ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lecture 18: Undo/Redo

Weiterführende 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.

Wieso machen wir das?

Nützlich in jeder interaktiven Anwendung

- Denken Sie nicht einmal daran, ein interaktives System ohne Undo/Redo-Mechanismus zu schreiben.

Nützliches Entwurfsmuster ("Command-Pattern")

Illustration genereller Algorithmik und Datenstrukturen

Ein Rückblick auf O-O-Techniken: Vererbung, aufgeschobene Klassen, polymorphe Datenstrukturen, dynamisches Binden…

Schön und elegant

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 eine Zeilen-orientierte Ansicht aus Einfachheitsgründen, aber die Diskussion kann auch auf kompliziertere Ansichten angewendet werden.

Zugrundeliegende Klasse (aus dem Modell; vgl MVC)

```
class EDIT_CONTROLLER feature
        text : TWO_WAY_LIST[STRING]
        remove
                        -- Lösche Linie an aktueller Position.
                require
                        not off
                do
                        text.remove
                end
        put_right (line : STRING)
                        -- Füge line nach der aktuellen Position ein.
                require
                        not after
                do
                        text.put_right (line)
                end
        ... Auch: item, index, go_ith, put_left ...
end
```

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, im speziellen 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"

Die „History“ einer Sitzung speichern

Die History-Liste:

Am Ältesten

Am Neusten

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

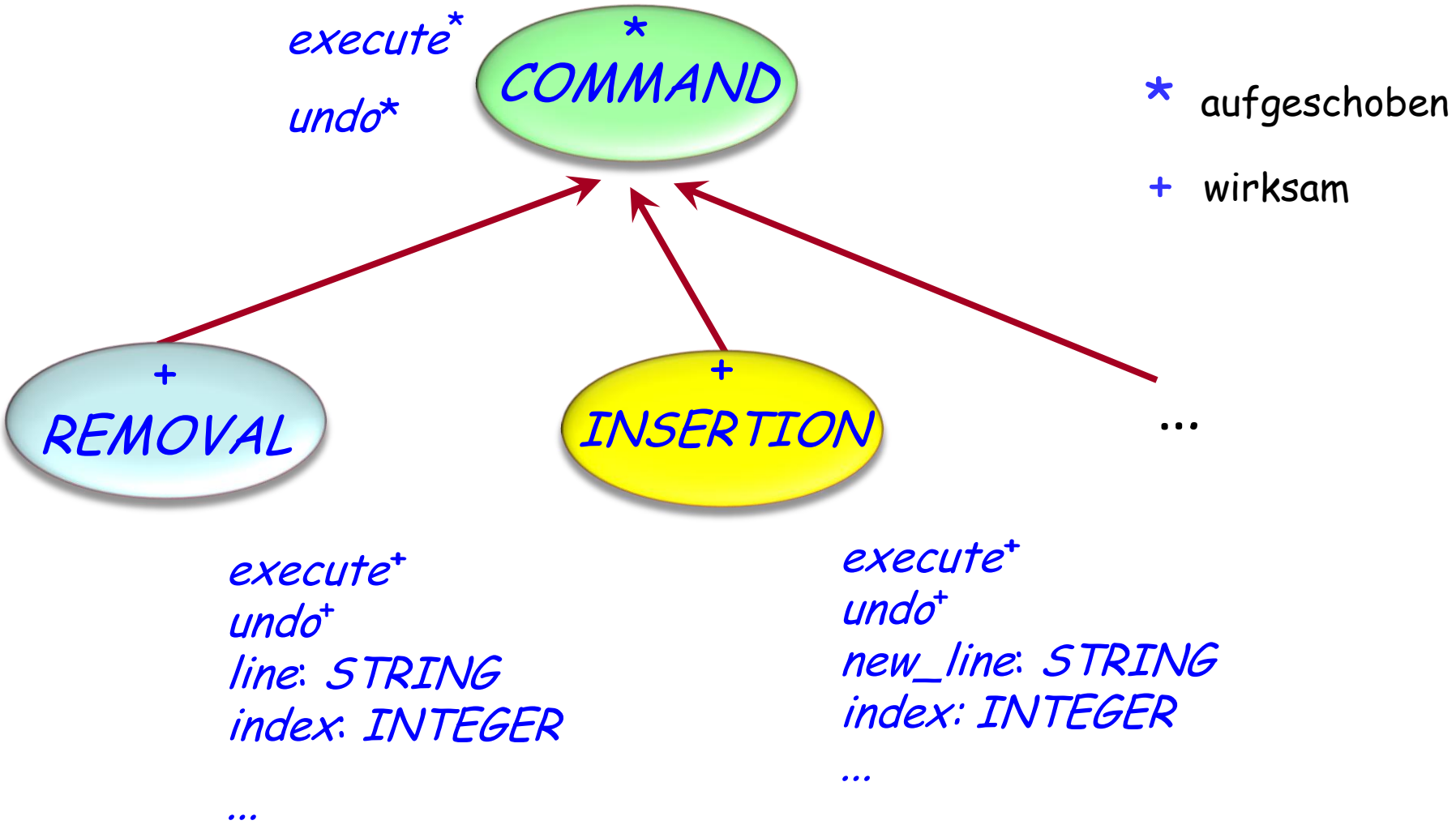
Beispiel: In einem “**Löschungs**”-Objekt brauchen wir:

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

Genereller Begriff eines Befehls (Klasse COMMAND)

```
deferred class COMMAND feature
    done: BOOLEAN
            -- Wurde dieser Befehl ausgeführt?
    execute
            -- Befehl einmal ausführen.
        require not_yet: not done
        deferred
        ensure already: done
        end
    undo
            -- Frühere Ausführung rückgängig machen.
        require already: done
        deferred
        ensure not_yet: not done
        end
end
```

Hierarchie der Befehlsklassen



Zugrunde liegende Klasse

```
class EDIT_CONTROLLER feature
        text : TWO_WAY_LIST[STRING]
        remove
                        -- Lösche Linie an aktueller Position.
                require
                        not off
                do
                        text.remove
                end
        put_right (line : STRING)
                        -- Füge line nach der aktuellen Position ein.
                require
                        not after
                do
                        text.put_right (line)
                end
```

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

```
end
```

Eine Befehlsklasse (1)

```
class REMOVAL inherit COMMAND feature
        make (t: EDITABLE_TEXT)
                        -- Speichere Text, Linie und Position
                do
                        text := t
                        index := text.index
                        line := text.item
                end

        text: EDITABLE_TEXT
                        -- Zugriff zum Geschäftsmodell.
        line : STRING
                        -- Zu löschende Linie.
        index : INTEGER
                        -- Position der zu löschenden Linie.
        ...
```

Eine Befehlsklasse (2)

```
        ...
        execute
                        -- Lösche aktuelle Linie.
                do
                        text.go_i_th (index)
                        text.remove
                        done := True
                end
        undo
                        -- Wiedereinfügen einer gelöschten Linie
                do
                        text.go_i_th (index)
                        text.put_left (line)
                        done := False
                end
end
```

Die „History“ einer Sitzung speichern

Eine polymorphe Datenstruktur:

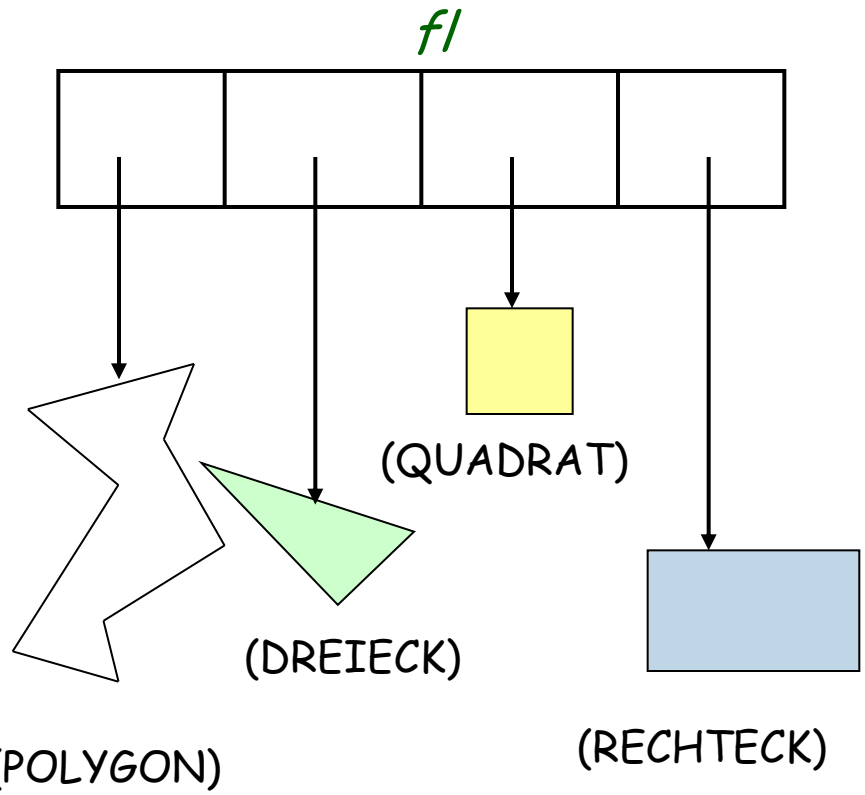
Am Ältesten *Am Neusten*

history : TWO_WAY_LIST [COMMAND]

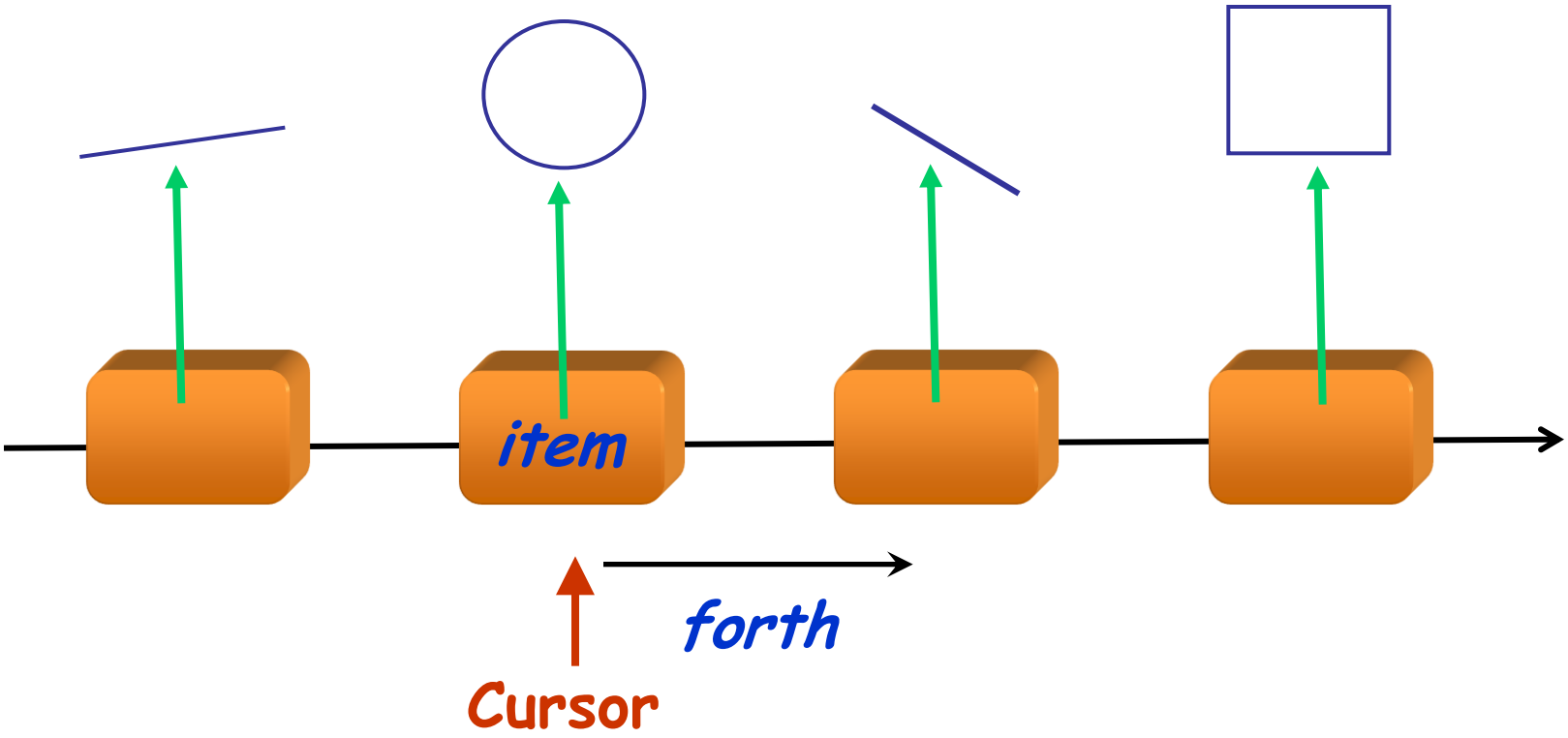
Erinnerung: Die Liste von Figuren

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

fl: LIST[FIGUR]
r: RECHTECK
q: QUADRAT
d: DREIECK
p: POLYGON
...
fl.extend(p); fl.extend(d); fl.extend(q); fl.extend(r)
from fl.start until fl.after loop fl.item.display; fl.forth end
```



Erinnerung: Eine zusammengesetzte Figur als Liste



Die History-Liste

Eine polymorphe Datenstruktur:

Am Ältesten

Am Neusten

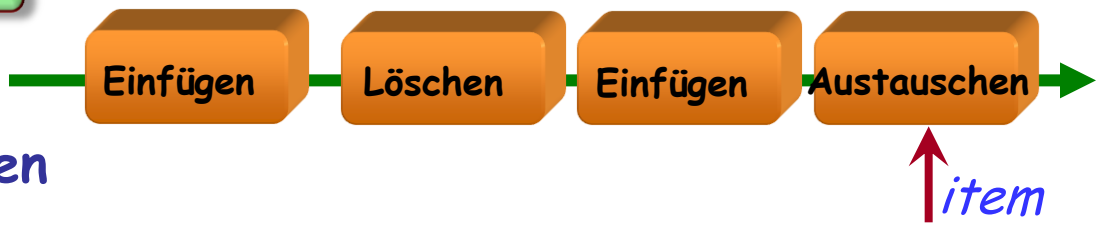
history : TWO_WAY_LIST [COMMAND]

Einen Benutzerbefehl ausführen

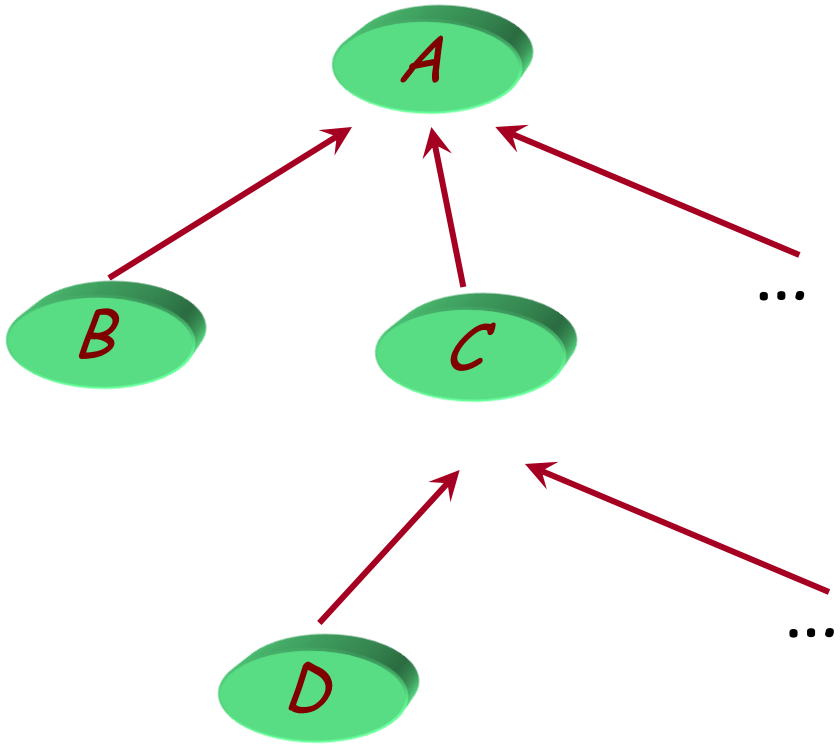
```
decode_user_request

if "Anfrage ist ein 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 nächste Implementation



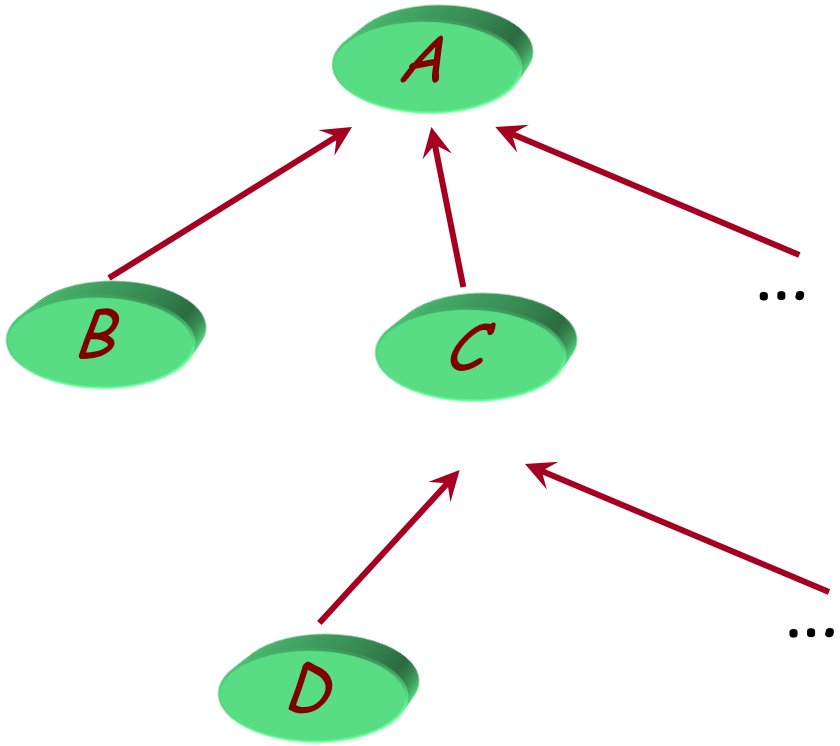
Bedingte Erzeugung (1)




```
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 ein 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
```

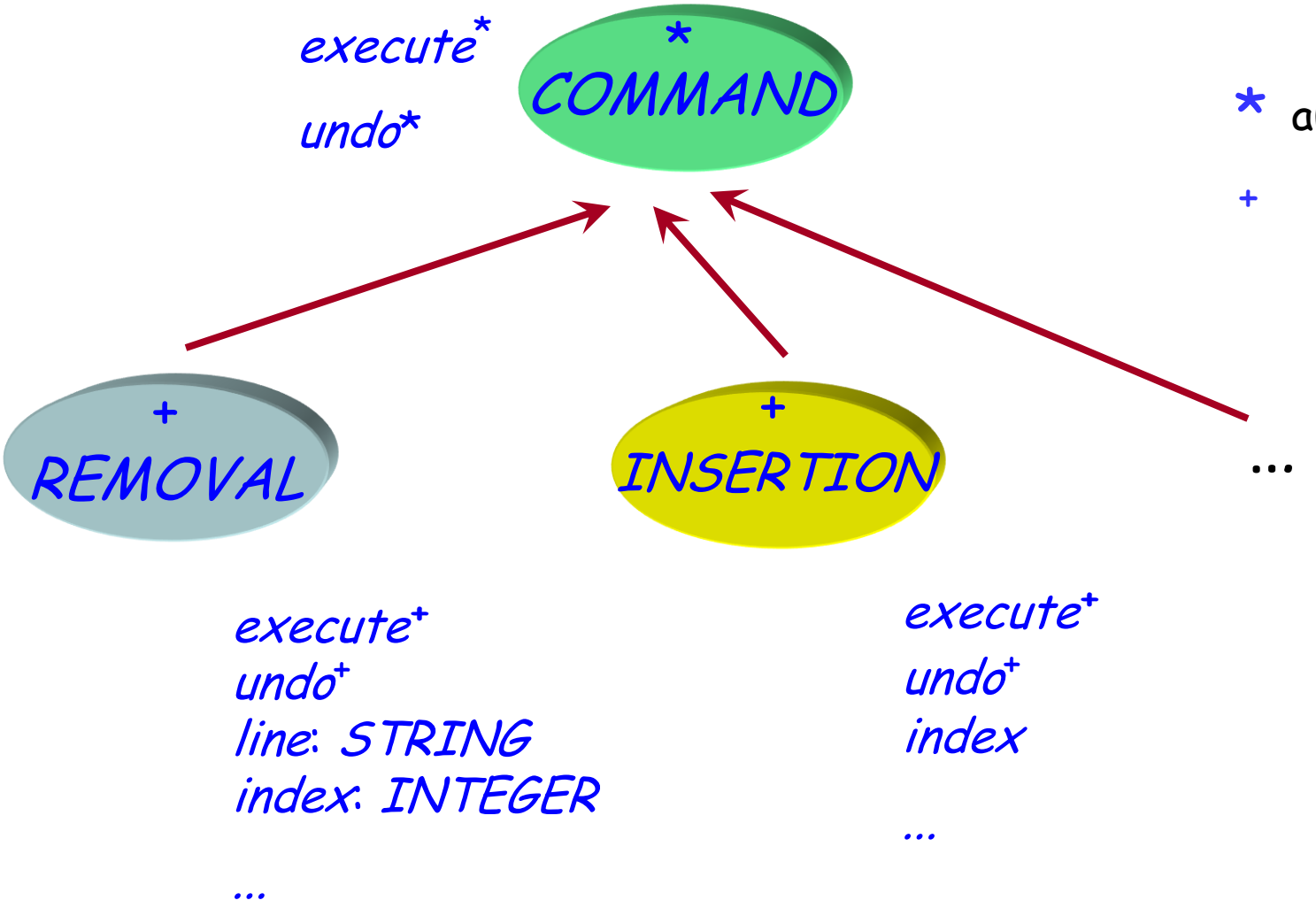
Befehlsobjekte erzeugen: Erster Ansatz

```
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

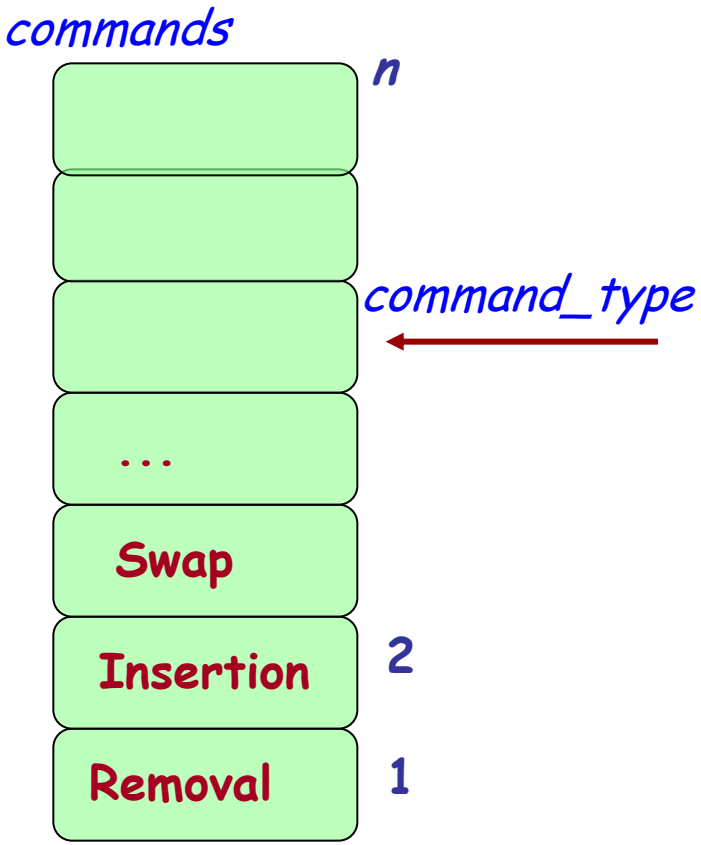
Geben Sie jedem Befehl-Typ eine Nummer

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 (oder Command-) Pattern

- Wurde extensiv genutzt (z.B. in EiffelStudio und anderen Eiffel-Tools)
- Ziemlich einfach zu implementieren
- Details müssen genau betrachtet werden (z.B. sind manche Befehle nicht rückgängig machbar)
- Eleganter Gebrauch von O-O-Techniken
- Nachteil: Explosion kleiner Klassen

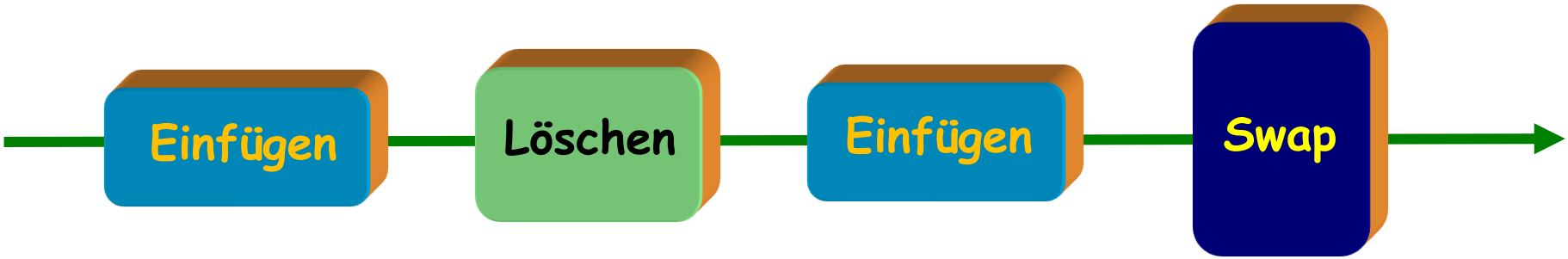
Der Gebrauch von Agenten

Für jeden Benutzerbefehl haben wir zwei Routinen:

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

Die History-Liste im Undo/Redo-Pattern

history : *TWO_WAY_LIST* [*COMMAND*]



Am Ältesten

Am Neusten

Die History-Liste mit Agenten

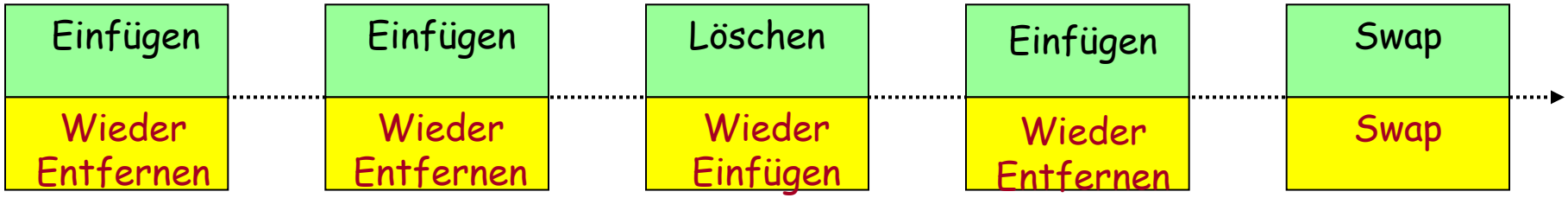
Die History-Liste wird einfach zu einer Liste von Agentenpaaren:

history : *TWO_WAY_LIST* [*TUPLE*

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

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

Benanntes Tupel

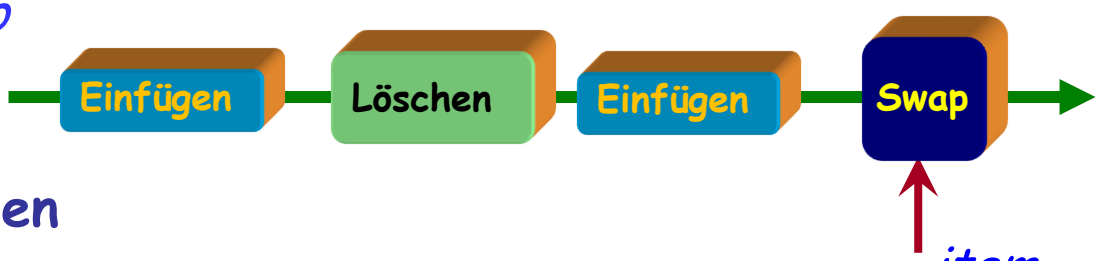


Das Grundschema bleibt dasselbe, aber man braucht nun keine Befehlsobjekte mehr; die History-Liste ist einfach eine Liste, die Agenten enthält.

Einen Benutzerbefehl ausführen (vorher)

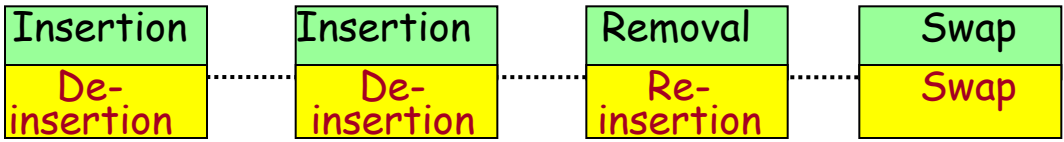
```
decode_user_request

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
```



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!