ETH Zürich
Chair of Software Engineering

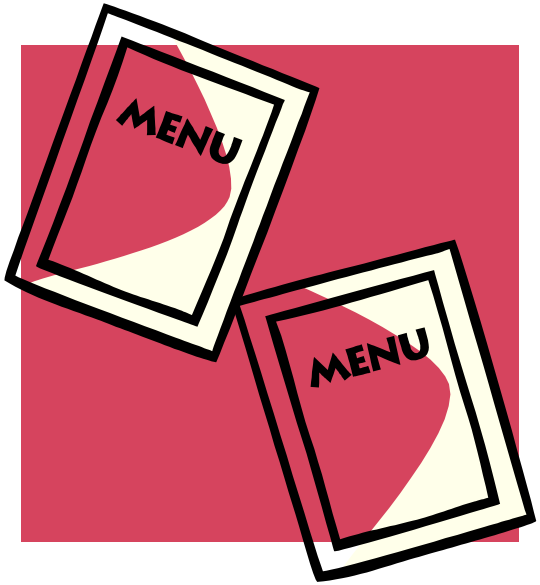
Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 6: Objekterzeugung

In dieser Lektion…

…werden wir sehen, wie man
Objekte erzeugt …



… und ein Paar verwandte Themen
(mehr über Invarianten, Void-Referenzen,
Ausführung eines Systems, Korrektheit…)

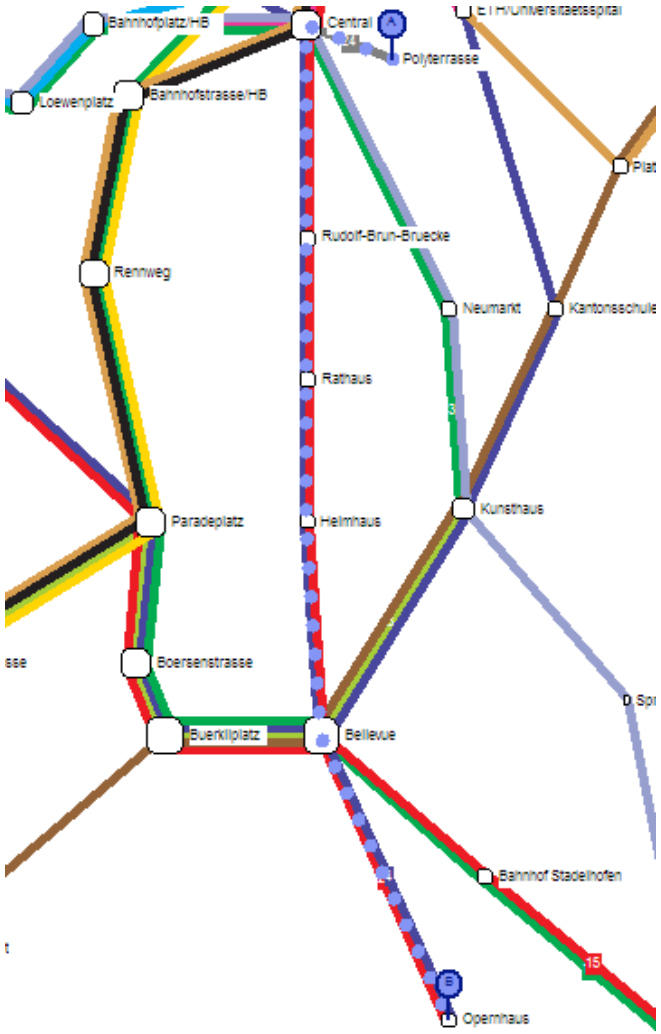
Einfaches Beispiel von Erzeugungsinstruktionen:

```
origin, p1, p2: POINT
…
create origin
create p1.make_cartesian (3, 4)
create p2.make_polar (5, Pi/4)
```

Objekte erzeugen

In früheren Beispielen haben *Zurich, Polybahn* etc. jeweils vordefinierte Objekte bezeichnet. Wir werden jetzt unsere eigenen Objekte erzeugen

Unser Ziel: Eine Route
von Polyterrasse nach Opernhaus
zu erzeugen



Beispiel: *ROUTE_BUILDING*

```
class ROUTE_BUILDING inherit
    ZURICH_OBJECTS
feature
    build_route
                -- Eine Route bauen und damit arbeiten.
            do
                -- "Opera_route erzeugen und Teilstrecken hinzufügen"
                Zurich.add_route (Opera_route)
                Opera_route.reverse
                -- "Weiter mit Opera_route arbeiten"
            end

    Opera_route : ROUTE
                -- Eine Route von Polybahn nach Opernhaus.
end
```

Pseudocode

Pseudocode

Bezeichnet eine Instanz der Klasse *ROUTE*

Pseudocode

«Pseudocode» bezeichnet Sätze in natürlicher Sprache, die noch nicht geschriebenen Programmcode darstellen.

Beispiel:

-- "*Opera_route* erzeugen und Teilstrecken hinzufügen"

Wir schreiben Pseudocode als Kommentar und in Anführungszeichen

Stil: wenn der wirkliche Code geschrieben wird, ist es eine gute Idee, den Pseudocode als normalen Kommentar im Programm zu behalten

Bezeichner, Entitäten, Variablen

Identifier

Ein **Bezeichner** ist ein vom Programmierer gewählter Name, um ein gewisses Programmelement zu repräsentieren. Beispiele sind:

- Eine Klasse, z.B. *ROUTE*
- Ein Feature, z.B. *i_th*
- Ein Laufzeitwert, wie etwa ein Objekt oder eine Objektreferenz, z.B. *Opera_route*

Entity

Ein Bezeichner, der einen Laufzeitwert bezeichnet, wird **Entität** genannt

Variable

Eine Entität, deren Wert sich ändern kann, ist eine **Variable**

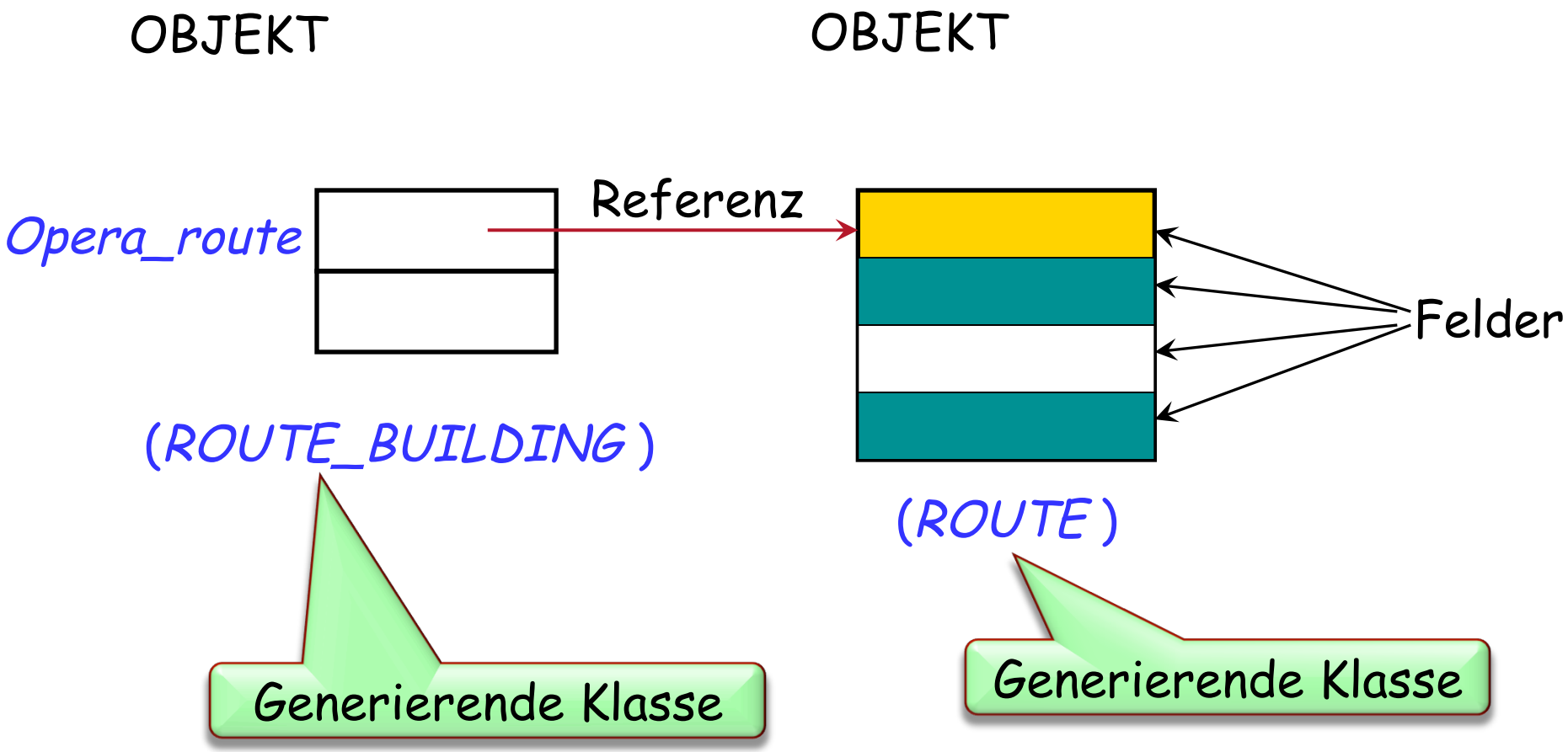
Attached

Während der Ausführung können Entitäten an Objekte **gebunden** sein

An ein Objekt gebundene Entitäten

Im Programm: eine Entität, wie z.B. *Opera_route*

Im Speicher, während der Ausführung: ein Objekt

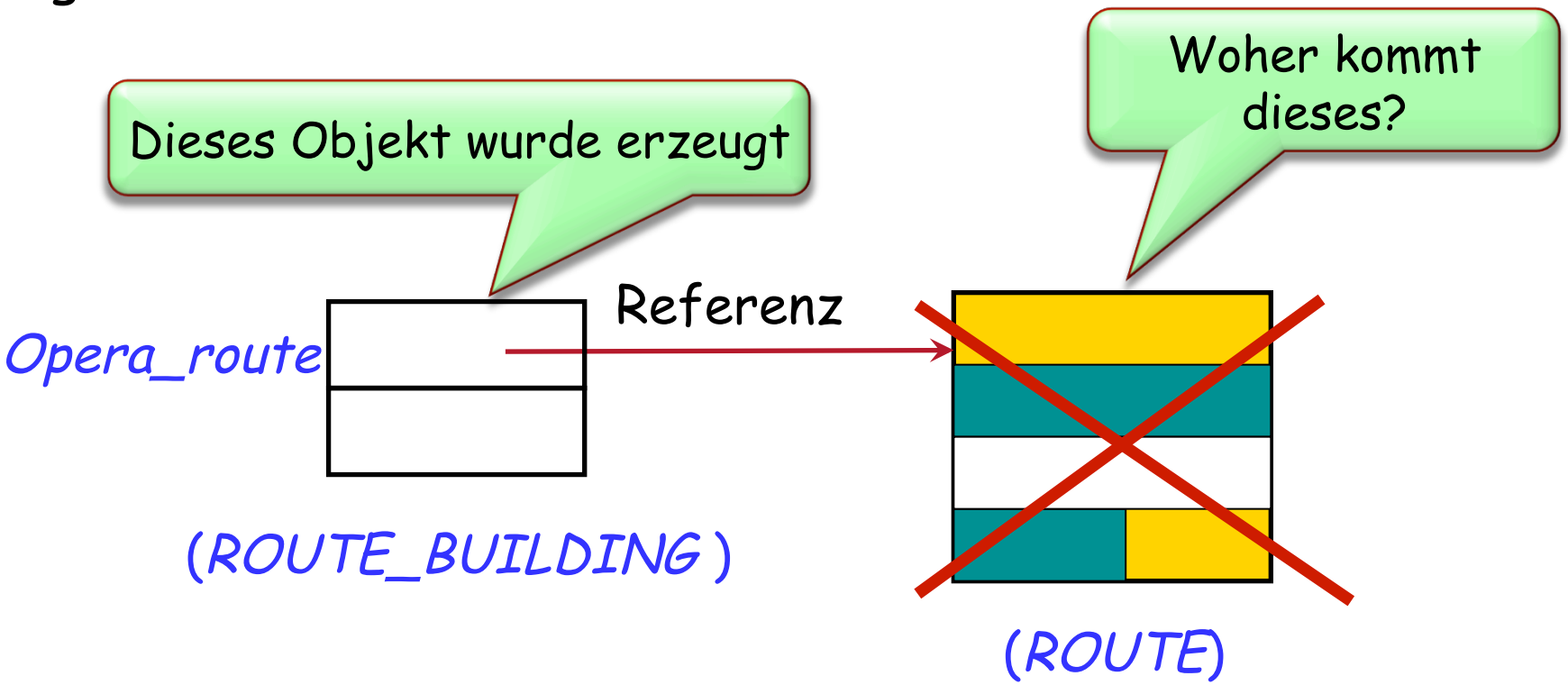


ROUTE_BUILDING

```
class ROUTE_BUILDING inherit
        ZURICH_OBJECTS
feature
        build_route
                        -- Eine Route bauen und damit arbeiten.
            do
                        -- "Opera_route  erzeugen und Teilstrecken hinzufügen"
                        Zurich.add_route (Opera_route)
                        Opera_route.reverse
                        -- "Weiter mit Opera_route arbeiten"

            end

        Opera_route : ROUTE
                        -- Eine Route von Polyterrasse nach Opernhaus.
end
```

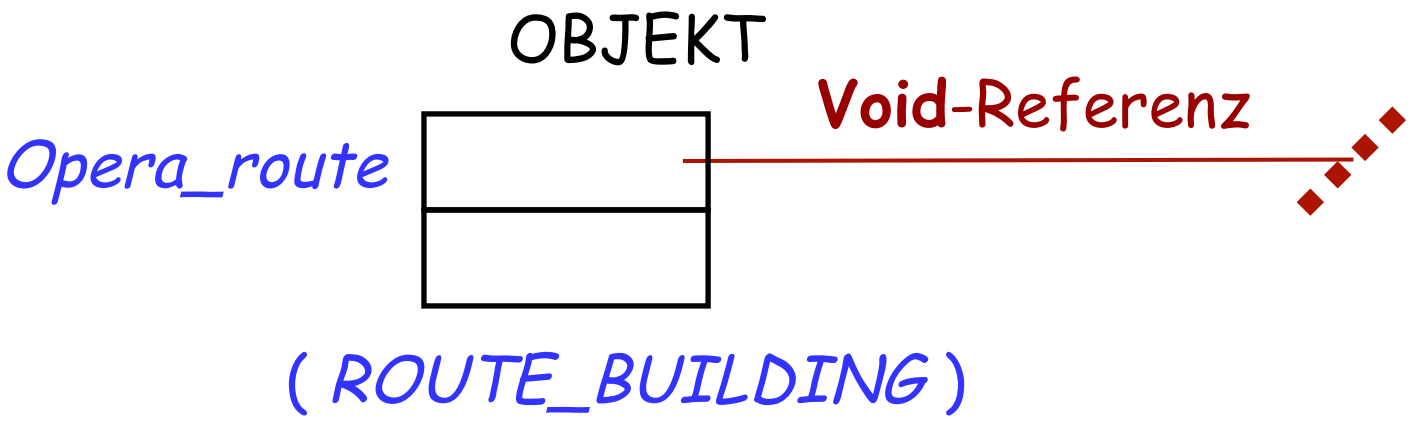
Der Grundzustand einer Referenz

Können wir in einer Instanz von *ROUTE_BUILDING* annehmen, dass *Opera_route* an eine Instanz von *ROUTE* gebunden ist?



Standard-Referenzen

Anfangs ist *Opera_route* nicht an ein Objekt gebunden:
Es ist eine **Void**-Referenz



Warum müssen wir Objekte erzeugen?

Können wir nicht annehmen, dass eine Deklaration der Form

Opera_route : *ROUTE*

eine Instanz von *ROUTE* erzeugt und sie an *Opera_route* bindet?

(Die Antwort darauf folgt bald…)

Zustände von Referenzen

Zur Laufzeit ist eine Referenz

- Entweder an ein gewisses Objekt gebunden
- Oder void

- Um eine Void-Referenz zu bezeichnen: Benutzen Sie das reservierte Wort **Void**
- Um herauszufinden, ob *x* void ist, können Sie folgende Abfrage benutzen:

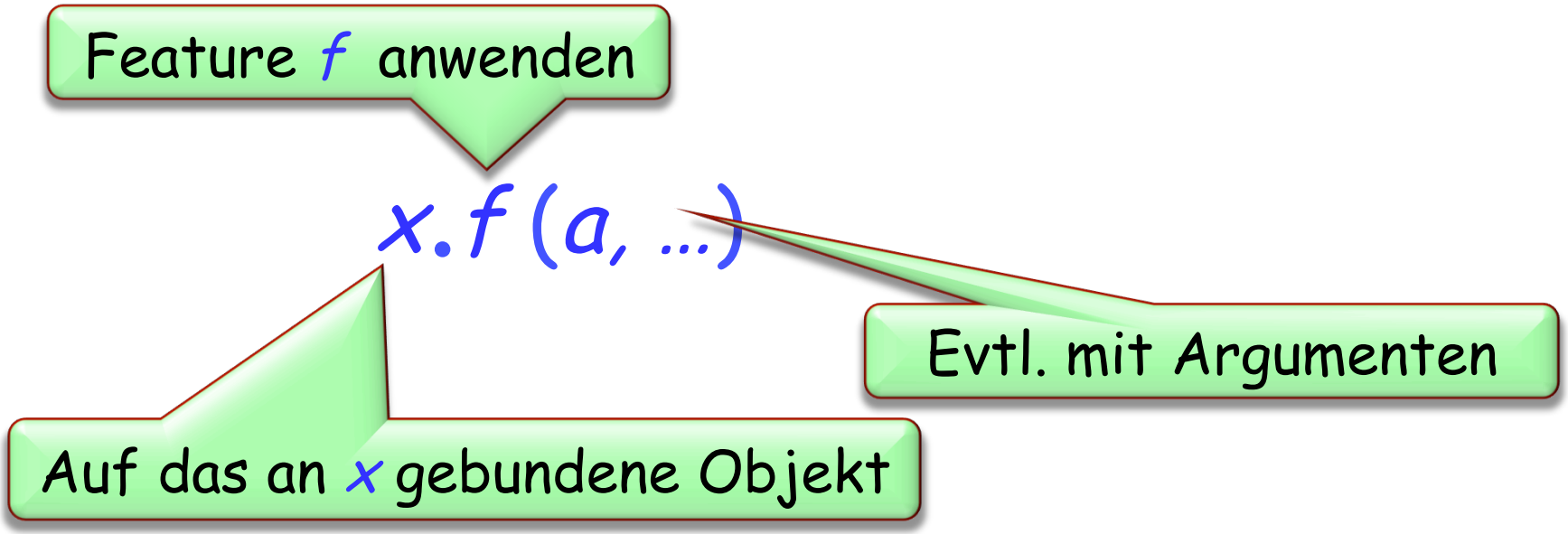
  x = **Void**

- Die inverse Abfrage (ist *x* an ein Objekt gebunden?):

  x /= **Void**

Das Problem mit Void-Referenzen

Der Grundmechanismus von (O-O) Programmen ist der Featureaufruf



Da Referenzen void sein können, kann x möglicherweise an kein Objekt gebunden sein

In solchen Fällen ist der Aufruf fehlerhaft!

Beispiel: Aufruf auf ein Ziel, das void ist

```
class ROUTE_BUILDING inherit
        ZURICH_OBJECTS
 feature
        build_route
                        -- Eine Route bauen und damit arbeiten.
            do
                        -- "Opera_route  erzeugen und Teilstrecken hinzufügen"
                        Zurich.add_route (Opera_route)
                        Opera_route.reverse
            end

        Opera_route : ROUTE
                        -- Eine Route von Polybahn nach Opernhaus.
end
```

Void-Referenz

Ausnahmen (exceptions)

Beispiele von abnormalen Ereignissen (*failures*) während der Ausführung:

- "Void call": *Opera_route.reverse,* wobei *Opera_route* void ist
- Der Versuch, *a / b* auszurechnen, wobei *b* den Wert 0 hat

Die Konsequenz ist ein Abbruch; es sei denn, das Programm hat Code, um sich von der Ausnahme zu „erholen" ("**rescue**" Klausel in Eiffel, "**catch**" in Java)

Jede Ausnahme hat einen **Typ**, der in den Laufzeit-Fehlermeldungen von EiffelStudio ersichtlich ist, z.B.

- Feature call on void target
- Arithmetic underflow
- Zusicherungsverletzung

Objekte explizit erzeugen

Um eine Ausnahme zu vermeiden:

- Verändern Sie die Prozedur *build_route*, so dass sie ein Objekt erzeugt und an *Opera_route* bindet, bevor es *reverse* aufruft

Warum müssen wir Objekte erzeugen?

Können wir nicht annehmen, dass eine Deklaration der Form

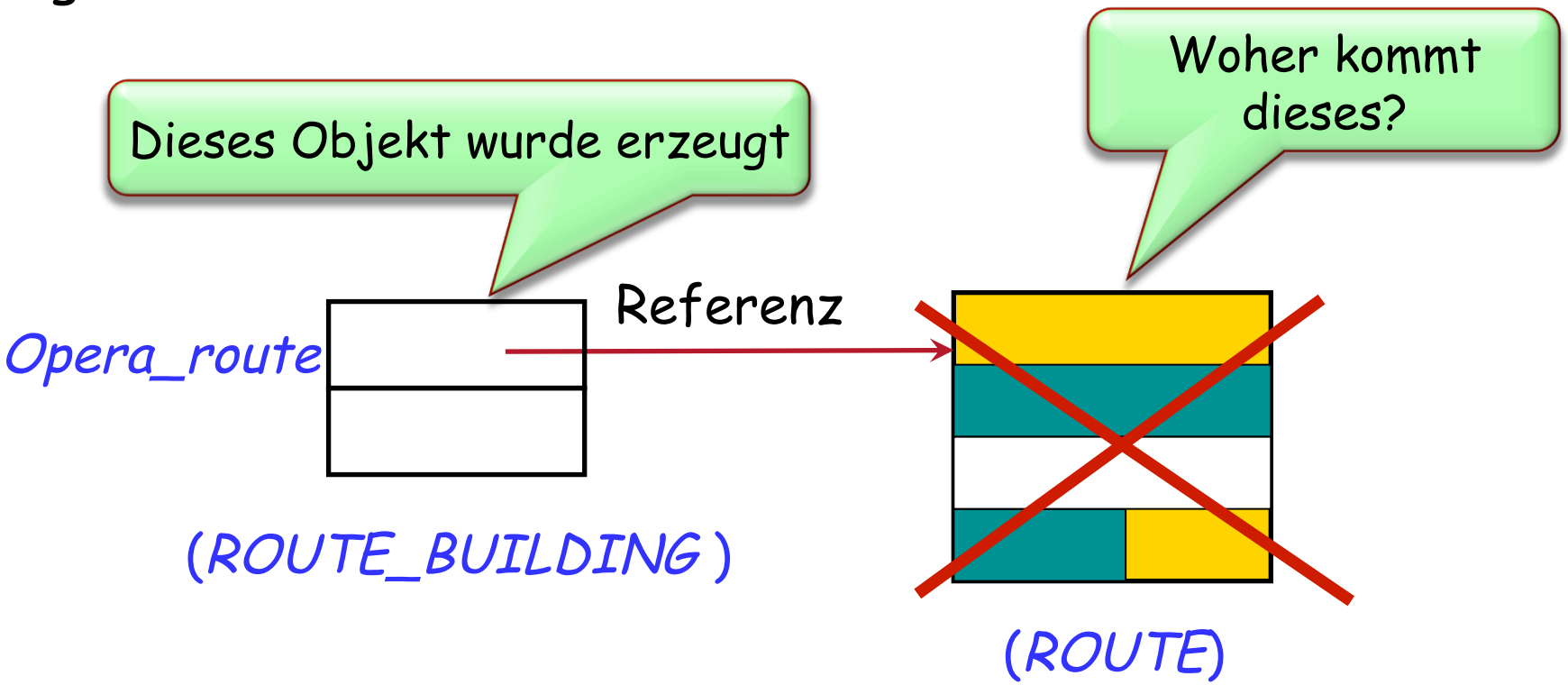
Opera_route : *ROUTE*

eine Instanz von *ROUTE* erzeugt und sie an *Opera_route* bindet?

(Die Antwort darauf folgt bald…)

Der Grundzustand einer Referenz

Können wir in einer Instanz von *ROUTE_BUILDING* annehmen, dass *Opera_route* an eine Instanz von *ROUTE* gebunden ist?

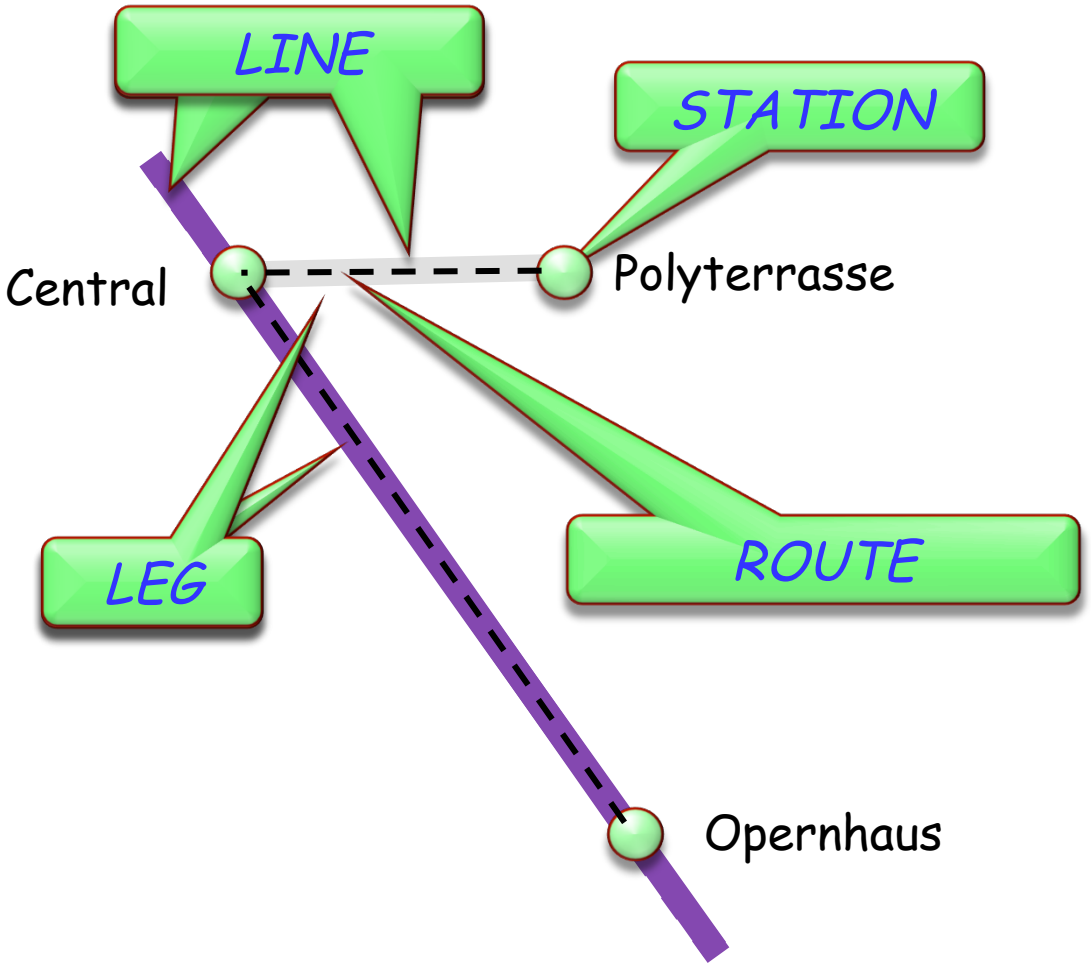


ROUTE_BUILDING

```
class ROUTE_BUILDING inherit
        ZURICH_OBJECTS
feature
        build_route
                        -- Eine Route bauen und damit arbeiten.
            do
                        -- "Opera_route erzeugen und Teilstrecken hinzufügen"
                        Zurich.add_route (Opera_route)
                        Opera_route.reverse
            end


        Opera_route : ROUTE
                        -- Eine Route von Polybahn nach Opernhaus.
end
```

Einfache Objekte erzeugen

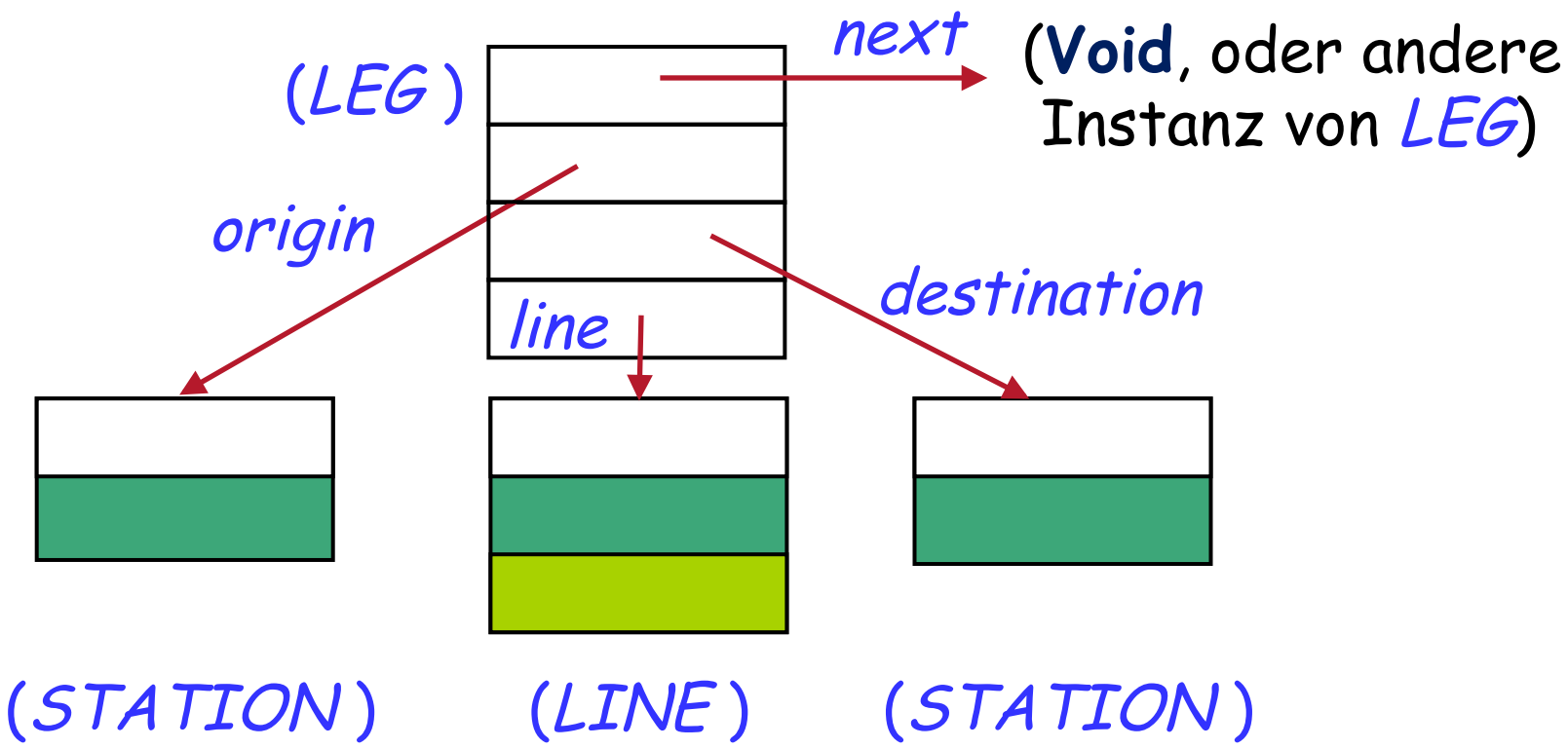
Um *Opera_route* zu erzeugen, müssen wir zuerst Objekte erzeugen, welche die Stationen, Linien und Teilstrecken (*legs*) repräsentieren



Teilstrecken

Eine Instanz der Klasse *LEG* hat:

- Referenzen zu einer Anfangstation, Endstation und einer Linie; dürfen nicht **Void** sein
- Eine Referenz zur nächsten Teilstrecke; **Void** am Schluss



Die Schnittstelle der Klasse *LEG* (erste Version)

```
class LEG feature
        origin, destination : STATION
                -- Anfang und Ende der Teilstrecke.

        line : LINE
                -- Die Linie, der die Strecke angehört.

        next : LEG
                -- Nächste Teilstrecke in der Route.

        make (o, d : STATION ; l : LINE)
                -- Anfang, Ende und Linie setzen.
            require
                linie_existiert: l /= Void
                auf_der_linie: l.has_station (o) and l.has_station (d)
            ensure
                stationen_gesetzt: origin  = o and destination = d
                linie_gesetzt: line = l

        link (other : LEG)
                -- other zur nächsten Teilstrecke der Route machen.
            ensure
                naechste_gesetzt: next = other
end
```

ROUTE_BUILDING

```
class ROUTE_BUILDING inherit
    ZURICH_OBJECTS
 feature


    build_route
            -- Eine Route bauen und damit arbeiten.
      do
            -- Erzeuge Opera_route und fülle sie mit Stationen.


            Zurich.add_route (Opera_route)
            Opera_route.reverse
      end
    Opera_route: ROUTE
            -- Eine Route von Polybahn nach Opernhaus.
end
```

Pseudocode

Eine Instanz von *LEG* erzeugen

```
class ROUTE_BUILDING inherit
    ZURICH_OBJECTS
 feature
        leg1 : LEG
                -- Erste Teilstrecke der Opera_route.
    build_route
                -- Eine Route bauen und damit arbeiten.
        do
            -- Erzeuge Opera_route und fülle sie mit Stationen.
            create leg1
            -- "Mehr Teilstrecken erzeugen und Opera_route  fertig bauen"
            Zurich.add_route (Opera_route)
            Opera_route.reverse
        end
    Opera_route: ROUTE
                -- Eine Route von Polybahn nach Opernhaus.
end
```

Jetzt ein gewöhnlicher Kommentar

Erzeugungs-instruktion

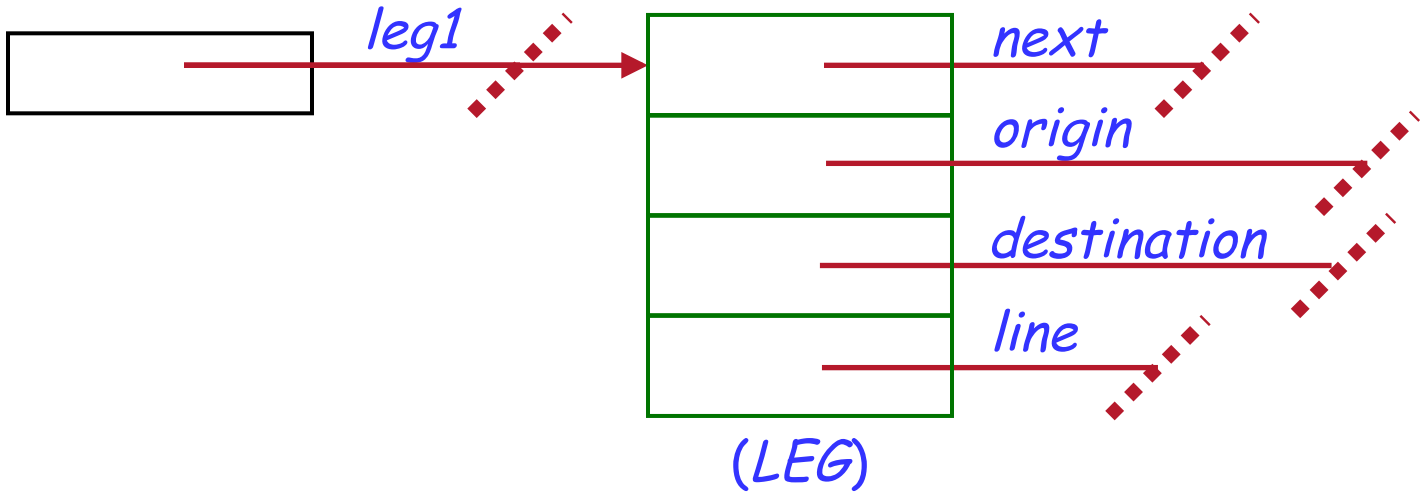
Neuer Pseudocode

Erzeugungsinstruktion

Grundoperation, um Objekte während der Laufzeit zu erzeugen:

- Erzeugt ein neues Objekt im Speicher
- Bindet eine Entität daran

create *leg1*



Der Typ eines erzeugten Objektes

Jede Entität ist mit einem gewissen Typ deklariert:

leg1 : *LEG*

Eine Erzeugungsinstruktion

create *leg1*

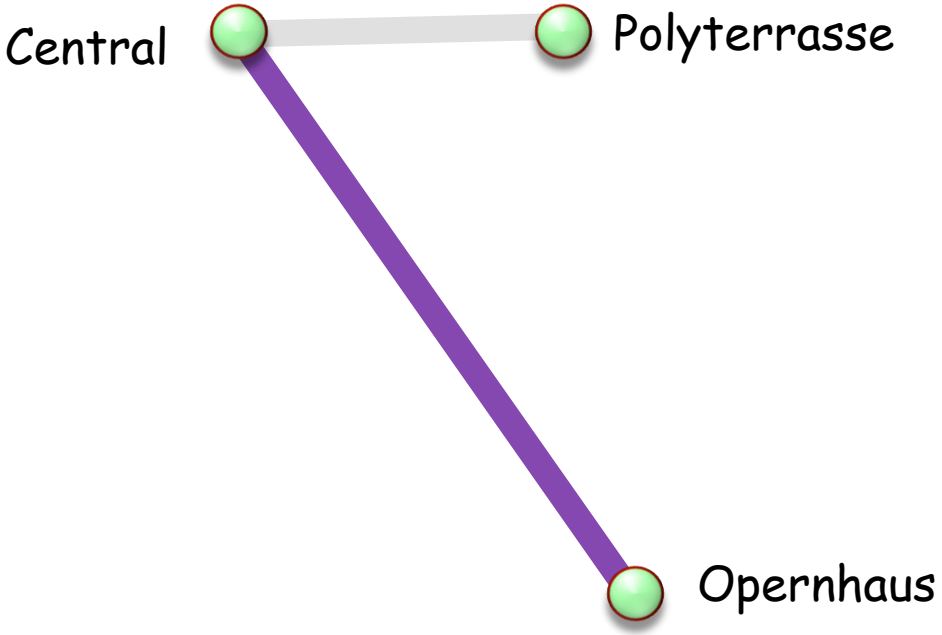
produziert während der Laufzeit ein Objekt dieses Typs

Eine Route mit zwei Teilstrecken

Wir möchten jetzt, dass *leg1* von Polyterrasse nach Central geht, und eine weitere Teilstrecke nach Opernhaus fortfährt.

Als Erstes deklarieren wir die entsprechenden Attribute:

leg1, leg2 : LEG



build_route

```
build_route
        -- Eine Route bauen und damit arbeiten.
    do
        -- Erzeuge die Teilstrecken und setze die Stationen und die Linien.
        create leg1
        leg1 .make (Polyterrasse, Central, Polybahn)
        create leg2
        leg2.make (Central, Opernhaus, Line4)
        -- Verbinde die erste Teilstrecke mit der zweiten.
        leg1.link (leg2)
        -- "Opera_route erzeugen und ihr die eben erstellten Teilstrecken zuweisen"

        Zurich.add_route (Opera_route)

        Opera_route.reverse
    end
```

Vordefiniert in *ZURICH_OBJECTS*

Immer noch Pseudocode!

Nochmals: warum müssen wir Objekte erzeugen?

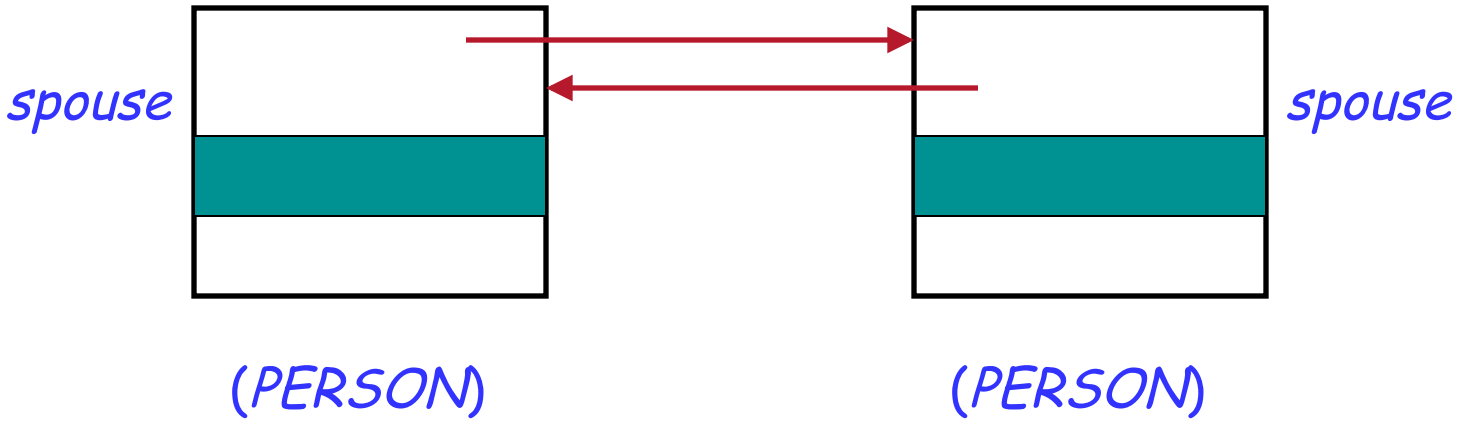
Können wir nicht annehmen, dass eine Deklaration der Form

Opera_route : *ROUTE*

eine Instanz von *ROUTE* erzeugt und sie an *Opera_route* bindet?

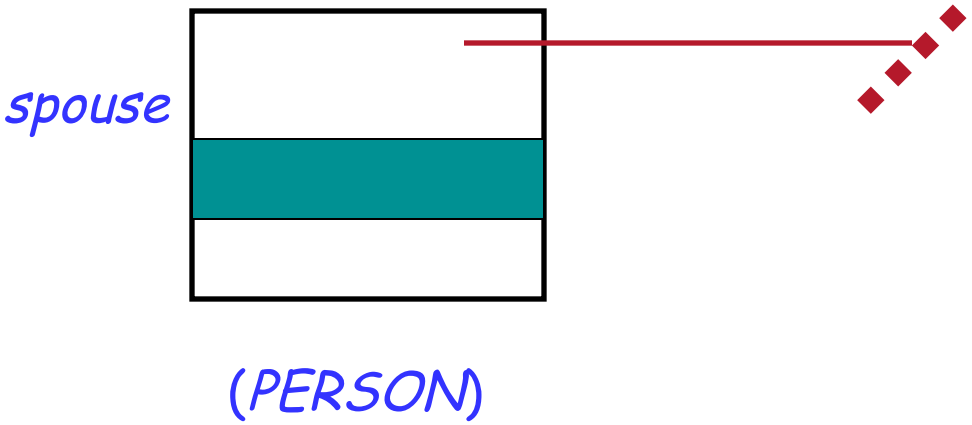
Void-Referenzen sind nützlich!

Verheiratete Personen:



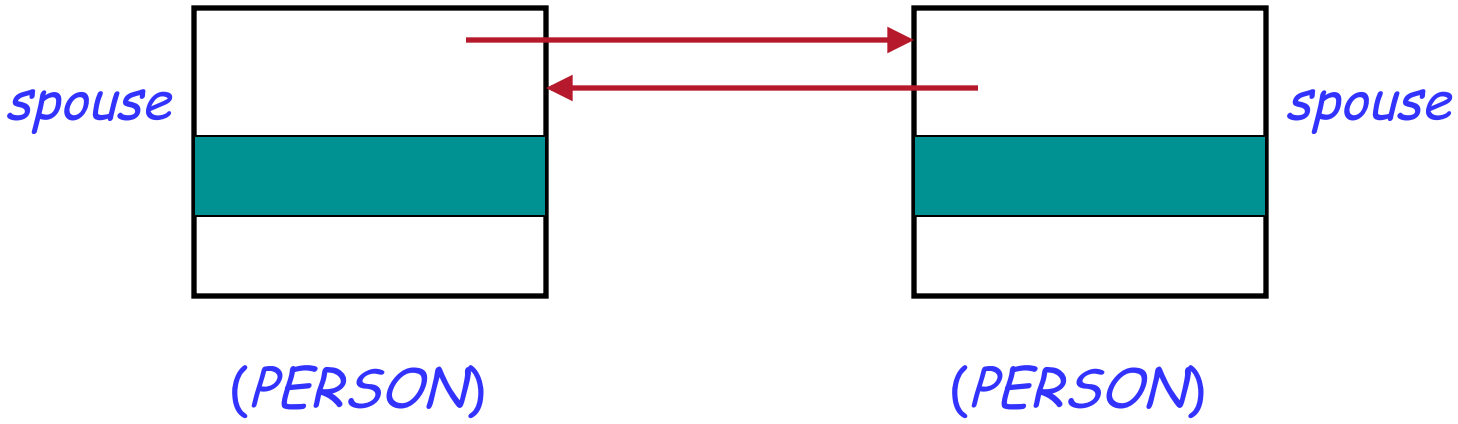
Void-Referenzen sind nützlich!

Ledige Personen:



Void-Referenzen sind nützlich!

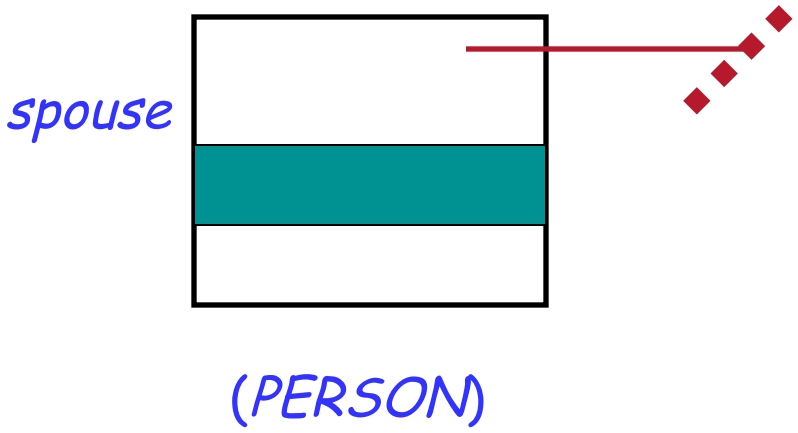
Auch bei verheirateten Personen…



… sollten wir nicht jedes Mal, wenn wir eine Instanz von *PERSON* erzeugen, auch ein Objekt für *spouse* erzeugen? (Warum?)

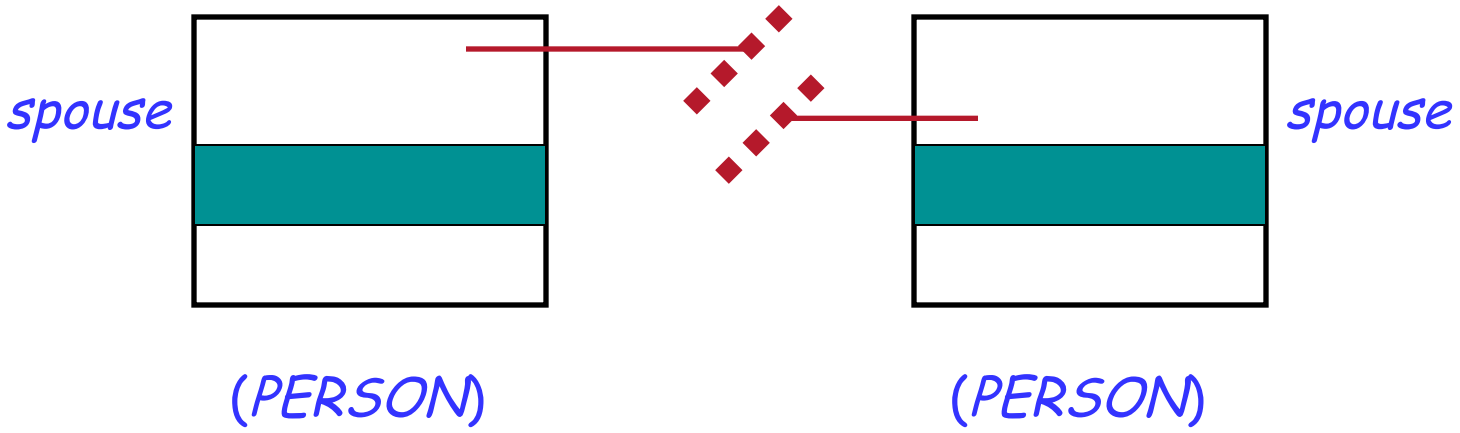
Der Gebrauch von Void-Referenzen

Jedes *PERSON*-Objekt wird mit einer Void-Referenz *spouse* erzeugt.



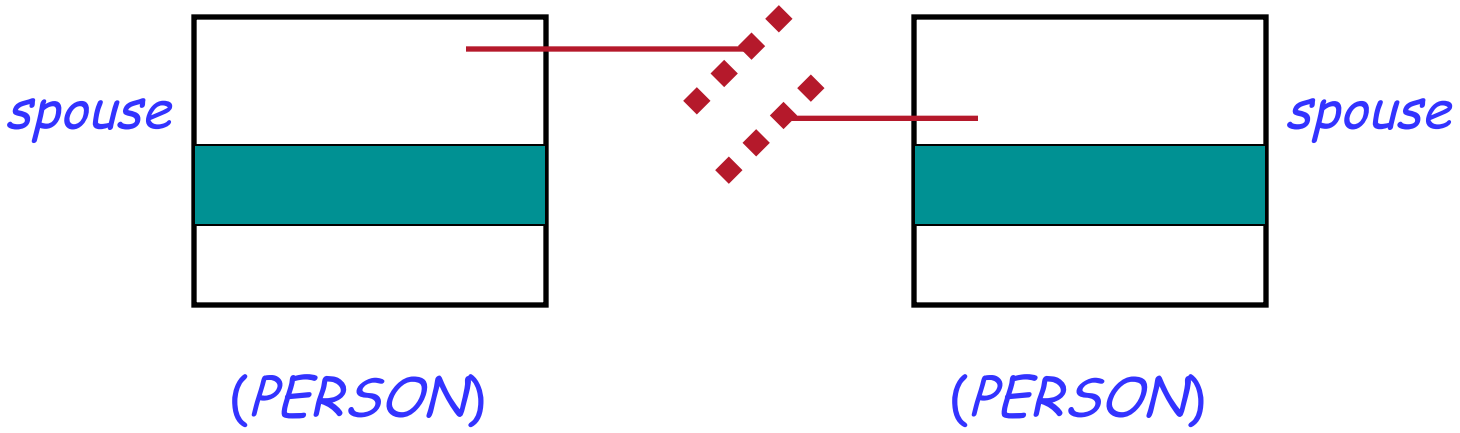
Der Gebrauch von Void-Referenzen

Jedes *PERSON*-Objekt wird mit einer Void-Referenz *spouse* erzeugt.



Der Gebrauch von Void-Referenzen

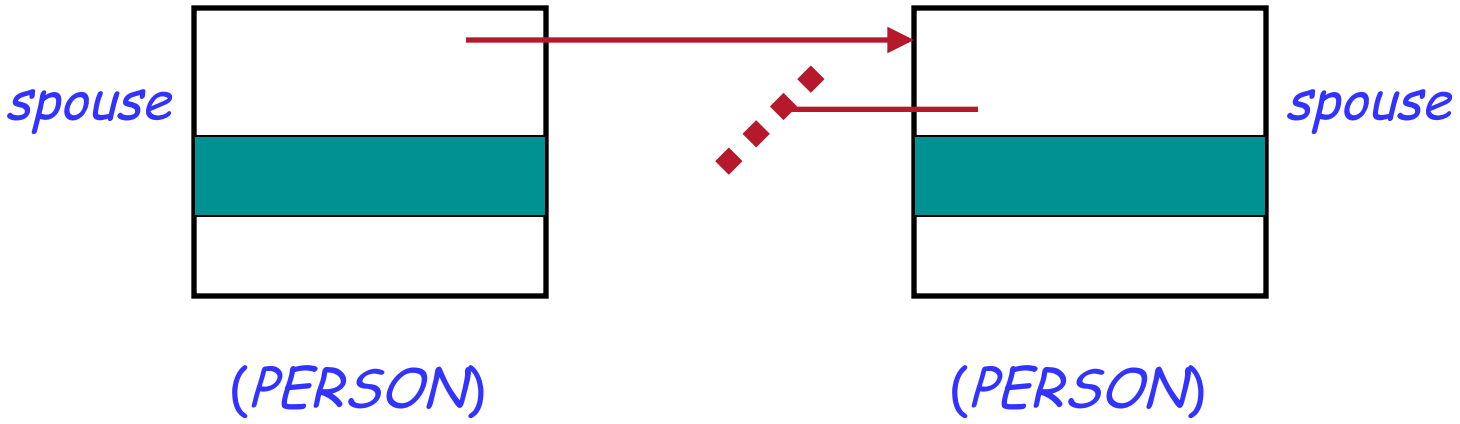
Jedes *PERSON* -Objekt wird mit einer Void-Referenz *spouse* erzeugt...



... und danach werden die *spouse* - Referenzen durch entsprechende Instruktionen gebunden

Der Gebrauch von Void-Referenzen

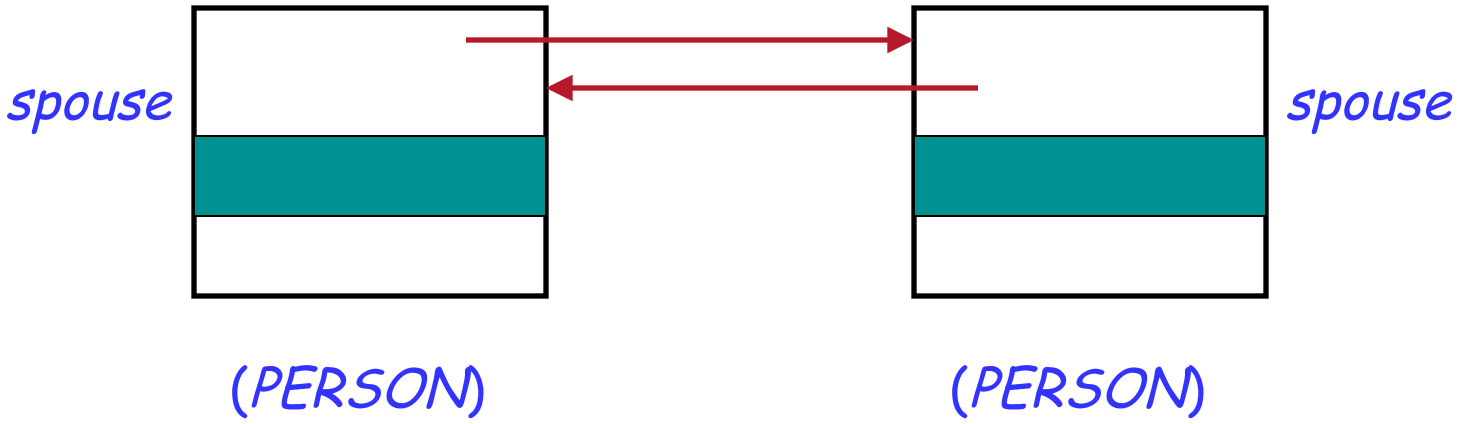
Jedes *PERSON* -Objekt wird mit einer Void-Referenz *spouse* erzeugt...



... und danach werden die *spouse* - Referenzen durch entsprechende Instruktionen gebunden

Der Gebrauch von Void-Referenzen

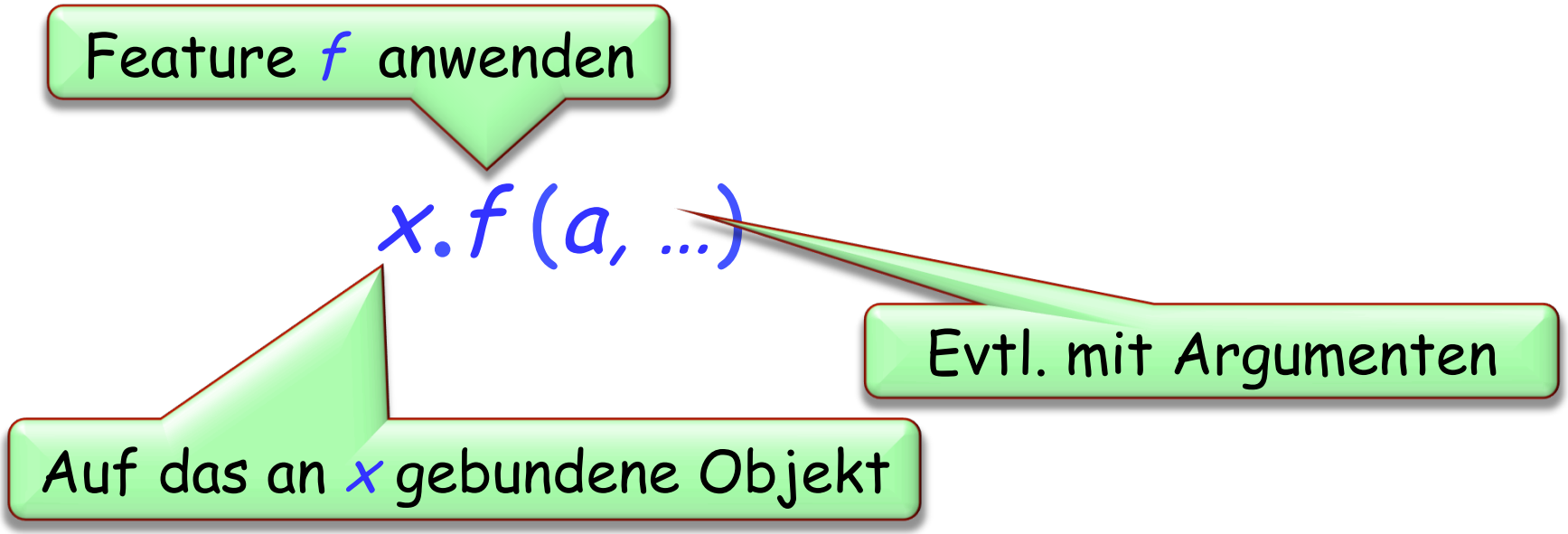
Jedes *PERSON* -Objekt wird mit einer Void-Referenz *spouse* erzeugt...



... und danach werden die *spouse* - Referenzen durch entsprechende Instruktionen gebunden

(Nochmal) das Problem mit Void-Referenzen

Der Grundmechanismus von (O-O) Programmen ist der Featureaufruf



Da Referenzen void sein können, kann *x* möglicherweise an kein Objekt gebunden sein.

In solchen Fällen ist der Aufruf fehlerhaft!

Void-Aufrufe – Die ganze Geschichte

In ISO Eiffel werden Void-Aufrufe dank dem Begriff des „gebundenen Typs" (attached type) nicht mehr vorkommen.

Der Compiler lehnt jeden Aufruf *x.f* ab, bei dem *x* in einer Ausführung void sein könnte.

Dies ist ein grosser Fortschritt, der allerdings auch Kompatibilitätsprobleme für bereits existierenden Code mit sich führt. Deshalb wird dies schrittweise ab EiffelStudio 6.2 eingeführt.

Andere Sprachen kennen dies nicht, aber Spec#, eine auf C# basierende Sprache aus der Forschungsabteilung von Microsoft, spielte eine Vorreiterrolle mit ihren „non-null types".

In diesem Kurs benutzen wir immer noch die alten Regeln.

Void safety – um mehr zu wissen

B. Meyer, E. Stapf, A. Kogtenkov: *Avoid a Void: The Eradication of Null Dereferencing,* in *Reflections on the Work of C.A.R. Hoare*, 2010, auf

docs.eiffel.com/book/papers/void-safety-how-eiffel-removes-null-pointer-dereferencing

oder

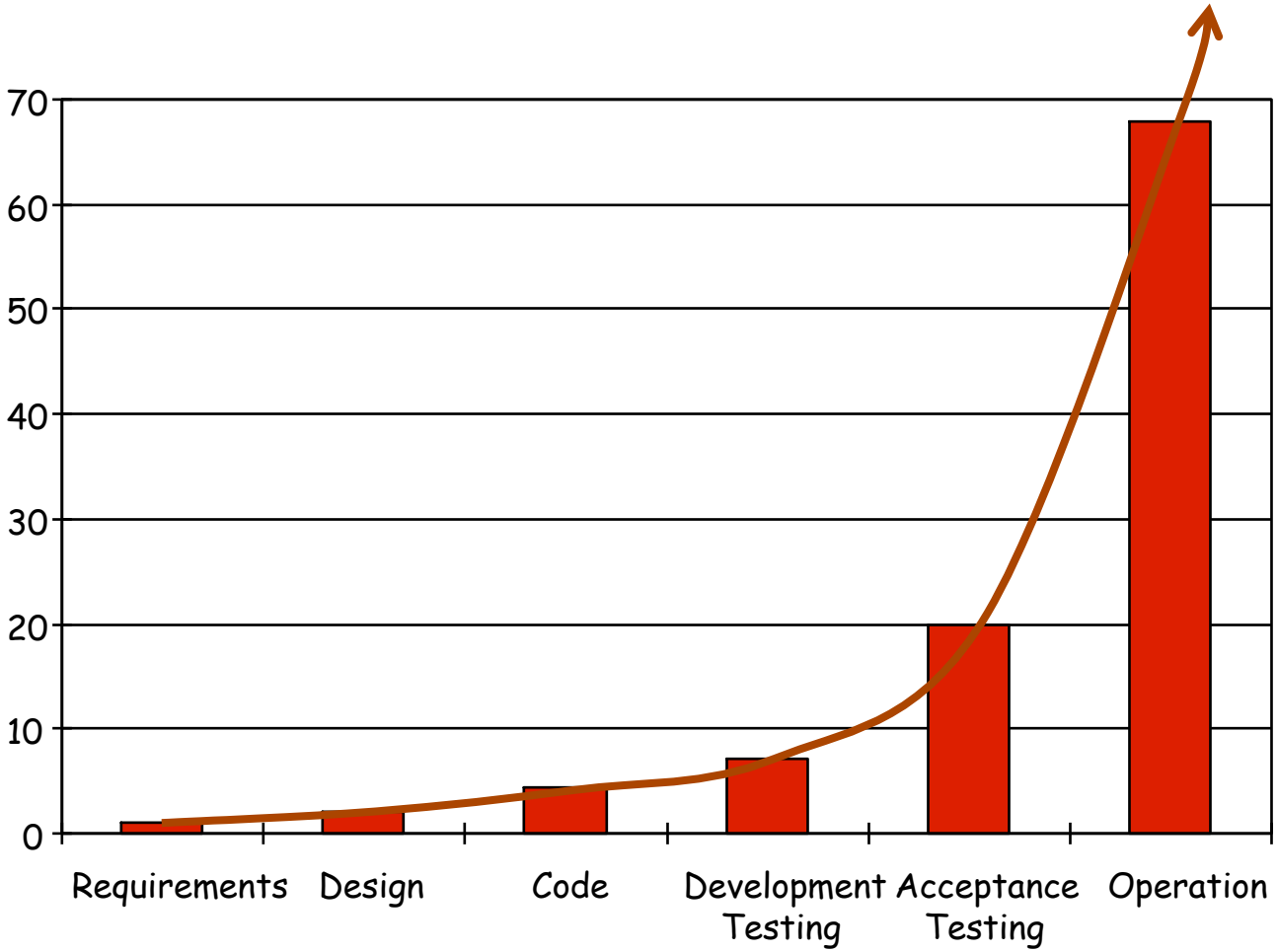
http://bit.ly/8wxXLO

verfügbar

Es lohnt sich, Fehler statisch zu entdecken

Source: Boehm 81

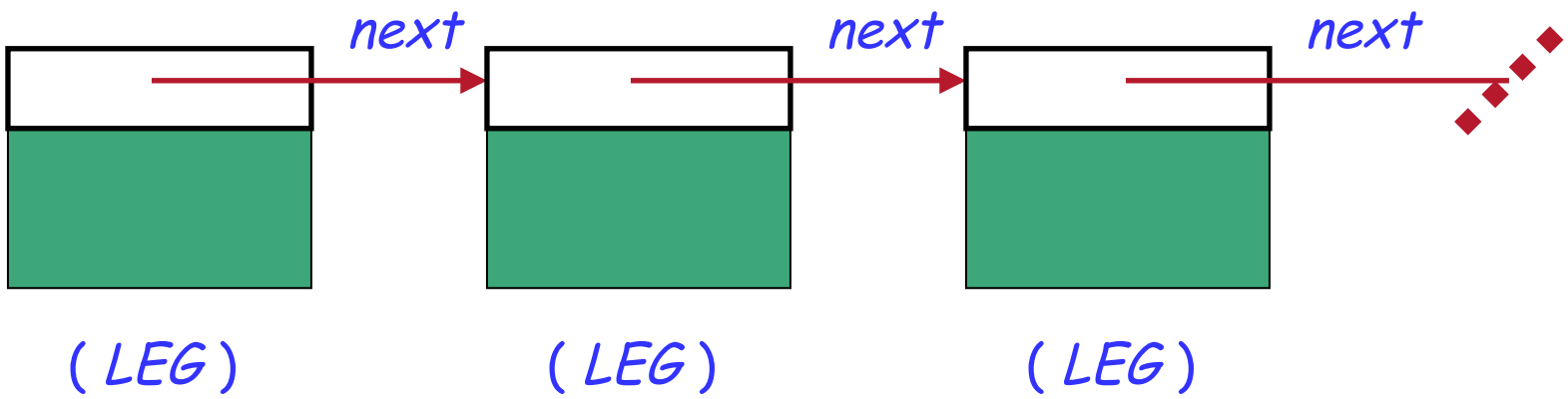
Relativer Kosten, einen Fehler zu beheben



(Nochmals) Die Schnittstelle der Klasse *LEG*

```
class LEG feature
        origin, destination: STATION
                -- Anfang und Ende der Teilstrecke.

        line: LINE
                -- Die Linie, der die Strecke angehört.

        next : LEG
                -- Nächste Teilstrecke in der Route.

        make (o, d : STATION; l: LINE)
                -- Anfang, Ende und Linie setzen.
            require
                linie_existiert: l /= Void
                auf_der_linie: l.has_station (o) and l.has_station (d)
            ensure
                stationen_gesetzt: origin  = o and destination = d
                linie_gesetzt: line = l

        link (other : LEG)
                -- other zur nächsten Teilstrecke der Route machen.
            ensure
                naechste_gesetzt: next = other
end
```

Referenzen in verketteten Strukturen



Die Liste wird durch eine *next* -Referenz, die void ist, beendet.

build_route

```
build_route
        -- Eine Route bauen und sie auf der Karte hervorheben.
    do
        -- Erzeuge die Teilstrecken und setze die Stationen und die Linien:
        create leg1

        leg1.make (Polyterrasse, Central, Polybahn)
        create leg2

        leg2.make (Central, Opernhaus, Line4)
        -- Verbinde die erste Teilstrecke mit der zweiten.

        leg1.link (leg2)
        -- "Opera_route erzeugen und ihr die eben erstellten Teilstrecken zuweisen"

        Zurich.add_route  (Opera_route)
        Opera_route.reverse
    end
```

Vordefiniert in *ZURICH_OBJECTS*

Immer noch Pseudocode!

Die Schnittstelle der Klasse *LEG* (erste Version)

```
class LEG feature
        origin, destination : STATION
                -- Anfang und Ende der Teilstrecke.

        line : LINE
                -- Die Linie, der die Strecke angehört.

        next : LEG
                -- Nächste Teilstrecke in der Route.

        make (o, d : STATION ; l : LINE)
                -- Anfang, Ende und Linie setzen.
            require
                linie_existiert: l /= Void
                auf_der_linie: l.has_station (o) and l.has_station (d)
            ensure
                stationen_gesetzt: origin  = o and destination = d
                linie_gesetzt: line = l

        link (other : LEG)
                -- other zur nächsten Teilstrecke der Route machen.
            ensure
                naechste_gesetzt: next = other
end
```

Der Invariante der Klasse *LEG*

linie_existiert: *line* /= **Void**

anfang_auf_der _linie : *line.has_station* (*origin*)

ende_auf_der_linie : *line.has_station* (*destination*)

Die Notwendigkeit von Erzeugungsprozeduren

Erzeugung und Initialisierung eines *LEG* Objektes:

Nach der Erzeugung:
Invariante nicht erfüllt!

```
create leg
leg.make (station1, station2, line)
```

Die Invariante der Klasse:

```
invariant
        linie_existiert: line /= Void
        anfang_auf_der_linie: line.has_station (origin)
        ende_auf_der_linie: line.has_station (destination)
```

Erzeugungsprozeduren

Eine bessere Lösung:

- Deklarieren Sie *make* als **Erzeugungsprozedur**, um Initialisierung mit Erzeugung zu verbinden:

```
create leg1.make (Polyterrasse, Central, Polybahn)
        -- Gleicher Effekt wie die zwei vorherigen Instruktionen
```

- Einfachheit: Initialisierung bei Erzeugung.
- Korrektheit: Die Invariante wird von Anfang an erfüllt.

Erzeugungsprozeduren heissen auch **Konstruktoren**
(z.B. in Java oder C#)

Die Schnittstelle der Klasse *LEG*

```
class LEG create
        make
feature
        origin, destination: STATION
                        -- Anfang und Ende der Teilstrecke.
        line: LINE
                        -- Linie, der die Strecke angehört.
        next : LEG
                        -- Nächste Teilstrecke der Route.

        make (o, d : STATION; l: LINE)
                        -- Anfang, Ende und Linie setzen.
                require
                        linie_existiert: l /= Void
                        auf_der_linie: l.has_station (o) and l.has_station (d)
                ensure
                        stationen_gesetzt: origin  = o and destination = d
                        linie_gesetzt: line = l

        link (other : LEG)
                        -- other zur nächsten Teilstrecke der Route machen.
                ensure
                        naechste_gesetzt: next = n
invariant
        linie_existiert: line /= Void
        auf_der_linie: line.has_station (origin) and line.has_station (destination)
end
```

Auflistung der Erzeugungsprozeduren

Jetzt **auch** als Erzeugungsprozedur verfügbar

Erzeugungsprinzip

Falls eine Klasse eine nicht-triviale Invariante hat, muss sie eine oder mehrere Erzeugungsprozeduren definieren, die sicherstellen, dass jede Instanz nach der Ausführung einer Erzeugungsinstruktion die Invariante erfüllt.

Dies erlaubt dem Autor der Klasse, zu erzwingen, dass alle von Klienten erzeugten Instanzen korrekt initialisiert werden.

Erzeugungsprozeduren

Auch wenn keine starke Invariante vorhanden ist, sind Erzeugungsprozeduren nützlich, um Initialisierung und Erzeugung zu kombinieren.

```
class POINT create
        default_create, make_cartesian, make_polar
feature
        ...
end
```

Vererbt an alle Klassen, macht standardmässig nichts

Gültige Erzeugungsinstruktionen:

```
create your_point.default_create

create your_point
create your_point.make_cartesian (x, y)
create your_point.make_polar (r, t)
```

Objekterzeugung: Zusammenfassung

Um ein Objekt zu erzeugen:

- Falls die Klasse keine **create**-Klausel hat, benutzen Sie die Grundform: **create** *x*
- Falls die Klasse eine **create**-Klausel hat, die eine oder mehrere Prozeduren auflistet, benutzen Sie

  create *x.make* (…)

  wobei *make* eine der Erzeugungsprozeduren ist und "(…)" für allfällige Argumente steht.

Korrektheit einer Instruktion

Um mit dem Prinzip „Design by Contract" („Entwurf gemäss Vertrag") übereinzustimmen, müssen wir von jeder Instruktion genau wissen:

- Wie man die Instruktion richtig benutzt: die Vorbedingung
- Was wir dafür garantiert bekommen: die Nachbedingung

Zusammen definieren diese beiden Eigenschaften (zusammen mit der Invarianten) die **Korrektheit** eines Sprachmechanismus

Wie lautet die Korrektheitsregel für eine Erzeugungsinstruktion?

Korrektheit einer Erzeugungsinstruktion

Korrektheitsregel für Erzeugungsinstruktionen

Vor der Erzeugungsinstruktion:

1. Die Vorbedingung der Erzeugungsprozedur (falls vorhanden) muss erfüllt sein

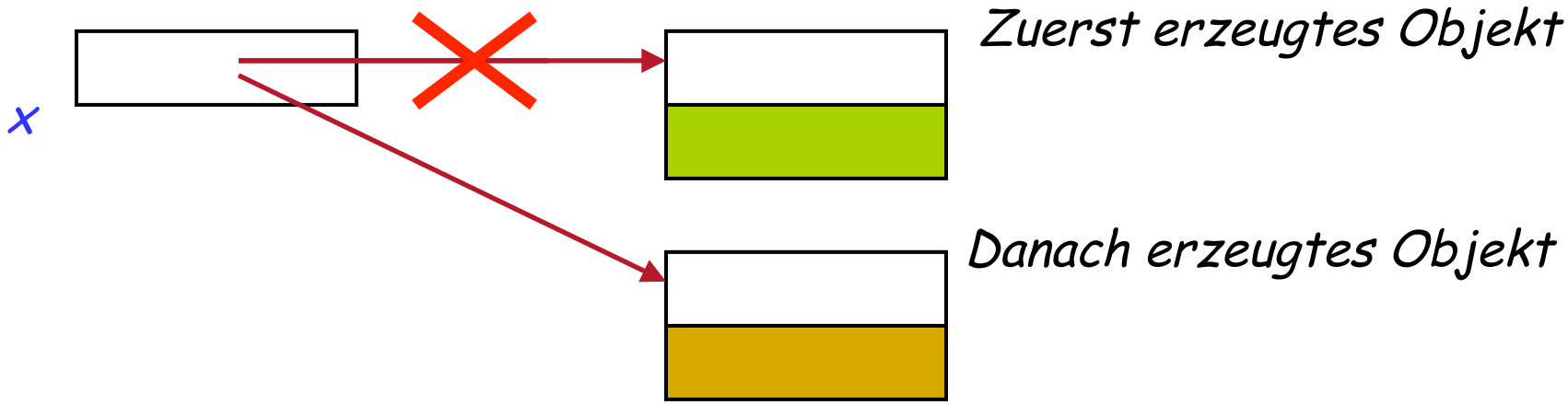
Nach der Erzeugungsinstruktion mit dem Ziel *x* vom Typ *C*:

2. *x* /= **Void** gilt
3. Die Nachbedingung der Erzeugungsprozedur ist erfüllt
4. Das an *x* gebundene Objekt erfüllt die Invariante von *C*

Aufeinanderfolgende Erzeugungsinstruktionen

Die Korrektheitsregel erfordert nicht, dass *x* void ist:

```
create x
        -- Hier ist x nicht void.
...
create x
```



Der Effekt einer Erzeugungsinstruktion

- *x* ist nach der Erzeugungsinstruktion nicht void (egal ob es vorher void war oder nicht)
- Das eben erzeugte Objekt ist an *x* gebunden
- Falls es eine Erzeugungsprozedur gibt, ist ihre Nachbedingung für das neue Objekt erfüllt
- Das Objekt erfüllt die Klasseninvariante

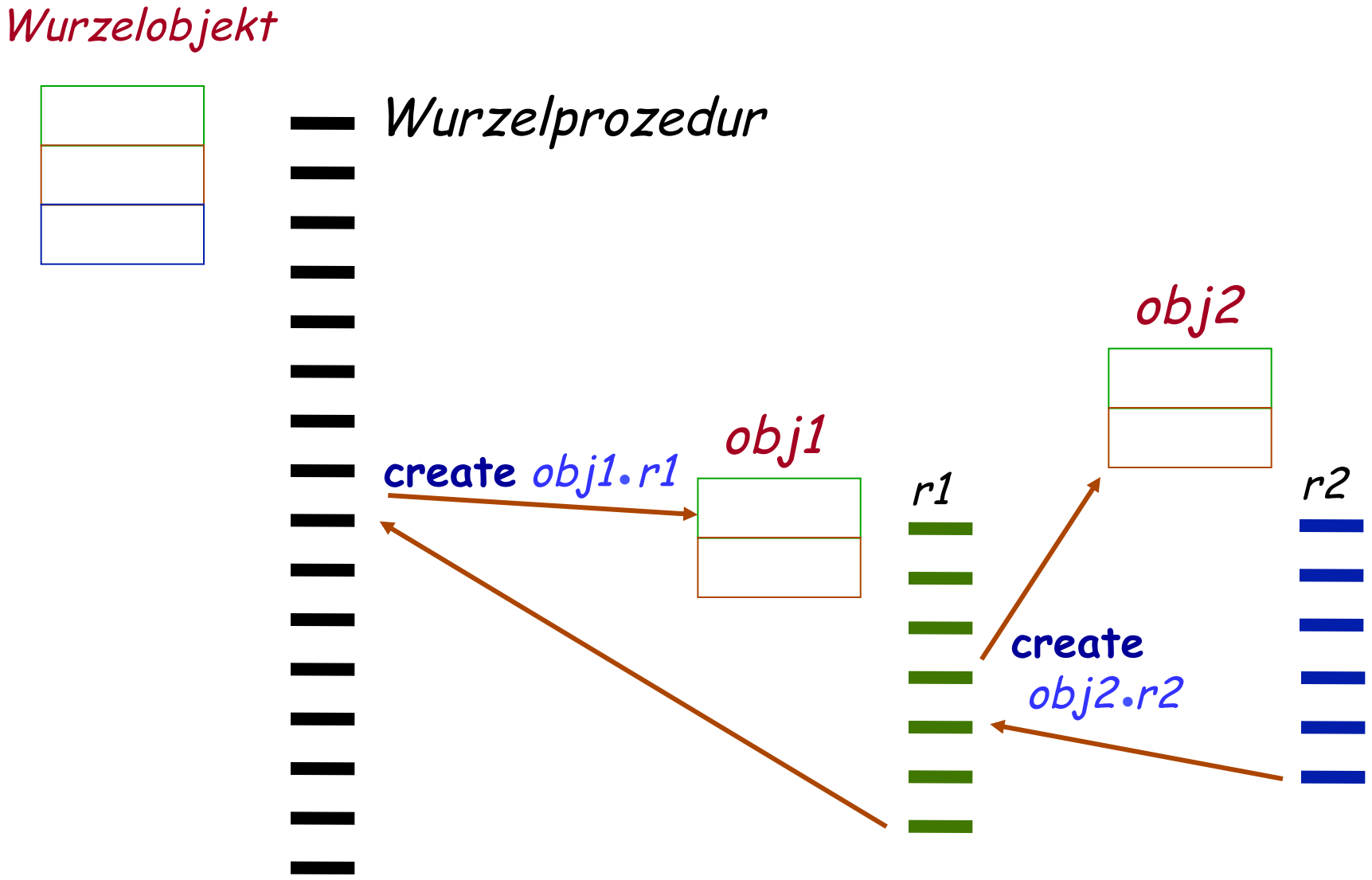
Wie alles beginnt…

Die Ausführung eines Systems beginnt mit der Erzeugung eines **Wurzelobjektes**, das eine Instanz einer vom System vorgesehenen Klasse (die **Wurzelklasse**) ist, mittels einer definierten Erzeugungsprozedur dieser Klasse (die **Wurzelerzeugungprozedur** oder einfach **Wurzelprozedur**)

Eine Wurzelprozedur kann:

- Neue Objekte erzeugen
- Auf diesen Features aufrufen, die wiederum neue Objekte erzeugen können
- Etc.

Ausführung eines Systems



Das aktuelle Objekt (current object)

Zu jeder Zeit während einer Ausführung gibt es ein aktuelles Objekt, auf welchem das aktuelle Feature aufgerufen wird.

Zu Beginn ist dies das Wurzelobjekt

Während eines „qualifizierten" Aufrufs $x.f(a)$, ist das neue aktuelle Objekt dasjenige, das an x gebunden ist..

Nach einem solchen Aufruf übernimmt das vorherige aktuelle Objekt wieder diese Rolle.

Der Entwurfsprozess

Ein System ist eine bestimmte Gruppe von gewissen Klassen, wobei eine Klasse als Wurzelklasse dient.

Die Klassen können auch ausserhalb des Systems wertvoll sein: sie sind dann wiederverwendbar.

Erweiterbarkeit & Wiederverwendbarkeit

- **Erweiterbarkeit**: Die Einfachheit, mit welcher es möglich ist, ein System den Wünschen eines Benutzers entsprechend anzupassen.

- **Wiederverwendbarkeit**: Die Einfachheit, bestehende Software für neue Applikationen wiederzuverwenden.

Ältere Software-Engineering-Ansätze, die auf einem Hauptprogramm und Unterprogrammen beruhen, beachten diese Bedürfnisse weniger.

Die Wurzel (root) ausfindig machen

Wie spezifiziert man die **Wurzelklasse** und die **Wurzel-Erzeugungsprozedur** eines Systems?

Benutzen Sie EiffelStudio

Was wir in dieser Vorlesung gesehen haben

- Klasseninvarianten
- Das Konzept "Design by Contract"
- Den Begriff einer Ausnahme
- Objekterzeugung
- Erzeugungsprozeduren
- Die Beziehung zwischen Erzeugungsprozeduren und Invarianten
- Den Vorgang einer Systemausführung