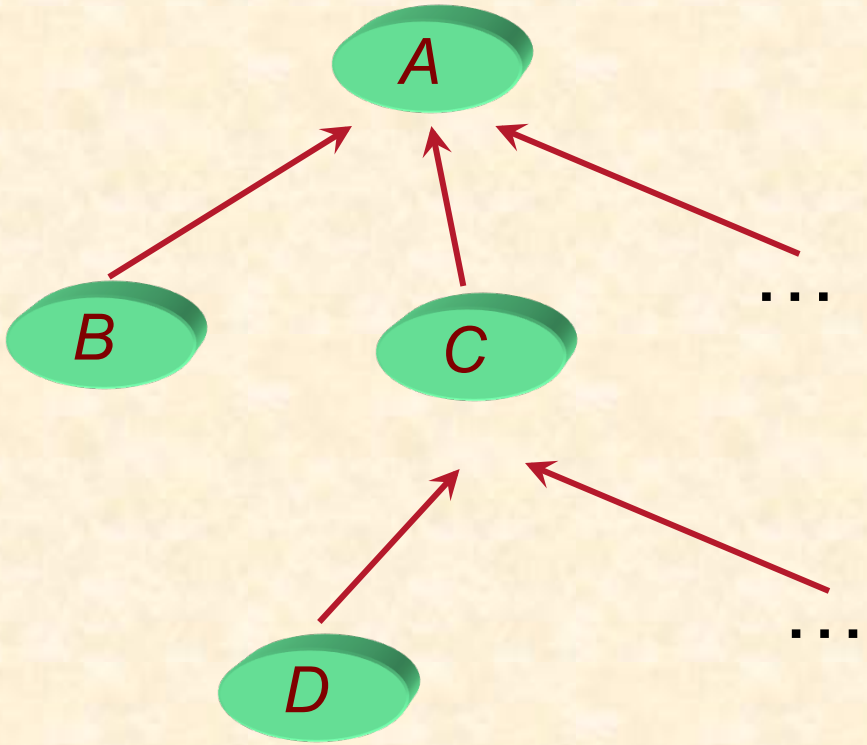
benutzeranfrage_decodieren

if "Anfrage ist normaler Befehl" **then**
"Erzeuge ein Befehlsobjekt *c* , der Anforderung entsprechend"
from until *cursor.is_last* **loop** *cursor.remove_right* **end**
verlauf.extend (c); cursor.forth
c.ausführen
elseif "Anfrage ist UNDO" **then**
if not *cursor.before* **then** -- Ignoriere überschüssige Anfragen
cursor.item.rückgängig_machen
cursor.back
end
elseif "Anfrage ist REDO" **then**
if not *cursor.is_last* **then** -- Ignoriere überschüssige Anfragen
cursor.forth
cursor.item.ausführen
end
end

Pseudocode, siehe nächste Implementation



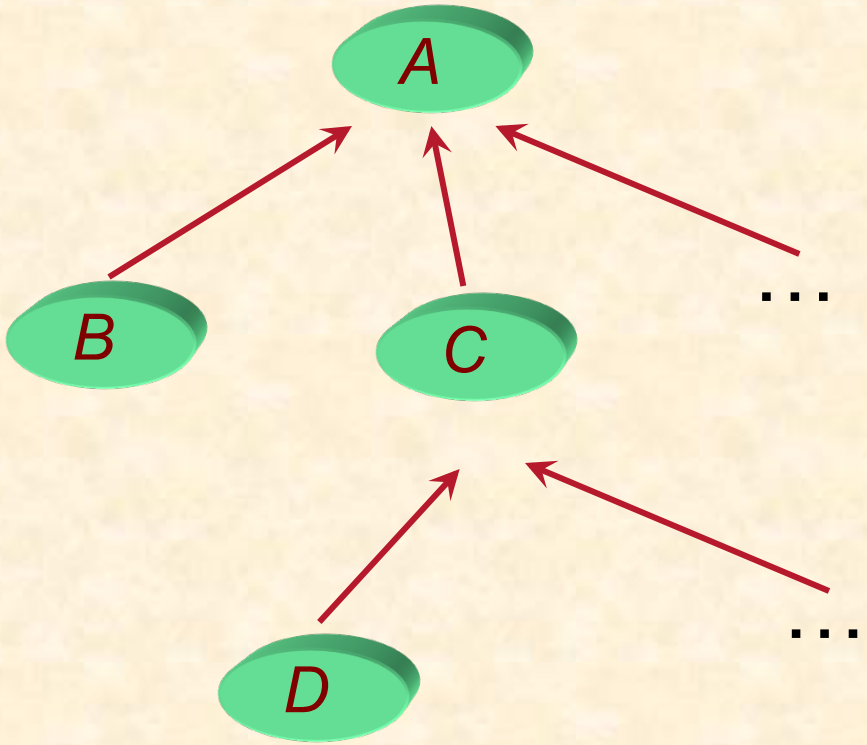
Bedingte Erzeugung (1)




```
a1 : A
if kondition_1 then
    -- "Erzeuge a1 als Instanz von B"
elseif kondition_2 then
    -- "Erzeuge a1 als Instanz von C"
... etc.
```

```
a1 : A; b1 : B ; c1 : C ; d1 : D ; ...
if kondition_1 then
        create b1.make (...)
        a1 := b1
elseif kondition_2 then
        create c1.make (...)
        a1 := c1
... etc.
```

Bedingte Erzeugung (2)



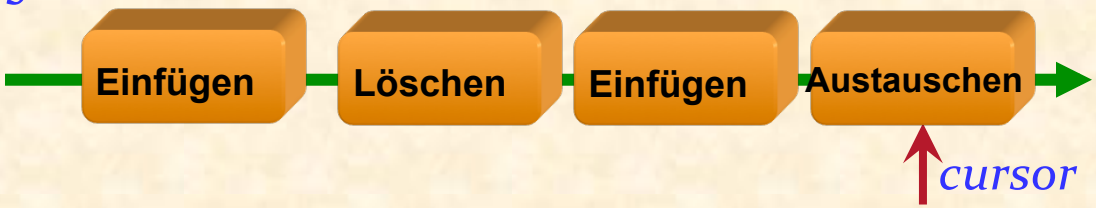

```
a1 : A
if kondition_1 then
    -- "Erzeuge a1 als Instanz von B"
elseif kondition_2 then
    -- "Erzeuge a1 als Instanz von C"
... etc.
```

```
a1 : A
if kondition_1 then
        create {B} a1.make (...)
elseif kondition_2 then
        create {C} a1.make (...)
... etc.
```

Einen Benutzerbefehl ausführen

benutzeranfrage_decodieren

```
if "Anfrage ist normaler Befehl" then
        "Erzeuge ein Befehlsobjekt c , der Anforderung entsprechend"
        from until cursor.is_last loop cursor.remove_right end
        verlauf.extend (c); cursor.forth
        c.ausführen
elseif "Anfrage ist UNDO" then
        if not cursor.before then -- Ignoriere überschüssige Anfragen
           cursor.item.rückgängig_machen
            cursor.back
        end
elseif "Anfrage ist REDO" then
        if not cursor.is_last then –– Ignoriere überschüssige Anfragen
            cursor.forth
           cursor.item.ausführen
        end
end
```



Befehlsobjekte erzeugen: Erster Ansatz

c : *BEFEHL*

...

benutzerbefehl_decodieren

if *"Anfrage ist* **Löschen***"* **then**

    create {*LÖSCHEN*} *c*

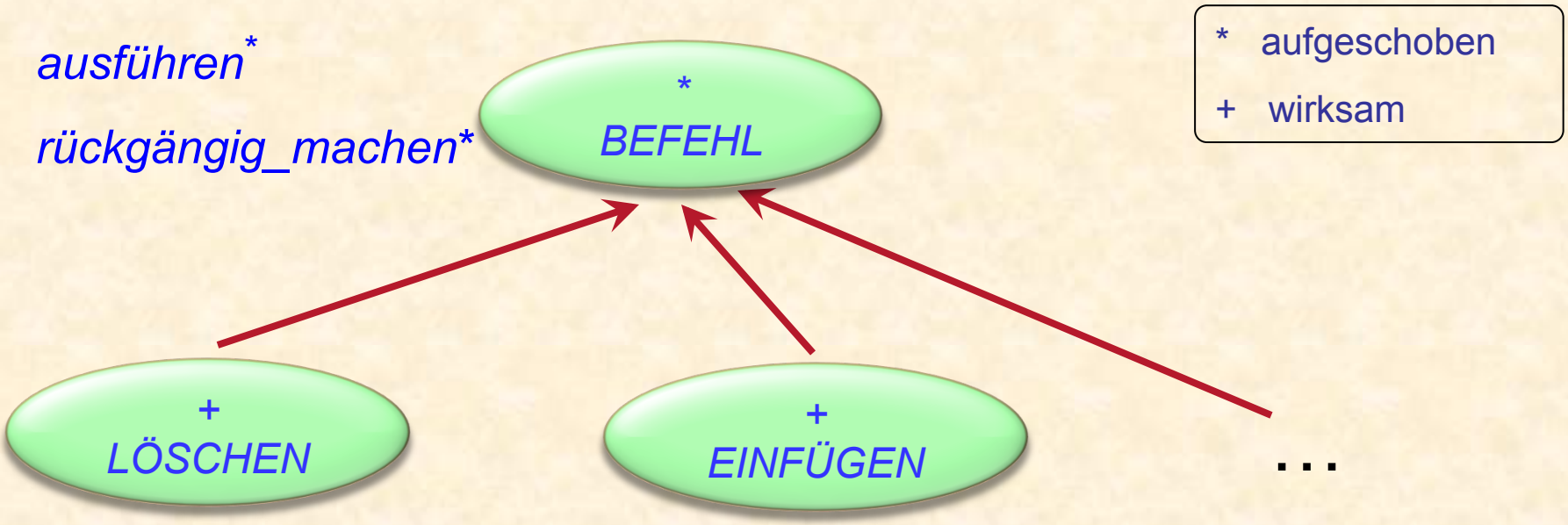
elseif *"Anfrage ist* **Einfügen***"* **then**

    create {*EINFÜGEN*} *c*

else

    etc.

Die Befehl-Klassenhierarchie



ausführen+
rückgängig_machen+
linie: STRING
index: INTEGER

...

ausführen+
rückgängig_machen+
index

...

Befehlsobjekte erzeugen: Besserer Ansatz

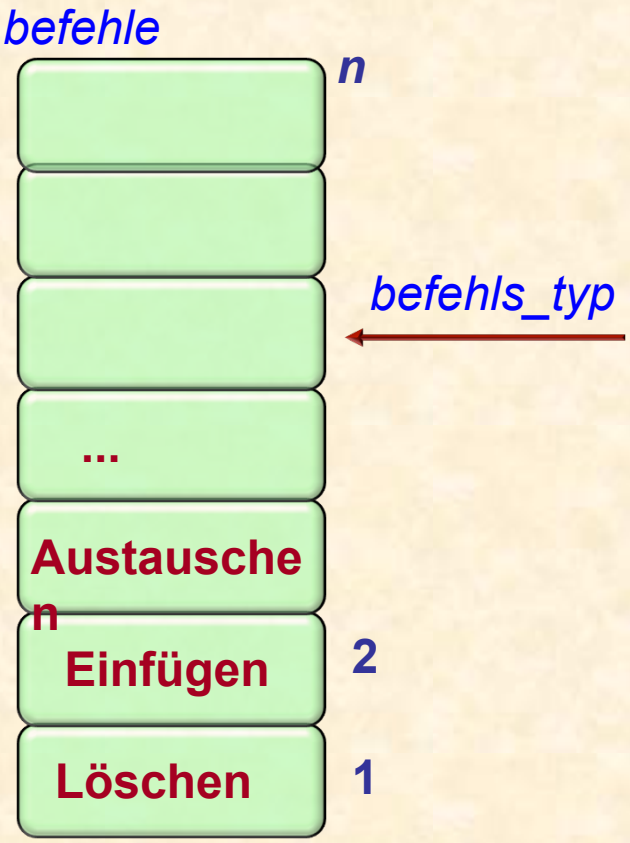
Geben Sie jedem Befehls-Typ eine Nummer

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

Um neue Befehlsobjekte zu erhalten:

"Bestimme *befehls_typ*"

c := (*befehle* [*befehls_typ*])*.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

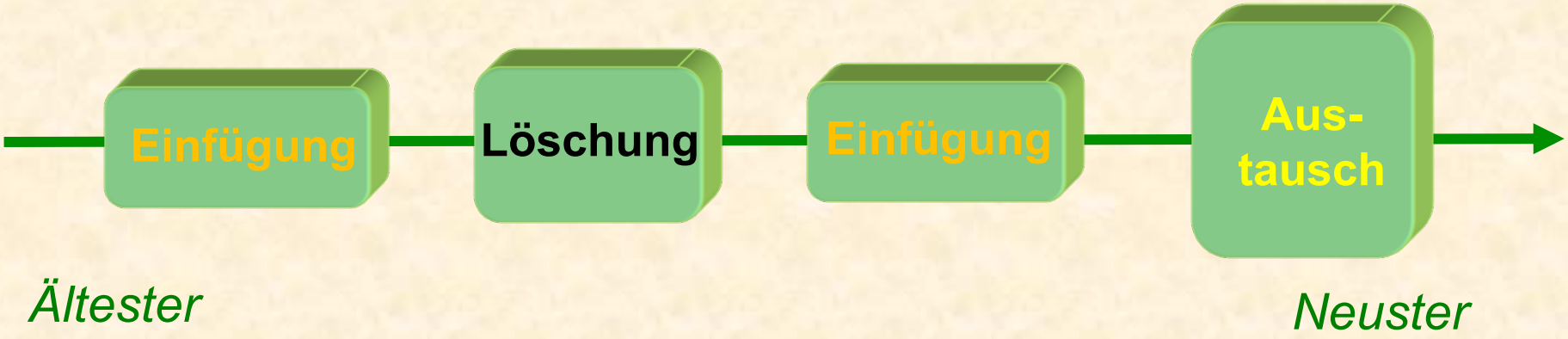
Der Gebrauch von Agenten

Für jeden Benutzerbefehl haben wir zwei Routinen:

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

Die Verlauf-Liste im Undo/Redo-Pattern

verlauf: *LIST* [*BEFEHL*]



Die Verlauf-Liste mit Agenten

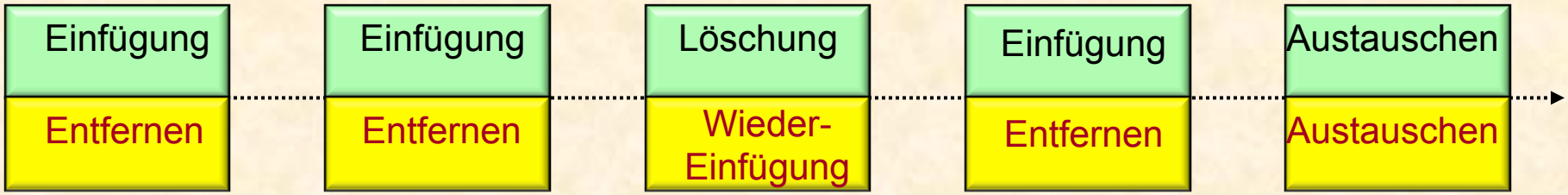
Die Verlauf-Liste wird einfach zu einer Liste von Agentenpaaren:

verlauf: *LIST* [*TUPLE*

[*ausführer* : *PROCEDURE* [*ANY, TUPLE*];

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

Benanntes Tupel

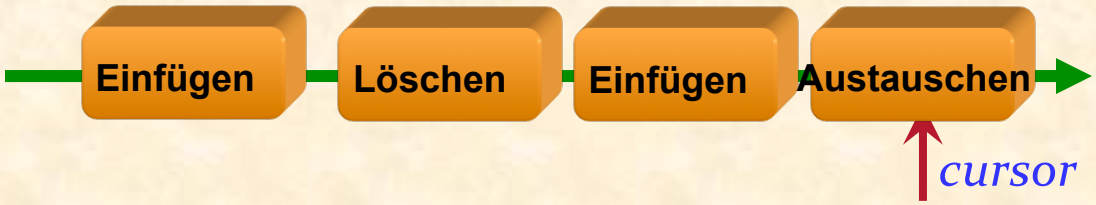


Das Grundschema bleibt dasselbe, aber man braucht nun keine Befehlsobjekte mehr; die Verlauf-Liste ist einfach eine Liste, die Agenten enthält.

Einen Benutzerbefehl ausführen (vorher)

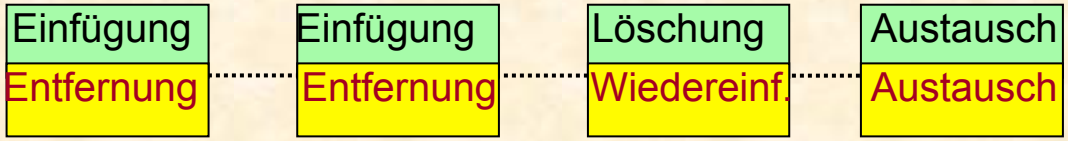
benutzeranfrage_decodieren

if "Anfrage ist normaler Befehl" **then**
    "Erzeuge ein Befehlsobjekt *c* , der Anforderung entsprechend"
    from until *cursor.is_last* **loop** *cursor.remove_right* **end**
    verlauf.extend (*c*); cursor.forth
    c.ausführen
elseif "Anfrage ist UNDO" **then**
    if not *cursor.before* **then** -- Ignoriere überschüssige Anfragen
        cursor.item.rückgängig_machen
        cursor.back
    end
elseif "Anfrage ist REDO" **then**
    if not *cursor.is_last* **then** -- Ignoriere überschüssige Anfragen
        cursor.forth
        cursor.item.ausführen
    end
end

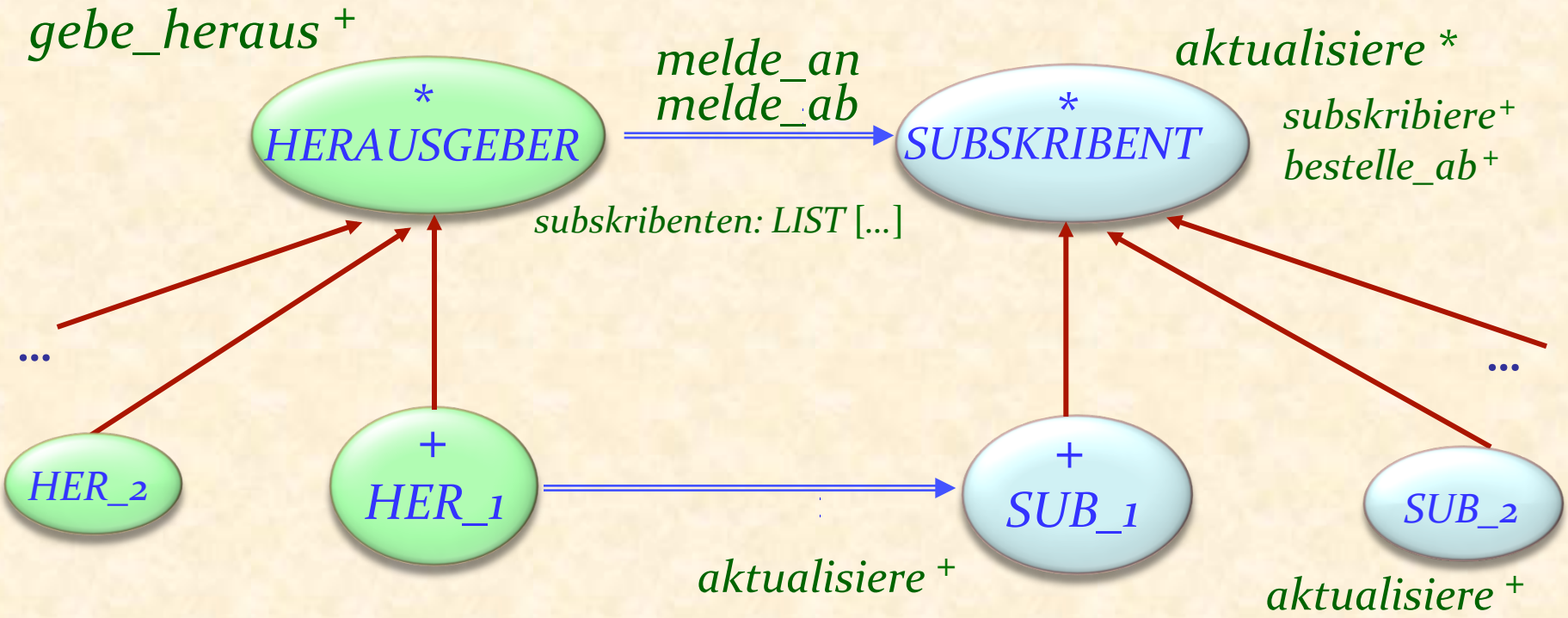


Einen Benutzerbefehl ausführen (jetzt)

```
"Dekodiere Benutzeranfrage mit zwei Agenten do_it and undo_it"
if "Anfrage ist normaler Befehl" then
        from until cursor.is_last loop cursor.remove_right end
        verlauf.extend ([do_it, undo_it ]); cursor.forth
        do_it.call ([])
elseif "Anfrage ist UNDO" then
        if not cursor.before then -- Ignoriere überschüssige Anfragen
          cursor.item.rückgängig_macher.call ([])
           cursor.back
        end
elseif "Anfrage ist REDO" then
        if not cursor.is_last then -- Ignoriere überschüssige Anfragen
           cursor.forth
          cursor.item.ausführer.call ([])
        end
end
```



Erinnerung: das Beobachter-Muster



* aufgeschoben (*deferred*)

\+ wirksam (*effective*)

Erbt von

Kunde (benutzt)

Erinnerung: Ereignis-orientierte Programmierung mit Agenten

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_posititon*, *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
```

Was wir gesehen haben

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
- Mit Agenten noch besser!