ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 4: Die Schnittstelle einer Klasse

Kunde, Versorger (client, supplier)

Definitionen

Ein Kunde eines Softwaremechanismus ist ein System beliebiger Art (wie z.B. ein Softwareelement, ein nicht-Software-System oder ein menschlicher Benutzer), welches diesen nutzt

Für seine Kunden ist der Mechanismus ein Versorger.

Darstellung der Kunde-Beziehung

(Siehe Diagramm-Tool von EiffelStudio)

Informatics Europe Best Practices Awards

TInformatics Europe is launching a prize:

The Informatics Europe Curriculum Best Practices Award

The award recognizes outstanding educational initiatives that improve the quality of informatics teaching and the attractiveness of the discipline, and can be applied and extended beyond their institutions of origin.

The award will be funded through industrial sponsorship; the 2011 awards are sponsored by Intel

Each edition of the award may be focused on one or more particular areas of informatics teaching. The focus of the 2011 award will include concurrency, with a prize of about EUR 30,000, and embedded systems, with a prize of about EUR 10,000

The award will be overseen by a committee appointed by the Board of Informatics Europe. Details will be published by Nov. 21.

Schnittstelle: Definition

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

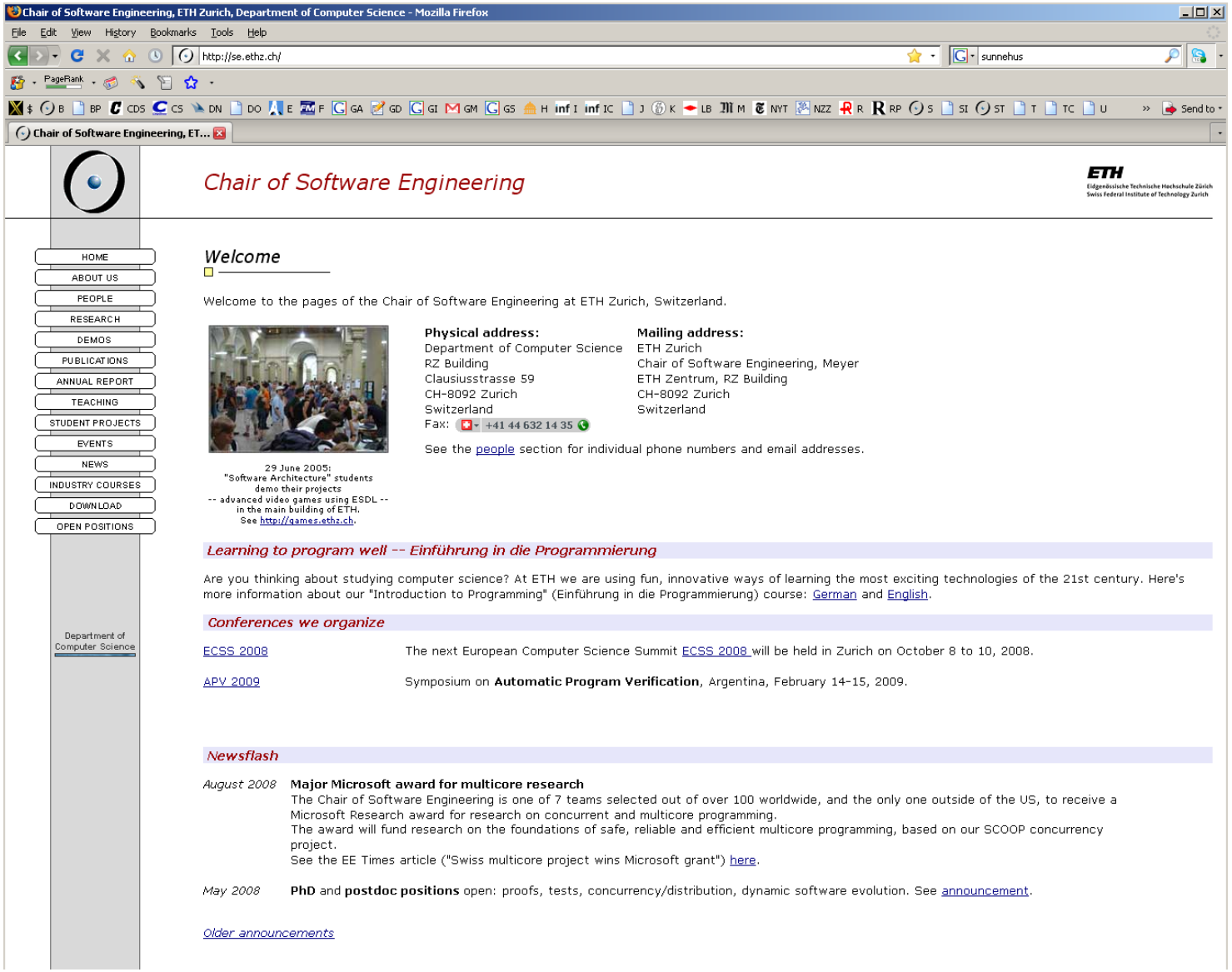
- GUI (Graphical User Interface): Graphische Benutzeroberfläche (oder: Benutzerschnittstelle)
- Textschnittstellen, Befehlszeilen-Schnittstellen...

Programmschnittstelle: Kunden sind andere Softwaresysteme

- API (Application (oder: Abstract) Program Interface): Programmierschnittstelle

Wir befassen uns jetzt mit Programmierschnittstellen

Eine graphische Benutzerschnittstelle (GUI)



Klassen

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.

Definition: Klasse

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.

Definitionen: Klasse, Instanz, generierende Klasse

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.

Objekte vs Klassen

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**.

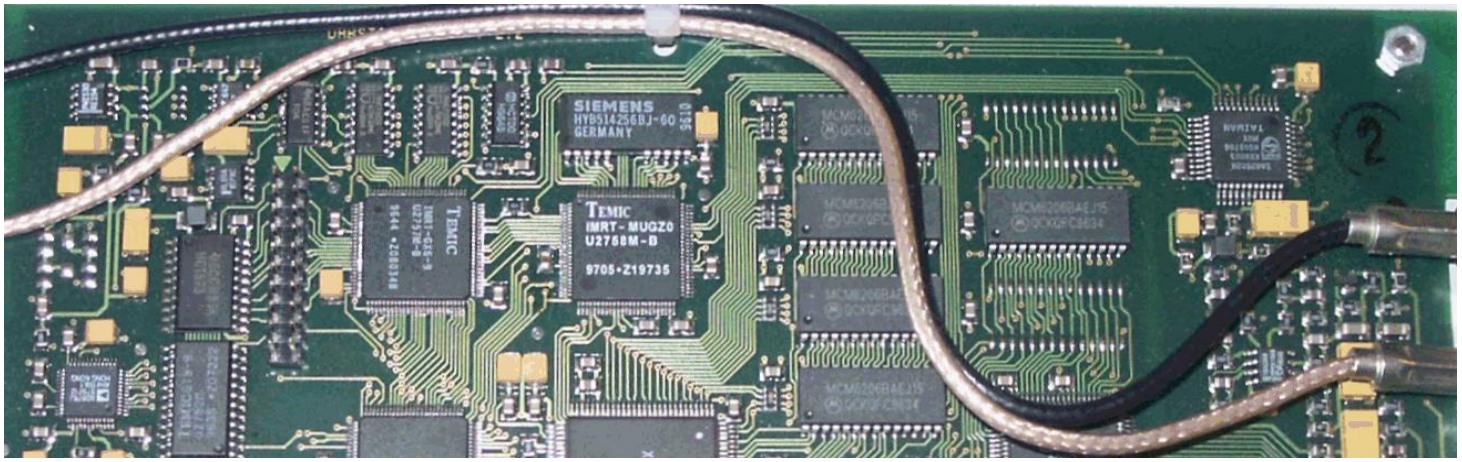
Beispiel: *Paris*

Ein Objekt hat eine Schnittstelle (interface)*

*von seiner generiender Klasse definiert

Ein Objekt hat eine **Implementation***

*von seiner generiender Klasse definiert



Softwarekonstruktion

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

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

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

Eine Klassenschnittstelle

Hier 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.)

Vergessen Sie nicht, das *TRAFFIC_* Präfix zu allen Klassen der Traffic Bibliothek hinzuzufügen. Benutzen Sie in EiffelStudio *TRAFFIC_LINE* und *TRAFFIC_SIMPLE_LINE* .

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-145C

Stilregel: Kopfkommentare

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

Ausdrücke und ihre Typen

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:

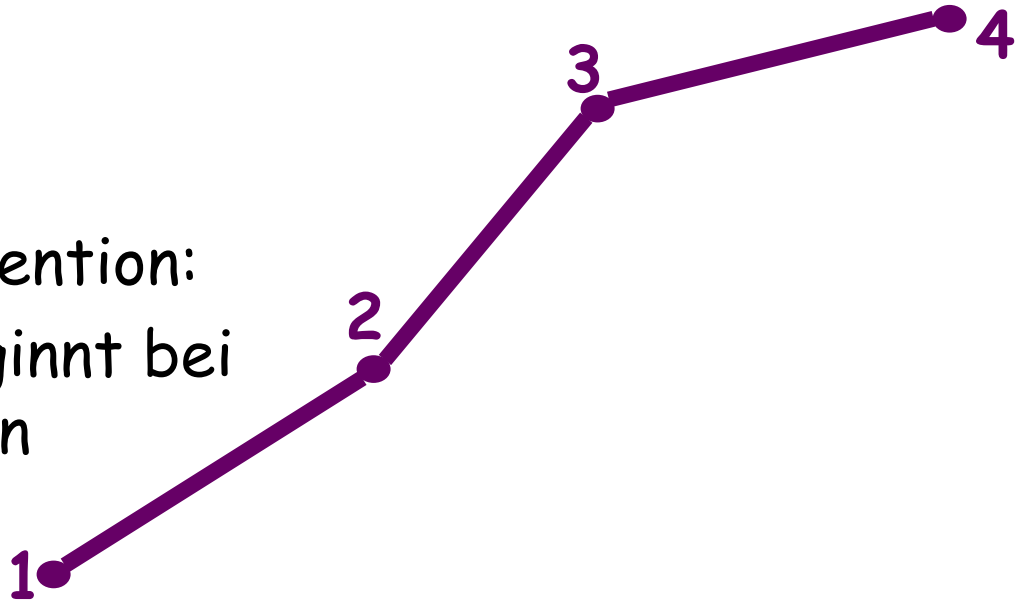
- *LINE* ist der Typ von *Line8*
- *INTEGER* ist der Typ von *Line8.count*

Eine weitere Abfrage: *i_th*

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

Konsistenz durch Konvention:

Die Nummerierung beginnt bei der südlichsten Station (south end)




```
i_th (i: INTEGER): STATION
        -- Die Station mit Index i an dieser Linie.
```

Zwei weitere Abfragen

Welches sind die Stationen am Ende einer Linie?

```
south_end: STATION
        -- Südliche Endstation
north_end: STATION
        -- Nördliche Endstation
```

Eigenschaften jeder Linie *l*:

- *l.south_end = l.i_th (1)*
- *l.north_end = l.i_th (l.count)*

Beispiele: die Klasse *QUERIES*

```
class QUERIES inherit
    TOURISM
 feature
        explore_on_click
            -- Abfragen auf Linien ausprobieren.
          do
            Paris.display
            Console.show (Line8.count)
            Console.show (Line8.i_th (1))
            Console.show (Line8.i_th (Line8.count))
          end
end
```

Ein Befehl: *remove_all_segments*

Wir möchten *Line8* von Grund auf neu bauen.

Wir beginnen damit, indem wir alle Segmente löschen:

```
remove_all_segments
        -- Alle Segmente ausser der südlichen
        -- Endstation entfernen.
```

Anmerkungen:

- Unsere Metrolinie hat immer mindestens eine Station, auch nach Anwendung des Befehls *remove_all_segments*
- Falls die Linie nur eine Station hat, bezeichnet sowohl *south_end* als auch *north_end* diese Station

Der Befehl *extend*

Neue Stationen zu einer Linie hinzufügen:

```
extend (s: STATION)
        -- s am Ende dieser Linie hinzufügen.
```

Die Klasse *COMMANDS*

```
class COMMANDS inherit
        TOURISM
 feature
        explore_on_click
                -- Einen Teil der Linie 8 wiederherstellen.
            do
                Line8.remove_all_segments
                        -- Nicht nötig, Station_Balard hinzuzufügen, da
                        -- remove_all_segments die südl. Endstation beibehält.
                Line8.extend (Station_Lourmel)
                Line8.extend (Station_Boucicaut)
                Line8.extend (Station_Felix_Faure)
                        -- Keine weiteren Stationen, und Resultate anzeigen:
                Console.show (Line8.count)
                Console.show (Line8.north_end.name)
            end
end
```

Saubere Schnittstellen definieren

Nicht jedes Feature ist mit jedem Argument auf jede Instanz anwendbar

Beispiel: *Line8.i_th (200)* ist falsch!

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

Erster Versuch...

Informationen zum Kopfkommentar 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 offiziellere Stellung in der Schnittstelle haben

Verträge (contracts)

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

Drei Hauptarten:

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

Vorbedingung

Eine Eigenschaft, die ein Feature von jedem Kunden erwartet

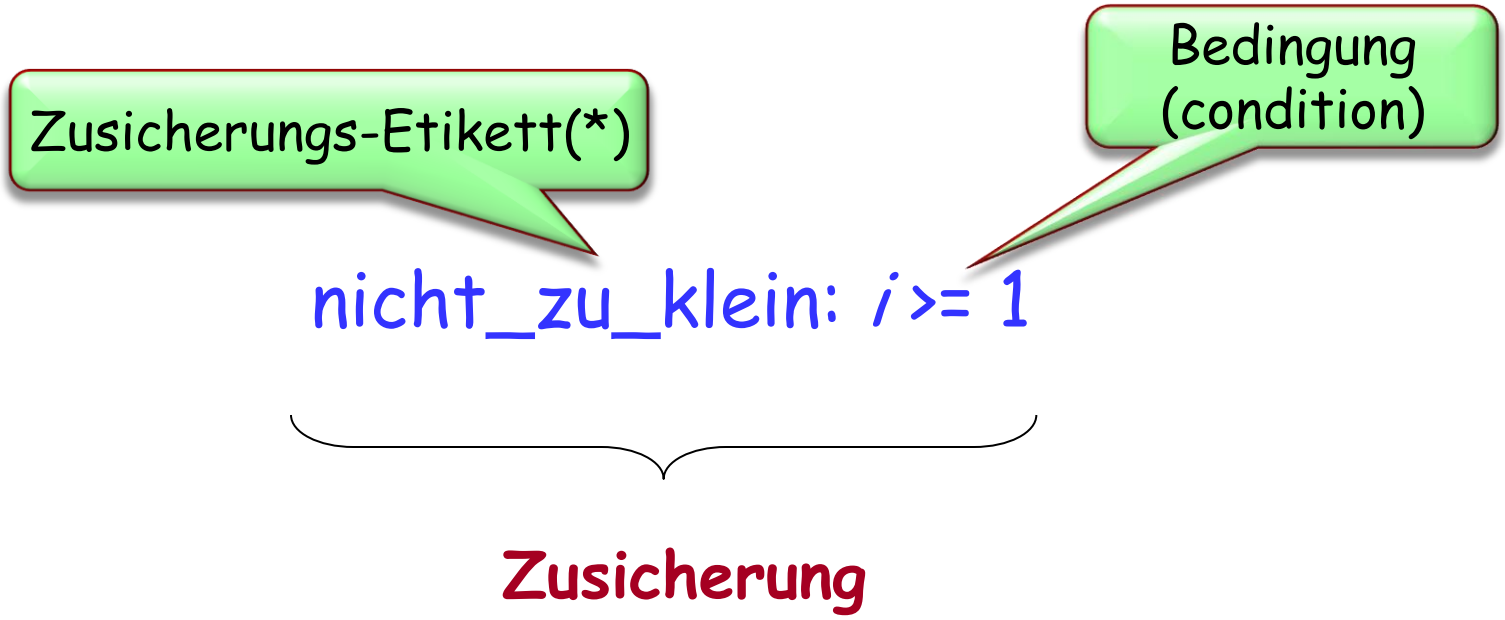
```
i_th (i: INTEGER): STATION
        -- Die i-te Station dieser Linie
    require
        nicht_zu_klein: i >= 1
        nicht_zu_gross: i <= count
```

Die Vorbedingung von *i_th*

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

```
require
    immer_OK: True
```

Zusicherungen (assertions)



(*) oder: „Tag", Sprich: „täág"

Das Vorbedingungsprinzip

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

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 südlichen Endstation entfernen
    ensure
        nur_eine_bleibt: count = 1
        beide_enden_gleich: south_end = north_end
```

```
extend (s: STATION)
        -- s am Ende der Linie hinzufügen.
    ensure
        neue_station_da: i_th (count) = s
        im_norden: north_end = 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
```

Das Nachbedingungsprinzip

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.

Was wir heute gesehen haben:

- 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