



# Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 4: Die Schnittstelle einer Klasse



## Definitionen

Ein **Kunde** (oder **Klient**) eines Softwaremechanismus ist ein System beliebiger Art

(Softwareelement, nicht-Software-System, menschlicher Benutzer...)

welches diesen nutzt.

Für seine Kunden ist der Mechanismus ein **Versorger**.

# Darstellung der Kunde-Beziehung



(Siehe Diagramm-Tool von EiffelStudio)





Eine **Schnittstelle** einer Menge von Softwaremechanismen ist die Beschreibung von Techniken, die es den *Kunden* ermöglicht, diese Mechanismen zu benutzen

# Arten von Schnittstellen (interfaces)

---



**Benutzerschnittstelle:** Kunden sind Menschen

- **GUI** (Graphical User Interface, häufig nur "UI"): Graphische Benutzeroberfläche (oder: Benutzerschnittstelle)
- Textschnittstellen, Befehlszeilen-Schnittstellen...

**Programmschnittstelle:** Kunden sind andere Softwaresysteme

- **API** (Abstract\* Program Interface): Programmierschnittstelle

Wir befassen uns in dieser Vorlesung mit Programmierschnittstellen

\*Früher: *Application*

# Eine graphische Benutzerschnittstelle (GUI)



The screenshot shows a Mozilla Firefox browser window displaying the website for the Chair of Software Engineering at ETH Zurich. The browser's address bar shows the URL <http://se.inf.ethz.ch/>. The website features a navigation menu with links for Home, People, Courses, Student Projects, Research, Publications, and Events. Below the menu is a banner image showing four people in a professional setting. The main heading is "Chair of Software Engineering".

## Welcome

The Chair of Software Engineering is devoted to the development of methods and tools for improving software quality. Although we attempt to cover the whole field of software engineering, our areas of emphasis are:

- Software verification
- Concurrency
- Persistence and evolution
- Object-oriented reengineering

As part of our activities we organize events such as the [TOOLS](#) conference series and the annual [Laser Summer School](#). Until 2010 we published the [Journal of Object Technology \(JOT\)](#).

## Upcoming events

[LASER 2012: Innovative Languages for Software Engineering](#), 2 - 8 September 2012 — Elba Island, Italy.

<b>Address</b>	<b>Contact</b>
Chair of Software Engineering, Meyer Clausiusstrasse 59 RZ Building 8092 Zurich Switzerland	Secretary: <a href="#">Claudia Günthart</a> Fax: +41 44 632 14 35

Frontpage photos by [Ivan Krechetov](#).



Ein **Objekt** ist eine Softwaremaschine, die es Programmen erlaubt, auf eine Ansammlung von Daten zuzugreifen und diese zu verändern

Objekte repräsentieren z.B. (in Traffic)

- Eine Stadt
- Eine Tramlinie
- Eine Route durch die Stadt
- Ein Element des GUI's, wie z.B ein Knopf (Button)

Jedes Objekt gehört zu einer gewissen **Klasse**, die die anwendbaren Operationen (**Features**) definiert

Beispiele:

- Die Klasse aller Städte
- Die Klasse aller Knöpfe
- etc.



## Klasse

Eine **Klasse** ist die Beschreibung einer Menge von möglichen Laufzeitobjekten, auf die die gleichen Features anwendbar sind

Eine **Klasse** repräsentiert eine Kategorie von Dingen

Ein **Objekt** repräsentiert eines dieser Dinge

## Instanz, generierende Klasse

Falls ein Objekt  $O$  eines der durch die Klasse  $C$  beschriebenen Objekte ist:

- $O$  ist eine **Instanz** von  $C$
- $C$  ist die **generierende Klasse** von  $O$

Eine **Klasse** repräsentiert eine Kategorie von Dingen

Ein **Objekt** repräsentiert eines dieser Dinge



Klassen existieren nur im **Softwaretext**:

- Definiert durch einen Klassentext
- Beschreiben Eigenschaften von assoziierten Instanzen

Objekte existieren nur zur **Laufzeit**:

- Sichtbar im Programmtext durch Namen, die Laufzeitobjekte **bezeichnen**

Beispiele: *Zurich\_map, Polyterrasse, console*

# Ein Objekt hat eine **Schnittstelle (interface)\***

\*von seiner generierenden Klasse definiert



# Ein Objekt hat eine **Implementation\***



\*von seiner generierender Klasse definiert





Passende Klassen zu finden ist ein zentraler Teil des **Softwaredesigns**

(Die Entwicklung der **Architektur** eines Programms)

Die Details auszuarbeiten ist ein Teil der **Implementation**



In dieser Vorlesung benutzen wir "Schnittstelle" im Sinne einer Programmierschnittstelle (nicht Benutzerschnittstelle).

Wir schauen uns jetzt die Schnittstelle von *SIMPLE\_LINE* (eine vereinfachte Version von *LINE*) an.

Diese wird in EiffelStudio angezeigt. (Benutzen Sie den "Interface" Knopf.)

# Eine Abfrage: "count"



Wie lange ist diese Linie? Siehe Abfrage *count*

```
count: INTEGER
```

```
-- Anzahl der Stationen auf dieser Linie.
```

**Kopfkommentar:** beschreibt den Zweck dieses Features.

"**diese Linie**": Die Instanz von *SIMPLE\_LINE*, auf die *count* angewendet wird.

Die Form einer Abfrage-Deklaration:

```
feature_name: RÜCKGABE_TYP
```

Möglicherweise mit  
Featurerumpf

*INTEGER*: ein Typ, der ganze Zahlen bezeichnet  
(z.B. -23, 0, 256)



**NO  
STOPPING  
ANYTIME**  
↔

**RED ZONE**  
**DON'T  
EVEN  
THINK  
OF  
PARKING  
HERE**  
SP-149C



*Denken* Sie nicht einmal daran, ein Feature zu schreiben, ohne sofort einen Kopfkomentar zu verfassen, der den Zweck des Features erläutert

Zur **Laufzeit** hat jedes Objekt einen Typ: seine generierende Klasse

Beispiele:

- *LINE* ist der Typ des Objektes, das *Line8* referenziert
- *INTEGER* ist der Typ des Objektes, das *Line8.count* referenziert

Im **Programmtext** hat jeder *Ausdruck* einen Typ

Beispiele:

- *LINE* ist der Typ von *Line8*
- *INTEGER* ist der Typ von *Line8.count*

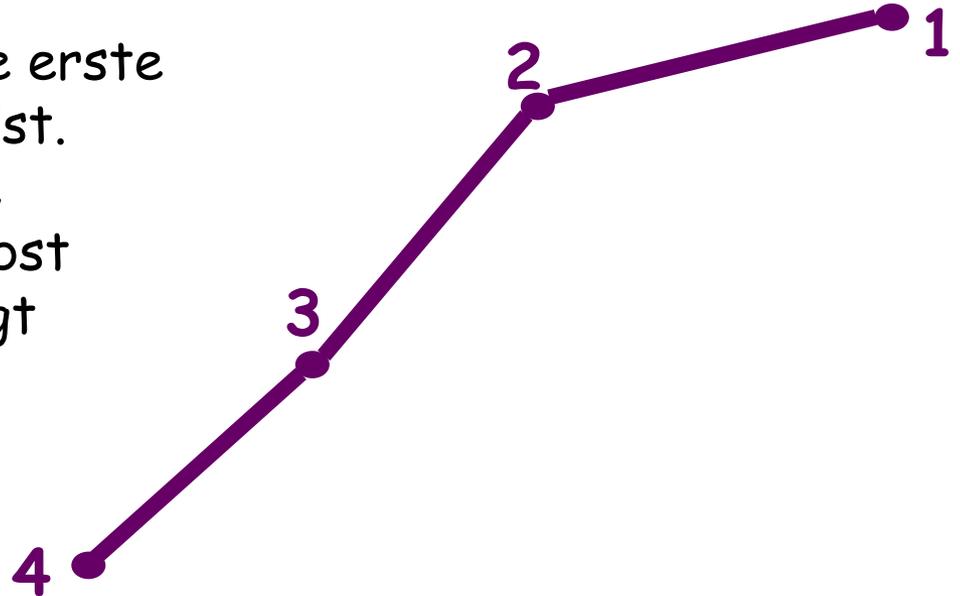
# Eine weitere Abfrage: *i\_th*



Welche ist die *i*-te Station einer Linie? Feature *i\_th*.

Es spielt keine Rolle welches die erste und welches die letzte Station ist. Traffic garantiert nur, dass die Reihenfolge sich nicht ändert, selbst wenn neue Stationen hinzugefügt werden.

In unserer Karte von Zürich ist die nördliche Endstation immer die Erste.



*i\_th* (*i*: INTEGER): STATION

-- Die Station mit Index *i* auf dieser Linie.

# Zwei weitere Abfragen

---



Welches sind die Stationen am Ende einer Linie?

*first: STATION*

-- Erste Station.

*last: STATION*

-- Letzte Station.

Eigenschaften jeder Linie *l*:

➤ *l.first = l.i\_th(1)*

➤ *l.last = l.i\_th(l.count)*

# Beispiele: die Klasse *QUERIES*

---



```
class QUERIES inherit  
  ZURICH_OBJECTS
```

```
feature
```

```
  explore
```

```
    -- Abfragen auf Linien ausprobieren.
```

```
  do
```

```
    console.output (Line10.count)
```

```
    console.output (Line10.i_th (1))
```

```
    console.output (Line10.i_th (Line10.count))
```

```
  end
```

```
end
```

# Ein Befehl: *remove\_all\_segments*

---



Wir möchten *Line10* von Grund auf neu bauen.

Wir beginnen damit, indem wir alle Stationen löschen:

*remove\_all\_segments*

-- Alle Stationen ausser der ersten entfernen.

Anmerkungen:

- Unsere Tramlinie hat immer mindestens eine Station, auch nach Anwendung des Befehls *remove\_all\_segments*
- Falls die Linie nur eine Station hat, bezeichnet sowohl *first* als auch *last* diese Station

# Der Befehl *append*

---



Neue Stationen zu einer Linie hinzufügen:

*append(s: STATION)*

-- *s* am Ende dieser Linie hinzufügen.

# Die Klasse *COMMANDS*



class *COMMANDS* inherit

*ZURICH\_OBJECTS*

feature

*explore*

-- Die Linie 10 wiederherstellen.

do

*Line10.remove\_all\_segments*

-- Es ist nicht nötig, *Haldenbach* hinzuzufügen.

*Line10.append (ETH\_Universitaetsspital)*

*Line10.append (Haldenegg)*

*Line10.append (Central )*

*Line10.append (Bahnhofplatz\_HB)*

-- Resultate anzeigen:

*console.output (Line10.count )*

*console.output (Line10.first.name )*

end

end



Nicht jedes Feature ist mit jedem Argument auf jede Instanz anwendbar!

➤ Beispiel: *Line10.i\_th(200)* ist falsch!

Die Klassenschnittstelle muss präzise genug sein, um daraus ihre korrekte Anwendung abzuleiten



Informationen zum Kopfkomentar hinzufügen:

```
i_th(i: INTEGER): STATION
```

```
-- Die i-te Station dieser Linie.
```

```
-- (Achtung: benutze nur mit i zwischen 1 und count, inklusive.)
```

Besser, aber immer noch nicht gut genug:

- Ein Kommentar ist nur eine informelle Erklärung
- Obige Einschränkung sollte eine verbindlichere Stellung in der Schnittstelle haben

# Verträge (contracts)

---



Ein Vertrag ist eine semantische Bedingung, die den Gebrauch einer Feature- oder Klasseeigenschaft charakterisiert

Drei Hauptarten:

- Vorbedingung (precondition)
- Nachbedingung (postcondition)
- Klasseninvariante (class invariant)

# Vorbedingung



Eine Eigenschaft, die ein Feature von jedem Kunden erwartet

```
i_th (i: INTEGER): STATION  
-- Die i-te Station dieser Linie
```

```
require
```

```
    nicht_zu_klein: i >= 1
```

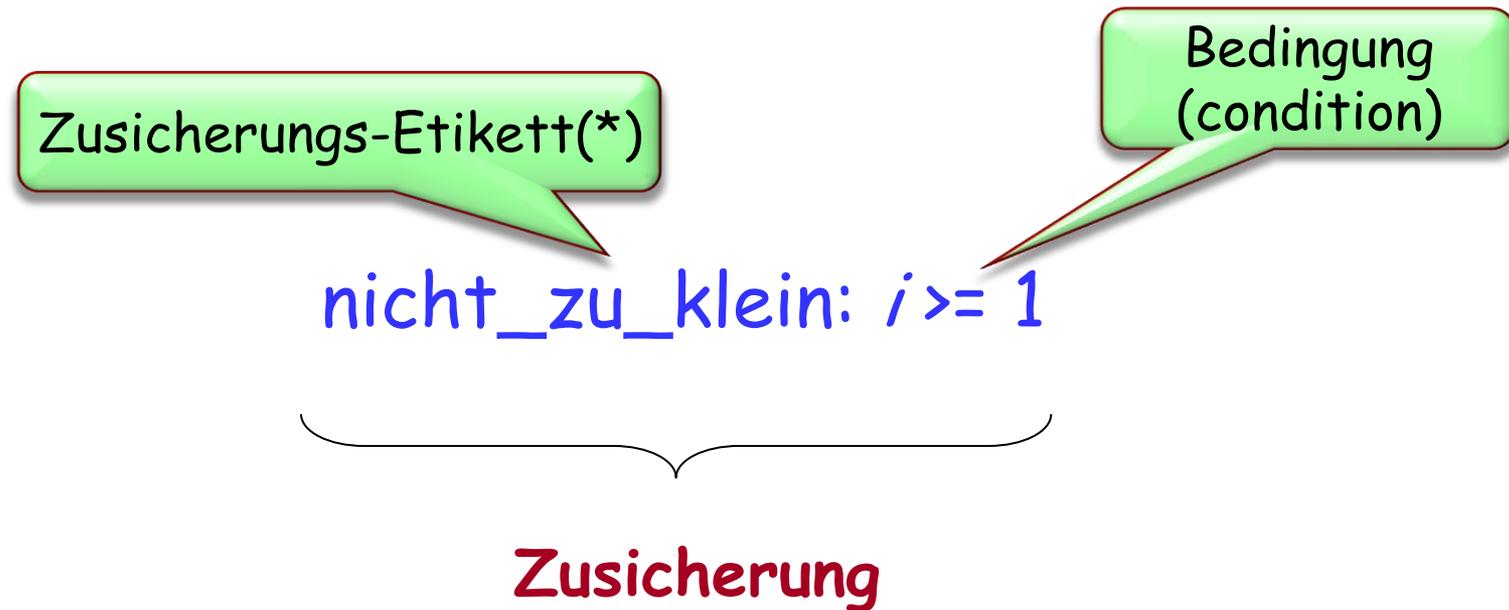
```
    nicht_zu_gross: i <= count
```

Die Vorbedingung  
von *i\_th*

Ein Feature ohne die **require** Klausel ist immer anwendbar, als ob es folgende Klausel hätte:

```
require
```

```
    immer_OK: True
```



(\*) oder: „Tag“, Sprich: „tääg“



Ein Kunde, der ein Feature aufruft, muss sicherstellen, dass die **Vorbedingung** vor dem Aufruf erfüllt ist

Ein Kunde, der ein Feature aufruft, ohne die Vorbedingung zu erfüllen, bezeichnet man als fehlerhafte ("buggy") Software



Verträge erleichtern den Prozess der Fehlerbeseitigung (Debugging)

Verträge sind auch nützlich als Dokumentation einer Schnittstelle

# Nachbedingungen



Vorbedingungen: Auflagen für Kunden

**Nachbedingungen:** Nutzen für Kunden

*remove\_all\_segments*

-- Alle Stationen ausser der ersten entfernen.

**ensure**

*nur\_eine\_bleibt: count = 1*

*beide\_enden\_gleich: first = last*

*append(s: STATION)*

-- *s* am Ende der Linie hinzufügen.

**ensure**

*neue\_station\_ist\_letzte: last = s*

*eine\_mehr: count = old count + 1*

Wert des  
Ausdrucks zum  
Zeitpunkt des  
Aufrufs

# Die old Notation

---



Nur in Nachbedingungen verwendbar

Bezeichnet den Wert eines Ausdrucks, den er beim Aufruf der Routine hatte

Beispiel (in einer Klasse *ACCOUNT*):

```
balance: INTEGER
    -- Aktueller Kontostand.

deposit (v: INTEGER)
    -- Addiere v zum Kontostand.
    require
        positiv: v > 0
    do
        ...
    ensure
        addiert: balance = old balance + v
    end
```



Ein Feature muss sicherstellen, dass, sofern seine Vorbedingung zu Beginn seiner Ausführung erfüllt wurde, seine Nachbedingung am Schluss erfüllt ist.

Ein Feature, welches seine Nachbedingung nicht erfüllen kann, nennt man fehlerhafte ("buggy") Software



- Klassen
- Objekte
- Den Begriff "Schnittstelle"
- GUI vs API
- Befehle und Abfragen
- Verträge: Vor- und Nachbedingungen (*Zusicherungen*)
- Verträge zur Fehlerbeseitigung und Dokumentation benutzen

# Zu lesen auf nächste Woche:

---



Kapitel 1 bis 6

Lesen Sie im Speziellen das Kapitel 5 (Logik), da wir nur kurz auf den in "*Diskrete Mathematik*" behandelten Teil eingehen werden und wir uns auf die Anwendungen in der Programmierung konzentrieren