ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 6: Objekterzeugung

Objekte erzeugen

In früheren Beispielen haben *Paris, Line8* etc. jeweils vordefinierte Objekte bezeichnet. Wir werden jetzt unsere eigenen Objekte erzeugen.

Unser Ziel: Eine fiktive Metrolinie, *fancy_line* :

Beispiel: *LINE_BUILDING*

```
class LINE_BUILDING inherit
        TOURISM
 feature
        build_a_line
                        -- Eine imaginäre Linie bauen und sie auf der Karte hervorheben.
            do
                        Paris.display
                        -- " fancy_line  erzeugen und Stationen hinzufügen"
                        Paris.put_line (fancy_line)
                        fancy_line.highlight
            end

        fancy_line: LINE
                        -- Eine imaginäre Linie der Metro
end
```

Pseudocode

Bezeichnet eine Instanz der Klasse *LINE*

Bezeichner, Entitäten (entities), Variablen

Ein Bezeichner (identifier) ist ein vom Programmierer / von der Programmiererin gewählter Name, um ein gewisses Programmelement zu repräsentieren.

Beispiele für Bezeichner :

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

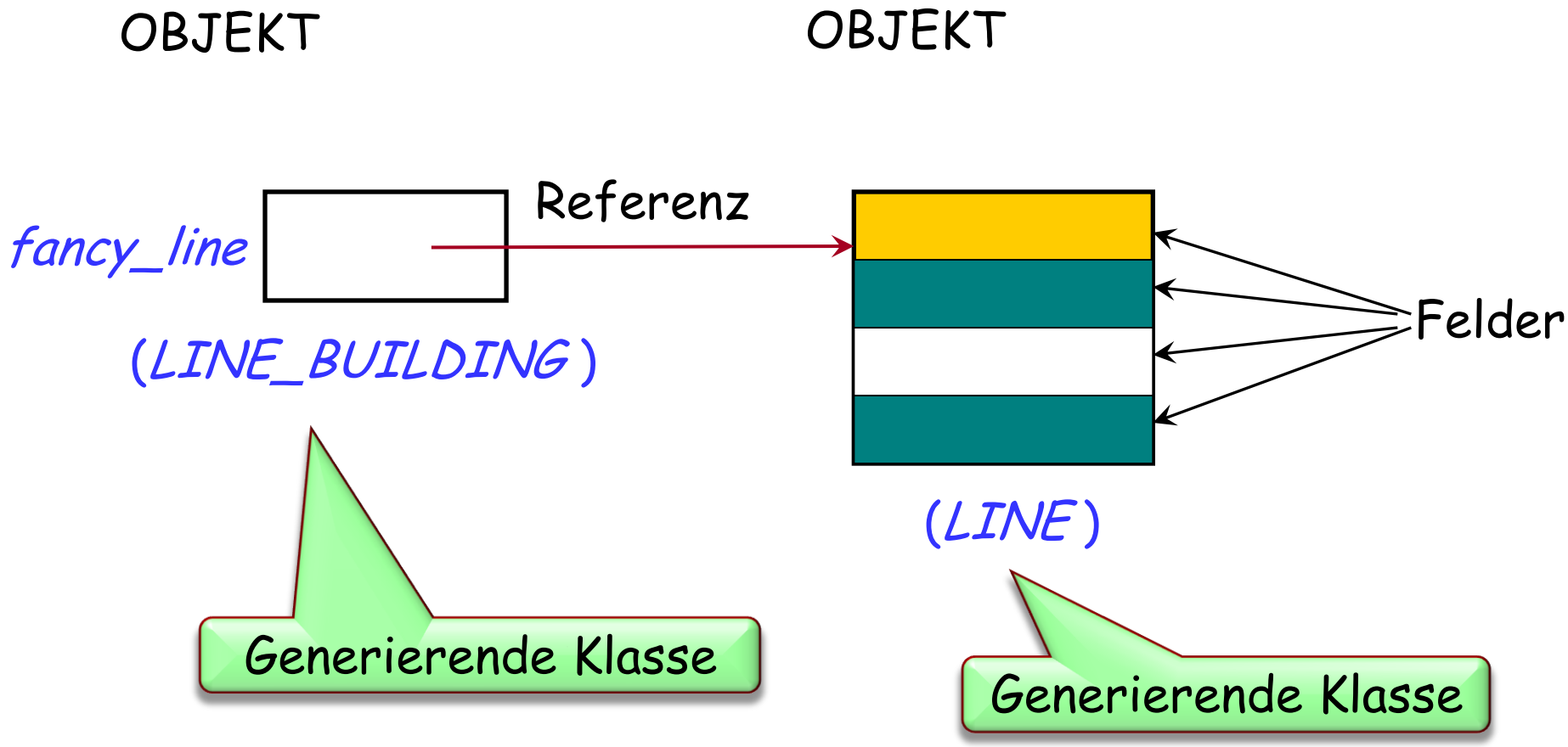
Ein Bezeichner, der einen Laufzeitwert bezeichnet, wird Entität oder Variable (falls sein Wert sich ändern kann) genannt.

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

An ein Objekt gebundene Entitäten

Im Programm: Eine Entität, wie z.B. *fancy_line*

Im Speicher, während der Ausführung: Ein Objekt

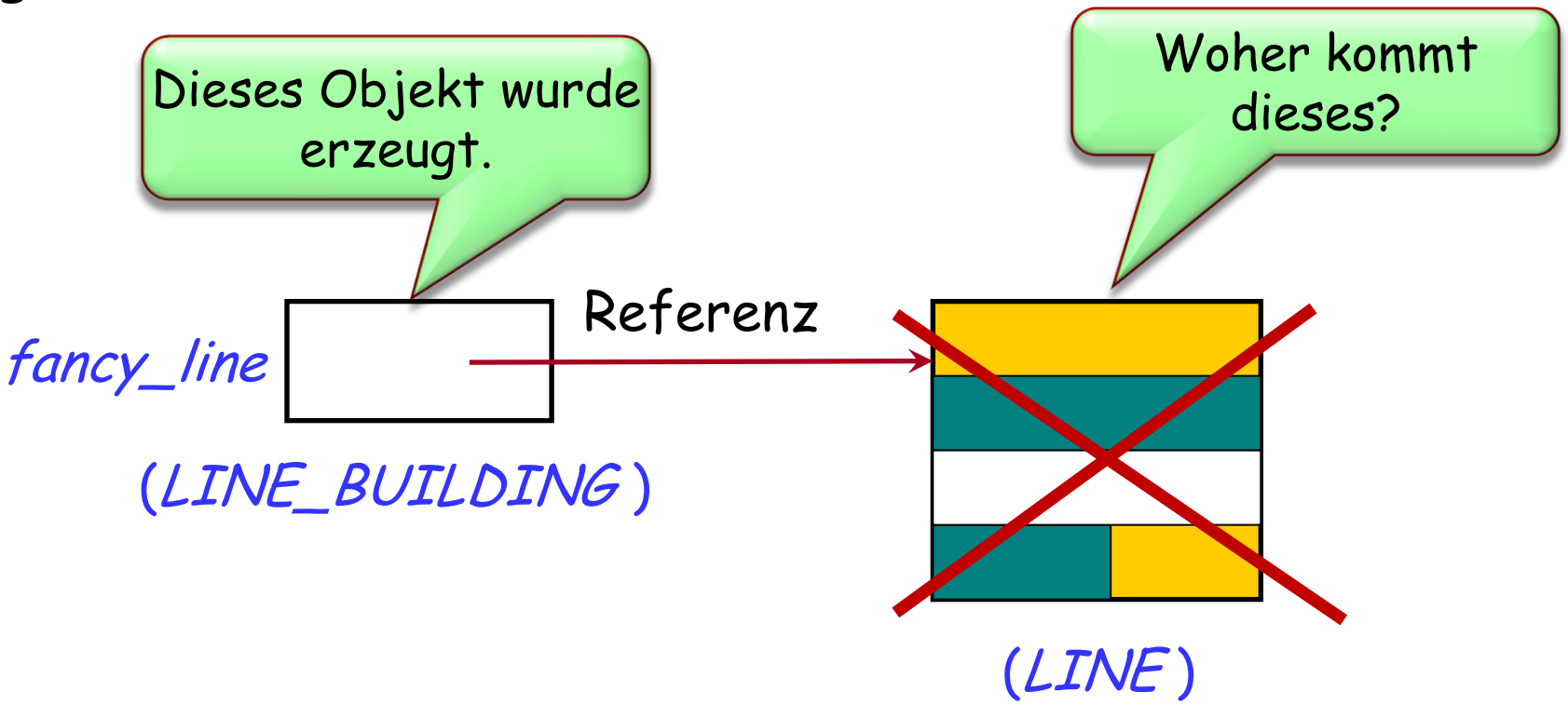


LINE_BUILDING

```
class LINE_BUILDING inherit
        TOURISM
 feature
        build_a_line
                        -- Eine imaginäre Linie bauen und sie auf der Karte hervorheben.
            do
                        Paris.display
                        -- "fancy_line  erzeugen und Stationen hinzufügen"
                        Paris.put_line (fancy_line)
                        fancy_line.highlight
            end

        fancy_line: LINE
                        -- Eine imaginäre Linie der Metro
end
```

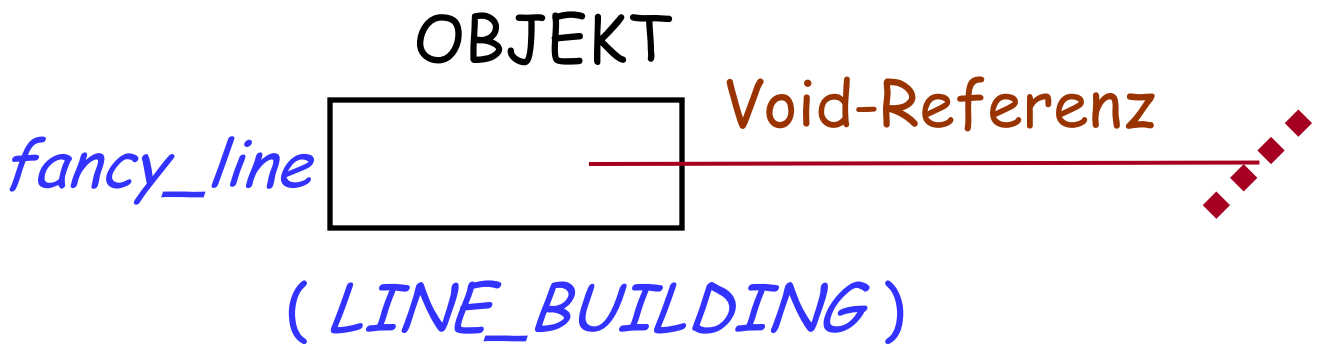
Der Grundzustand einer Referenz

Können wir in einer Instanz von *LINE_BUILDING* annehmen, dass *fancy_line* an eine Instanz von *LINE* gebunden ist?



Standard-Referenzen

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



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:

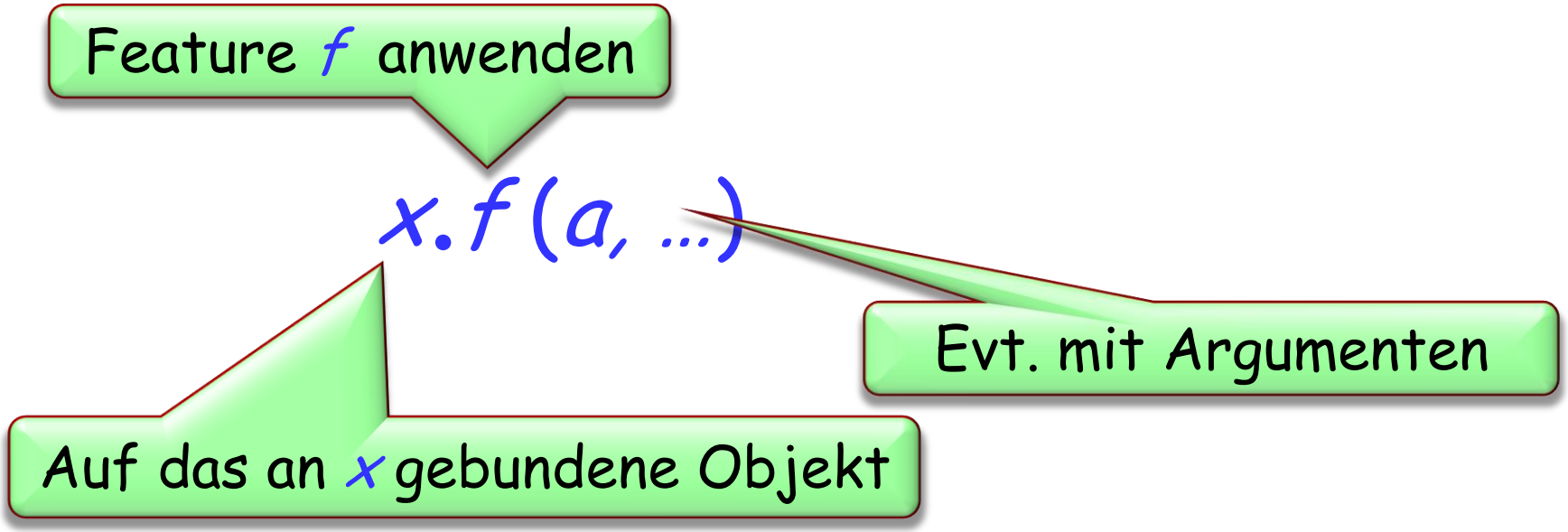
  x = **Void**

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

  x /= **Void**

Das Problem mit Void-Referenzen

Der Grundmechanismus von Programmen ist der Featureaufruf.



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

In diesen Fällen ist der Aufruf fehlerhaft!

Beispiel: Aufruf auf ein Ziel, das void ist

```
class LINE_BUILDING inherit
        TOURISM
 feature
        build_a_line
                        -- Eine imaginäre Linie bauen und sie auf der Karte hervorheben.
            do
                        Paris.display
                        -- "fancy_line erzeugen und Stationen hinzufügen"
                        Paris.put_line (fancy_line)
                        fancy_line.highlight
            end

        fancy_line: LINE
                        -- Eine imaginäre Linie der Metro
end
```

Void-Referenz

Ausnahmen (exceptions)

Abnormale Ereignisse während der Ausführung. Beispiele:

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

Ein Fehler ist die Konsequenz; 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

Objekte explizit erzeugen

Um eine Ausnahme zu vermeiden:

- Verändern Sie die Prozedur *build_a_line*, so dass sie ein Objekt erzeugt und an *fancy_line* bindet, bevor es *highlight* aufruft.

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*, bei dem *x* in einer Ausführung void sein könnte, ab.

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

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

In diesem Kurs benutzen wir immer noch die alten Regeln.

Warum müssen wir Objekte erzeugen?

Können wir nicht annehmen, dass eine Deklaration der Form

fancy_line : *LINE*

eine Instanz von *LINE* erzeugt und sie an *fancy_line* bindet?

(Die Antwort darauf folgt bald…)

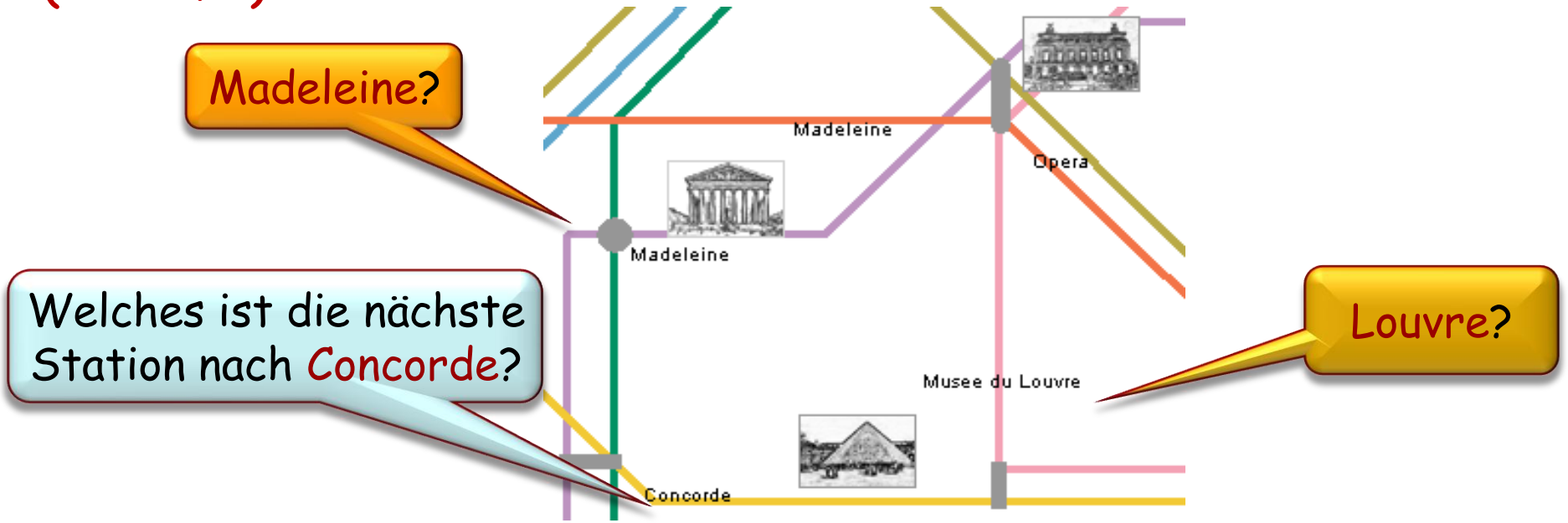
LINE_BUILDING

```
class LINE_BUILDING inherit
    TOURISM
feature
    build_a_line
            -- Eine imaginäre Linie bauen und sie auf der Karte hervorheben.
        do
            Paris.display
            -- "fancy_line erzeugen und Stationen hinzufügen"
            Paris.put_line (fancy_line)
            fancy_line.highlight
        end


    fancy_line: LINE
            -- Eine imaginäre Linie der Metro
```

Einfache Objekte erzeugen

Um *fancy_line* zu erzeugen, müssen wir zuerst Objekte erzeugen, die die Stationen und Haltestellen der Linie repräsentieren.

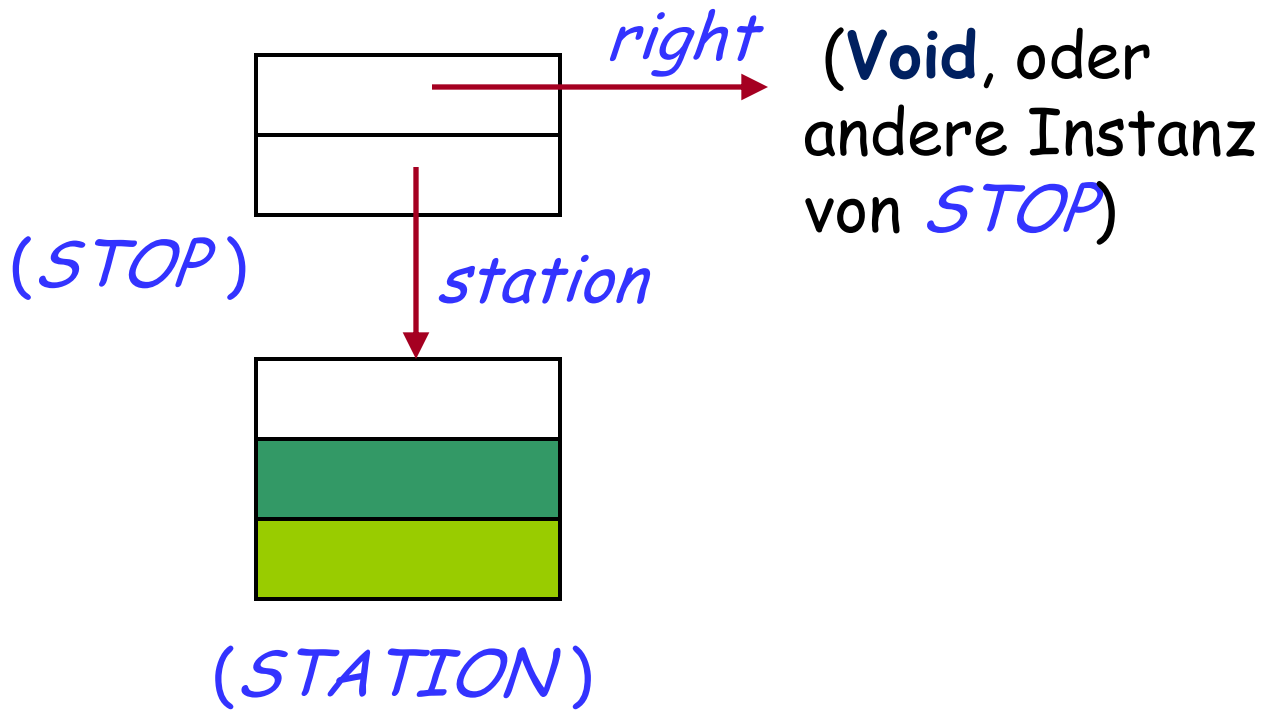
Wir brauchen Instanzen von *STATION* UND *STOP* (warum?)



Haltestellen

Eine Instanz von *STOP* hat:

- Eine Referenz zu einer Station; darf nicht void sein.
- Eine Referenz zur nächsten Haltestelle; void am Schluss.



Die Schnittstelle der Klasse *SIMPLE_STOP*

```
class SIMPLE_STOP feature

        station: STATION
                -- Station, welche diese Haltestelle repräsentiert.

        next: SIMPLE_STOP
                -- Nächste Haltestelle der selben Linie.

        set_station (s: STATION)
                -- Diese Haltestelle mit s assoziieren.
            require
                station_existiert: s /= Void
            ensure
                station_gesetzt: station = s

        link (s: SIMPLE_STOP)
                -- s zur nächsten Haltestelle dieser Linie machen.
            ensure
                naechste_gesetzt: next = s

end
```

LINE_BUILDING

```
class LINE_BUILDING inherit
      TOURISM
 feature

      stop1: SIMPLE_STOP
                -- Erste Haltestelle dieser Linie
      build_a_line
                -- Eine imaginäre Linie bauen und sie auf der Karte hervorheben.
          do
                Paris.display
                -- "fancy_line erzeugen und Stationen hinzufügen"



                Paris.put_line (fancy_line)
                fancy_line.highlight
          end
      fancy_line: LINE
                -- Eine imaginäre Linie der Metro
end
```

Eine Instanz von *SIMPLE_STOP* erzeugen

```
class LINE_BUILDING inherit
    TOURISM
feature
    stop1: SIMPLE_STOP
            -- Erste Haltestelle dieser Linie
    build_a_line
            -- Eine imaginäre Linie bauen und sie auf der Karte hervorheben.
        do
            Paris.display
                    -- Erzeuge fancy_line und fülle sie mit Stationen:
            create stop1
            -- "Mehr Stationen erzeugen und fancy_line fertig bauen"
            Paris.put_line (fancy_line)
            fancy_line.highlight
        end
    fancy_line: LINE
            -- Eine imaginäre Linie der Metro
end
```

Erzeugungs-instruktion

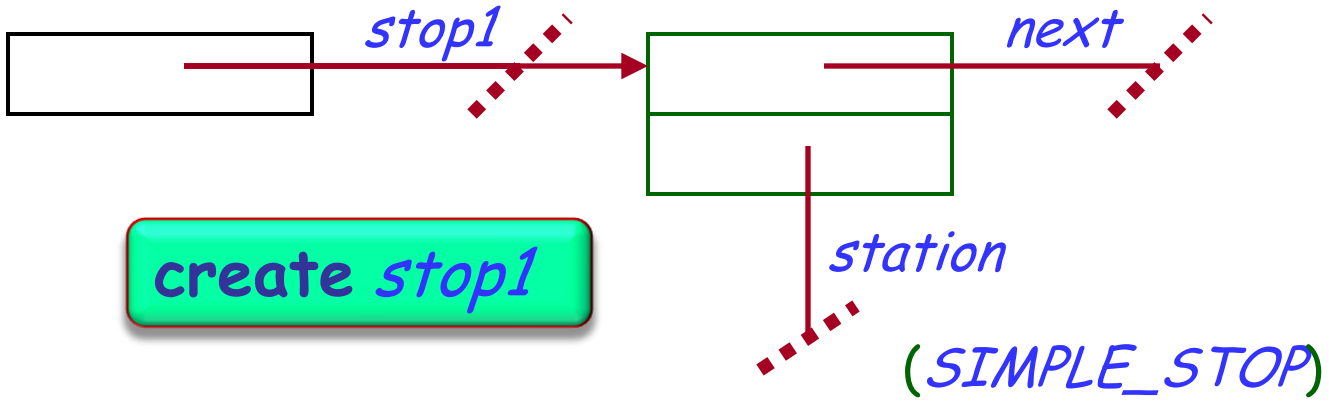
Jetzt ein gewöhnlicher Kommentar

Neuer Pseudocode

Erzeugungsinstruktion

Grundoperation, um Objekte während der Laufzeit zu erzeugen:

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



Der Typ eines erzeugten Objektes

Jede Entität ist mit einem gewissen Typ deklariert:

stop1: *SIMPLE_STOP*

Eine Erzeugungsinstruktion

create *stop1*

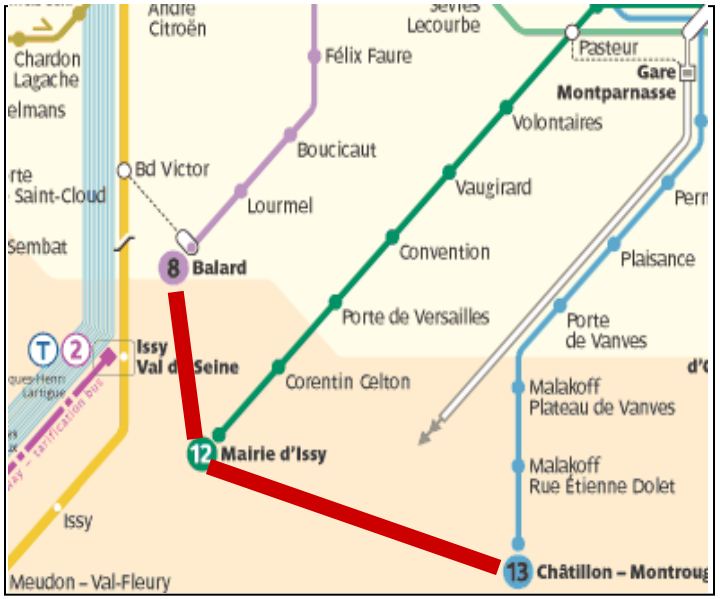
produziert während der Laufzeit ein Objekt dieses Typs.

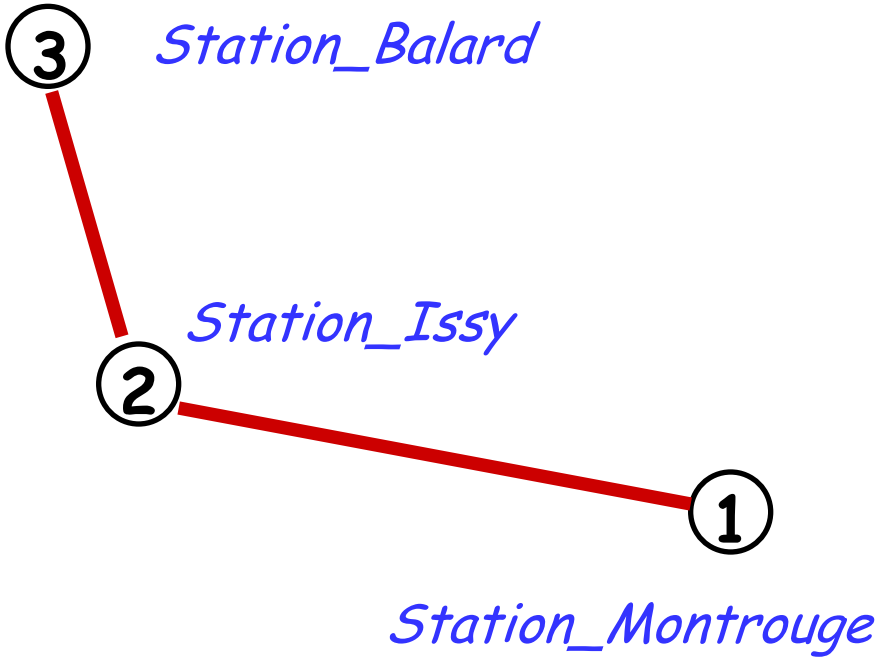
Eine Linie mit drei Haltestellen

Wir möchten jetzt drei Haltestellen hinzufügen.

Als Erstes deklarieren wir die entsprechenden Attribute:

stop1, stop2, stop3 : SIMPLE_STOP





build_a_line

```
build_a_line
        -- Eine imaginäre Linie bauen und sie auf der Karte hervorheben.
    do
        Paris.display
            -- Erzeuge die Stops und weise jedem seine Station zu:
        create stop1
        stop1.set_station (Station_Montrouge)
        create stop2
        stop2.set_station (Station_Issy)
        create stop3
        stop3.set_station (Station_Balard)
            -- Verbinde jeden Stop mit dem nächsten:
        stop1.link (stop2)
        stop2.link (stop3)
            -- "fancy_line erzeugen und ihr die eben erstellten Haltestellen zuweisen."
        Paris.put_line (fancy_line)
        fancy_line.highlight
    end
```

Vordefiniert in *TOURISM*

Immer noch Pseudocode!

Nochmals: Wieso müssen wir Objekte erzeugen?

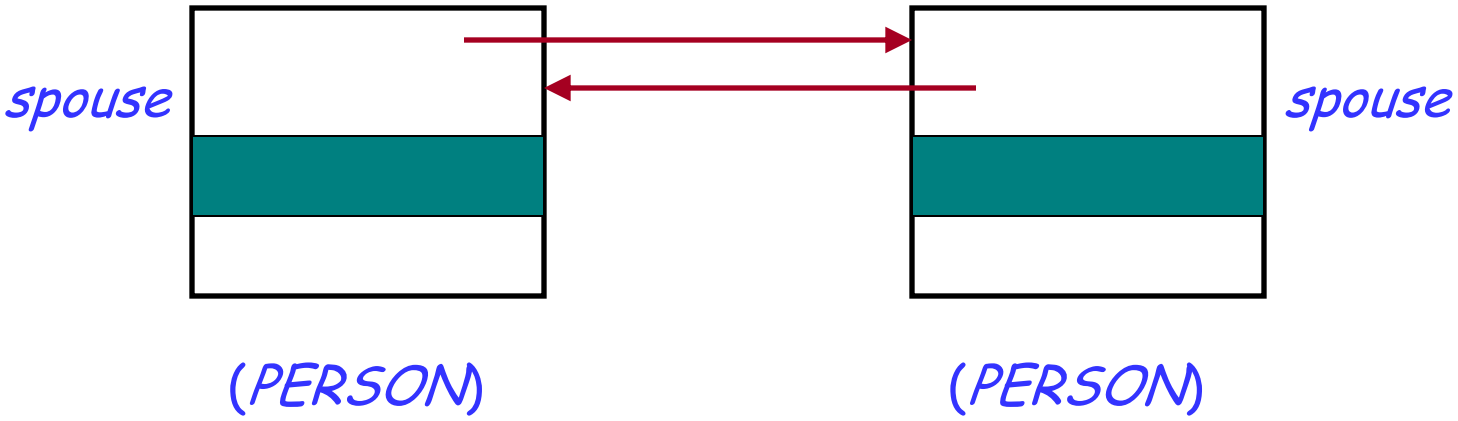
Können wir nicht annehmen, dass eine Deklaration der Form

fancy_line : *LINE*

eine Instanz von *LINE* erzeugt und sie an *fancy_line* bindet?

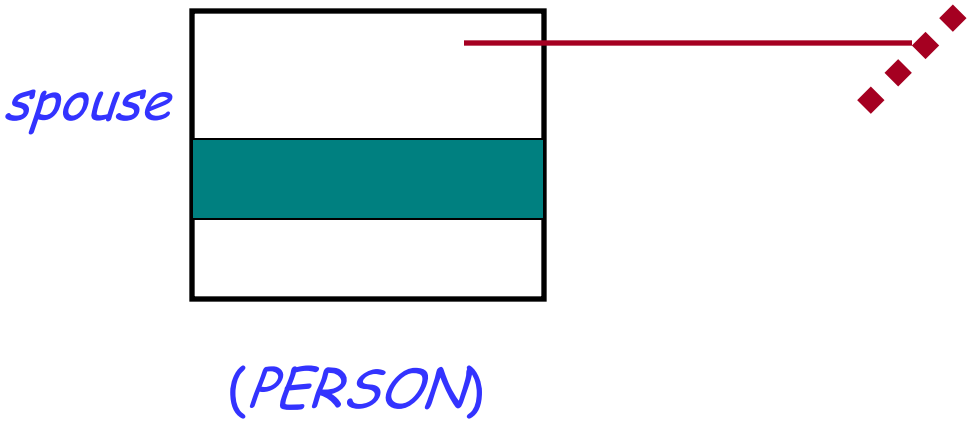
Void-Referenzen sind nützlich!

Verheiratete Personen:



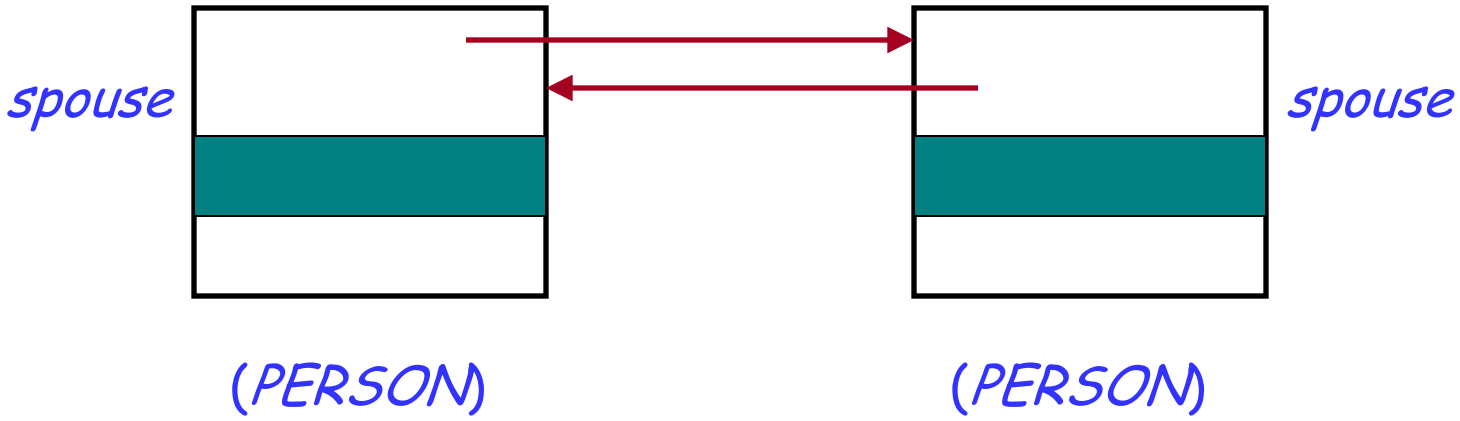
Void-Referenzen sind nützlich!

Ledige Personen:



Void-Referenzen sind nützlich!

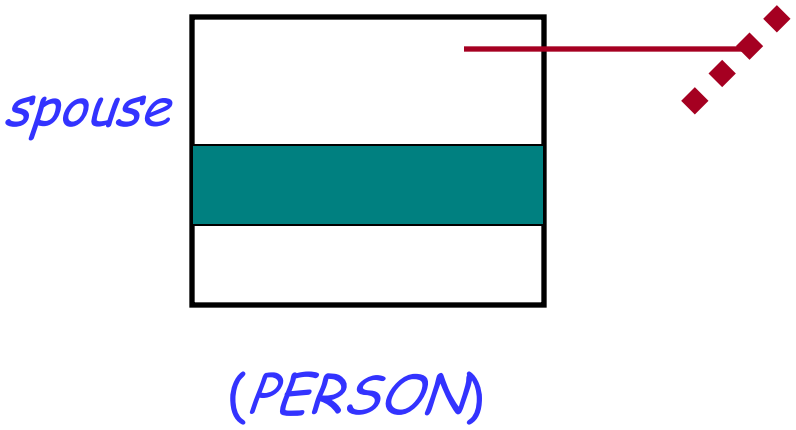
Auch bei verheirateten Personen…



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

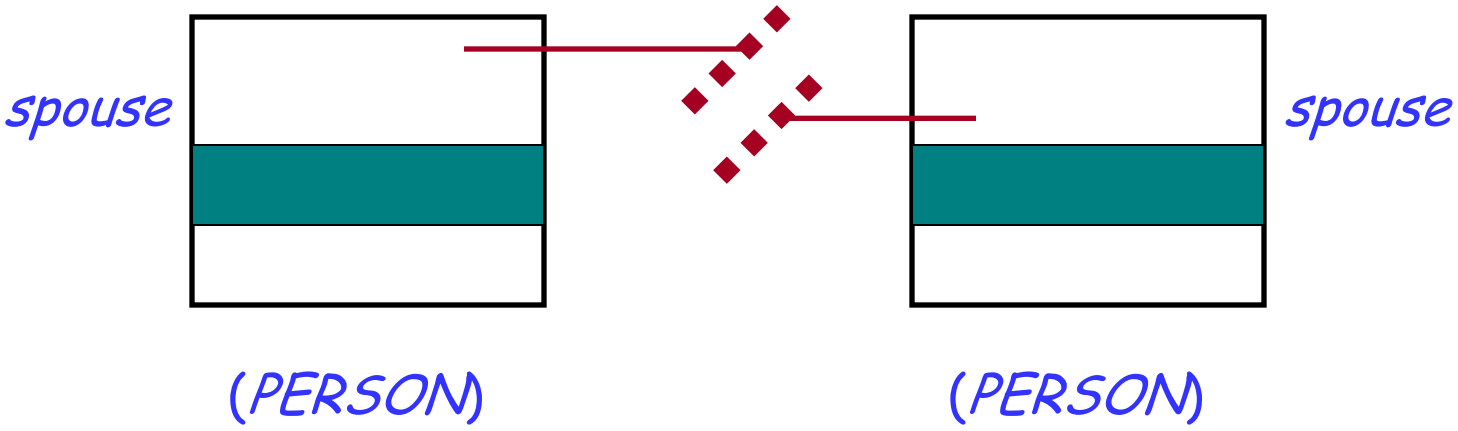
Der Gebrauch von Void-Referenzen

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



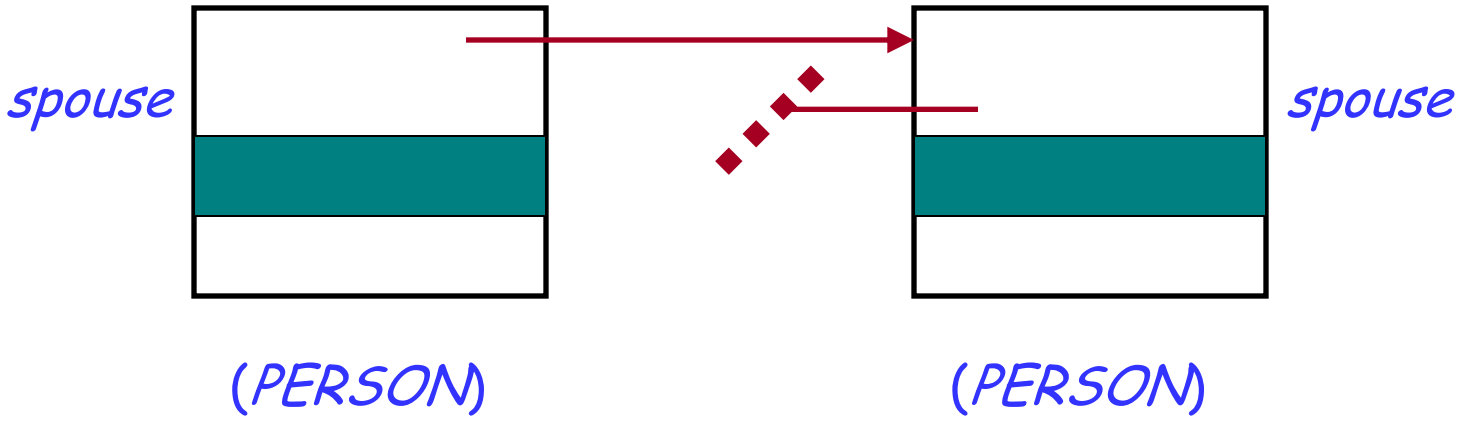
Der Gebrauch von Void-Referenzen

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



Der Gebrauch von Void-Referenzen

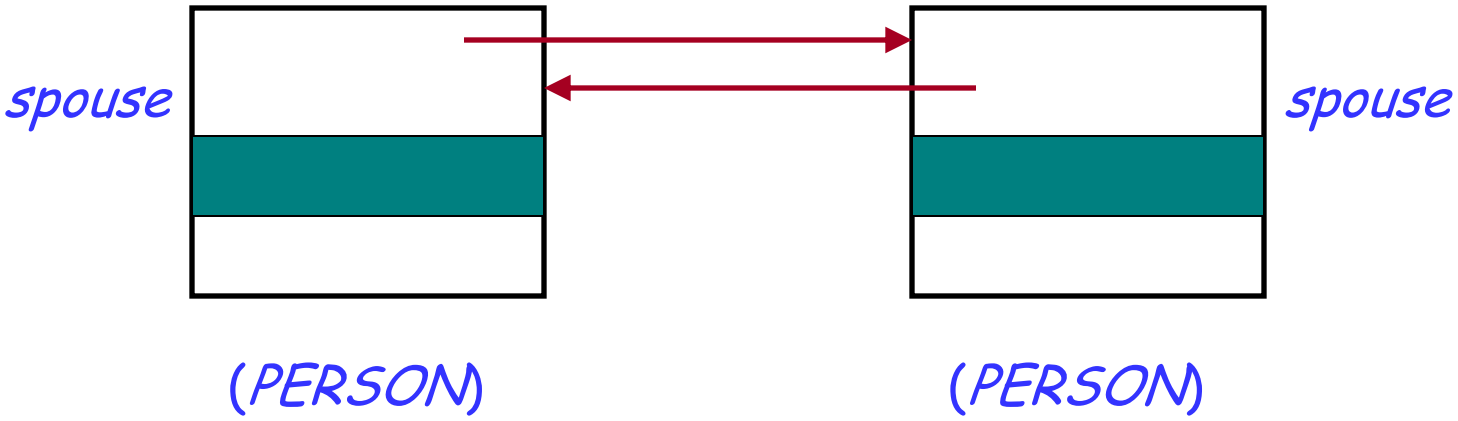
Jedes *PERSON*-Objekt wird mit einer Void-Referenz *spouse* erzeugt...



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

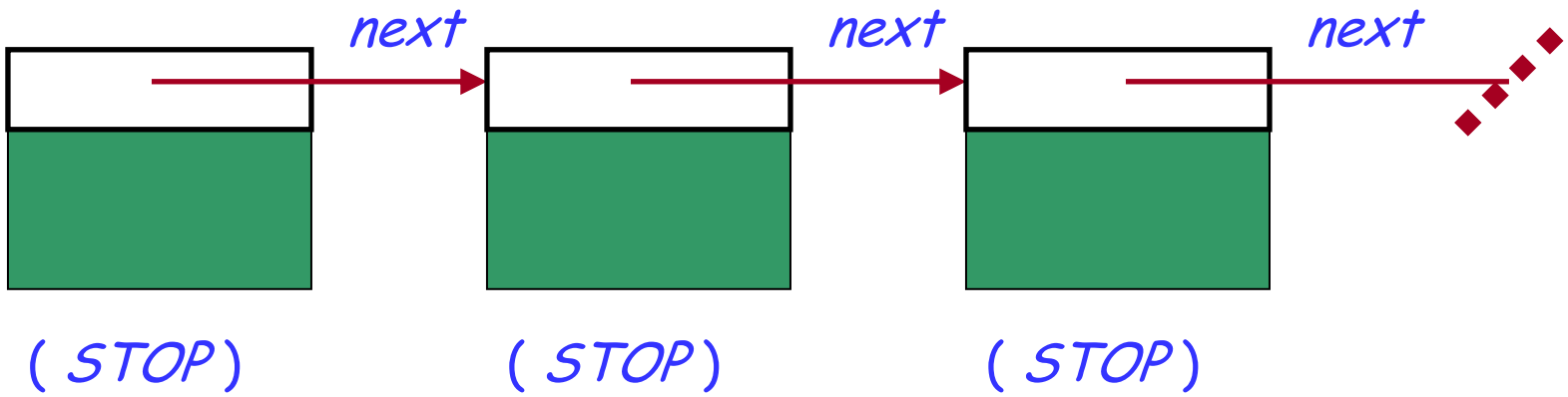
Der Gebrauch von Void-Referenzen

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



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

Referenzen in verketteten Strukturen



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

build_a_line

```
build_a_line
        -- Eine imaginäre Linie bauen und sie auf der Karte hervorheben.
    do
        Paris.display
            -- Erzeuge die Stops und weise jedem seine Station zu:
        create stop1
        stop1.set_station (Station_Montrouge)
        create stop2
        stop2.set_station (Station_Issy)
        create stop3
        stop3.set_station (Station_Balard)
            -- Verbinde jeden Stop mit dem nächsten:
        stop1.link (stop2)
        stop2.link (stop3)
            -- "fancy_line erzeugen und ihr die eben erstellten Haltestellen zuweisen."
        Paris.put_line (fancy_line)
        fancy_line.highlight
    end
```

Vordefiniert in *TOURISM*

Immer noch Pseudocode!

Die Notwendigkeit von Erzeugungsprozeduren

Erzeugung und Initialisierung eines *SIMPLE_STOP* Objektes:

```
create some_stop
some_stop.set_station (existing_station)
```

Nach der Erzeugung: Invariante nicht erfüllt!

Die Invariante der Klasse:

```
invariant
        station_existiert: station /= Void
```

Erzeugungsprozeduren

Eine bessere Lösung:

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

```
create new_stop1.set_station (Station_montrouge)
        -- Gleicher Effekt wie die zwei vorherigen Instruktionen
```

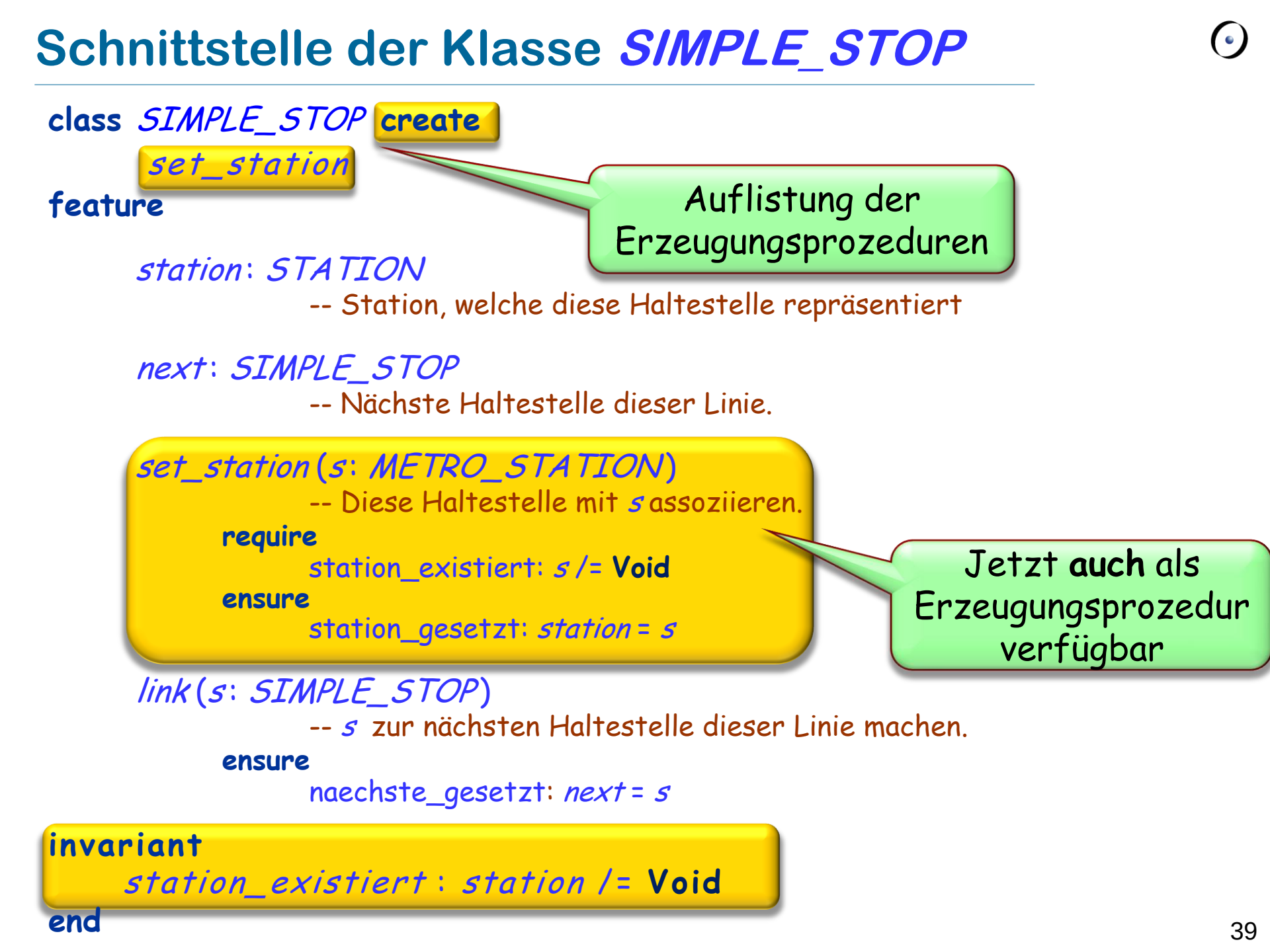
- Einfachheit: Initialisierung bei Erzeugung
- Korrektheit: Die Invariante wird von Anfang an erfüllt.

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

Die Schnittstelle der Klasse *SIMPLE_STOP*

```
class SIMPLE_STOP create
        set_station
feature

        station: STATION
                -- Station, welche diese Haltestelle repräsentiert

        next: SIMPLE_STOP
                -- Nächste Haltestelle dieser Linie.

        set_station (s: STATION)
                -- Diese Haltestelle mit s assoziieren.
            require
                station_existiert: s /= Void
            ensure
                station_gesetzt: station = s

        link (s: SIMPLE_STOP)
                -- s zur nächsten Haltestelle dieser Linie machen.
            ensure
                naechste_gesetzt: next = s

end
```

Schnittstelle der Klasse *SIMPLE_STOP*



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 ermöglicht dem Autor der Klasse, eine korrekte Initialisierung aller Instanzen, die Clients erzeugen, zu erzwingen.

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 gebraucht: die Vorbedingung.
- Was wir dafür garantiert bekommen: die Nachbedingung.

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

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.

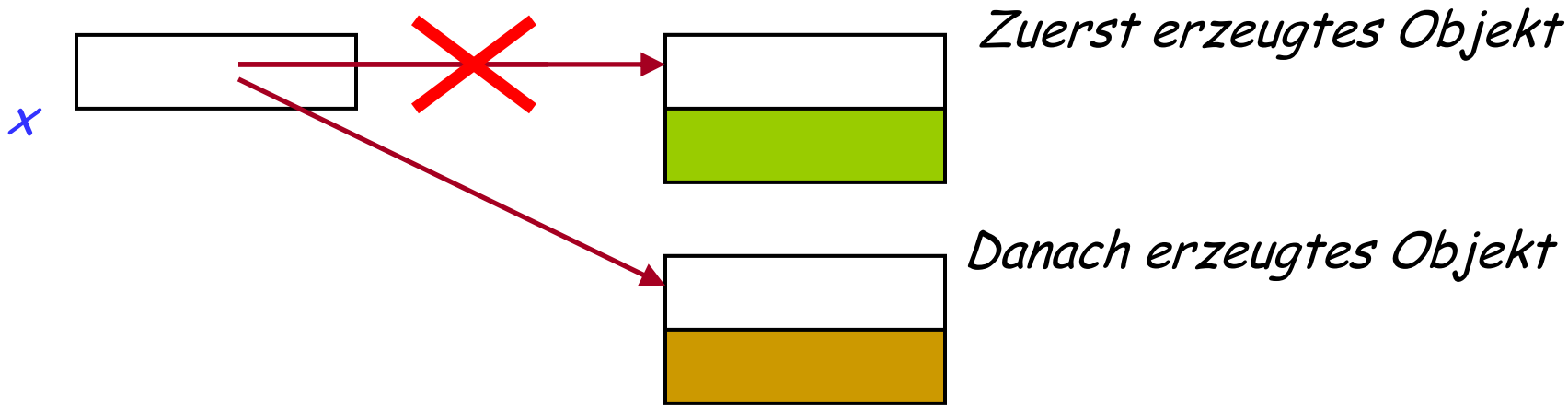
Nach der Erzeugungsinstruktion mit dem Ziel *x* vom Typ *C*:

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).
- Falls es eine Erzeugungsprozedur gibt, ist ihre Nachbedingung für das eben erzeugte Objekt erfüllt.
- Das Objekt erfüllt die Klasseninvariante.

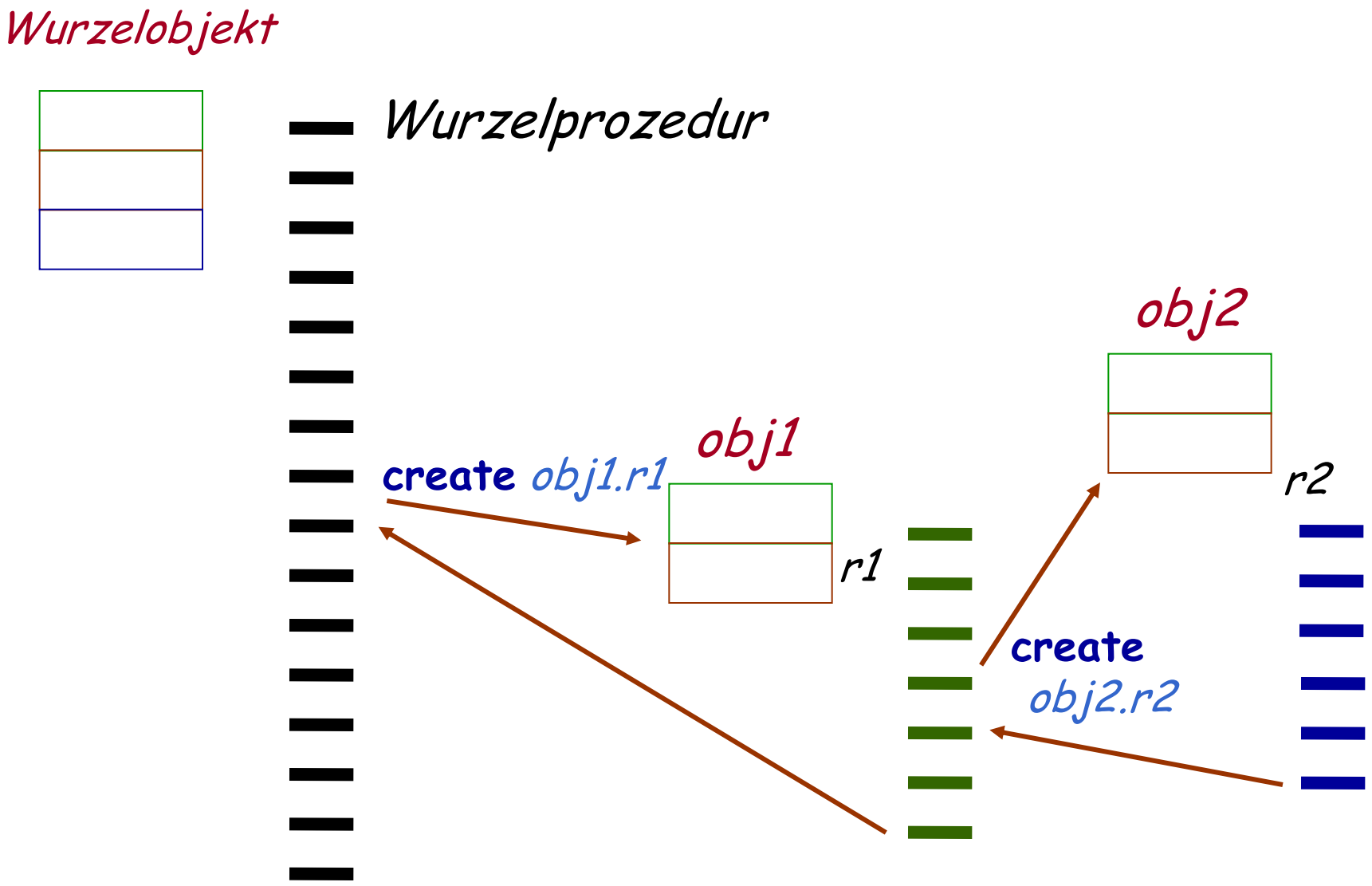
Wie alles beginnt...

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 **Wurzelprozedur**).

Eine Wurzel-Erzeugungsprozedur kann:

- Neue Objekte erzeugen
- Auf diese 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 welches 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 seine 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 heute gesehen haben

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