# *Gliederung*

- Warum testet man?
- Was ist Testen?
- Wogegen wird getestet?
- Der Tester
- Testen im Software-Entwicklungsprozess
- Korrektheit, Zuverlässigkeit, Fehler
- Klassifikation analytischer Verfahren
- Kontrollflussgraph
- Strukturtests
  - *Zweigüberdeckungstest*
  - *Boundary interior-Pfadtest*

# *Warum testet man?*

- in vielen Lebensbereichen der heutigen Gesellschaft auf fehlerfreie Funktionieren von SW-Systemen angewiesen → fehlerhafte SW oder Ausfall von SW kann zu grossem materiellen, finanziellen Schaden führen oder sogar Menschenleben gefährden → **Ziel:** ein möglichst fehlerfreies Produkt entwerfen
- kein Produkt ist fehlerfrei, Testen soll Fehler finden
- Testen versucht Fragen zu klären:

  *Erzeuge ich das richtige Produkt?* und

  *Erzeuge ich das Produkt richtig?*

# *Was ist Testen?*

- Testen ist ein komplexer Prozess
  beinhaltet:
  - Planung
  - Durchführung
  - Dokumentation der Testfällen
  - Erstellung von Testberichten
- Testen ist ein integraler Bestandteil des Software-Entwicklungsprozesses
- Messen des Grades der Funktionserfüllung (Korrektheit von Schnittstellen und Modulen, Anwenderfreundlichkeit, Flexibilität)
- Versuch des Nachweises der Fehlerfreiheit einer SW-Komponente

# *Wogegen wird getestet?*

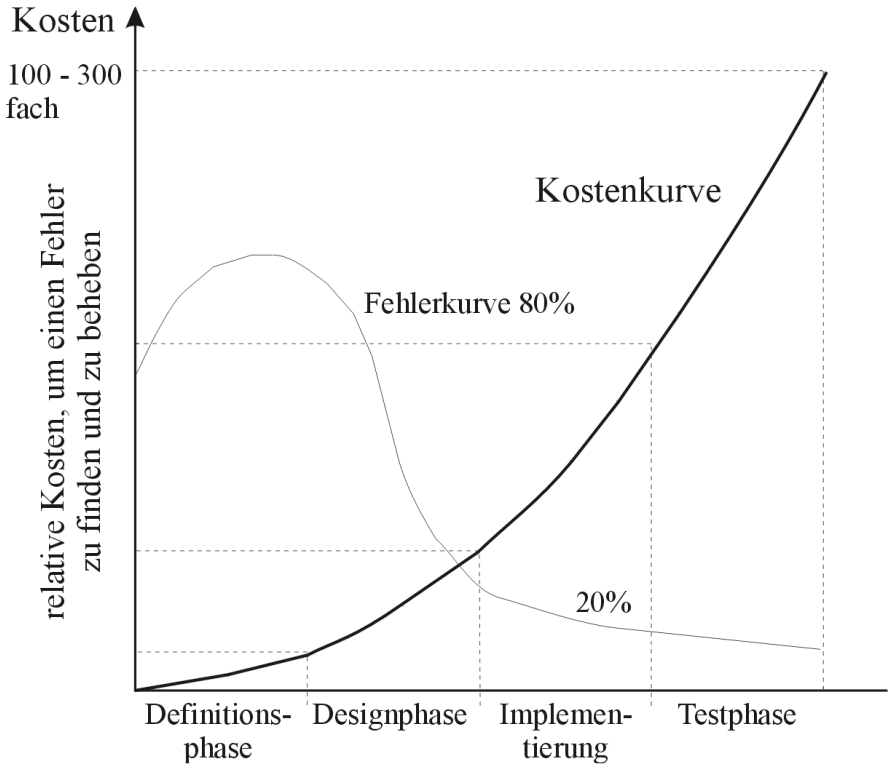
- gegen Systemrequirements
  - unverzichtbare Voraussetzungen für systematisches Testen
  - Maßstab an dem Programm und Programmfunktionen als korrekt oder inkorrekt beurteilt werden
- gegen Standards
  - eingehalten oder nicht eingehalten?

- Exkurs: **Systemrequirements**
  - enthält alle Anforderungen, die Produkt erfüllen muss
  - *Was soll erfüllt werden*? und *Wie soll es erfüllt werden?*
  - legt offen, wie genau Kunde weiss, was er will
  - klare, einfache, kurze, eindeutige und ausreichend detaillierte Aussagen

# *Der Tester*

- **Darf nicht...**
  - der Programmierer selbst sein
  - zu viel Nähe zum Entwicklerteam haben
  - zu viel Nähe zum Vertrieb haben
- **Muss bzw. sollte...**
  - ohne Rücksicht auf Verluste Fehler finden
  - zielstrebig sein
  - Höchstmaß an Kreativität besitzen
  - Person sein, die immer die Fragen stellt, auf die man **nicht** vorbereitet ist
  - muss Fachgebiet kennen, in dem Produkt eingesetzt wird
  - ausreichende Kenntnisse über Produkt haben
  - solide Kenntnisse über Testverfahren bzw. Testmethoden

# *Testen im SW-Entwicklungsprozess*




- Qualität von Software durch Qualitätsziele zu sichern
- Schwerpunkte der Qualitätssicherung: Produktdefinition, Produktentwurf, Produktimplementierung
- Testen ist Teil der analytischen Qualitätssicherung
- bei konventionellen Vorgehensweise, Testen erst relativ spät am Ende des Entwicklungsprozesses

- Testen muss Softwareentwicklung begleiten und in Entwicklungsprozess integriert sein, um frühzeitige Fehlererkennung zu realisieren

# *Korrektheit, Zuverlässigkeit, Fehler*

- **Korrektheit:** Fehlerfreiheit, Grad der Konsistenz zwischen Spezifikation und Programm, Grad der Erfüllung der Benutzererwartung (ANSI)

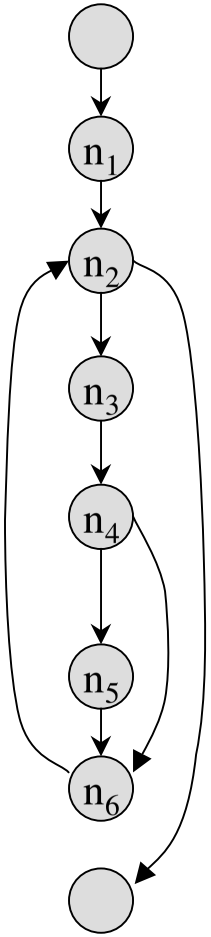
- **Zuverlässigkeit und Robustheit:** Eigenschaften eines Programmes, auf externe Ereignisse sinnvolle Reaktionen zu liefern und in extremen Situationen definiert zu arbeiten

- **Fehler ist ...**
  - ein Fehlschuss = Ziel verfehlt = Anforderungen nicht erfüllt
  - jede Abweichung von den Anforderungen des Auftraggebers
  - jede Inkonsistenz der Implementierung zur Spezifikation
  - jedes strukturelle Merkmal des Programmtextes, das ein fehlerhaftes Verhalten verursacht

# *Klassifikation analytischer Verfahren*

- **Testende Verfahren**
  - dynamische Testverfahren
    - Strukturtest (White Box-Test)
      - *Kontrollflussorientierter Test*
      - *Datenflussorientierter Test*
    - Funktionaler Test (Black Box-Test)
      - *Äquivalenzklassenbildung, Grenzwertanalyse, Zufallstest*
  - statische Testverfahren
    - manuelle Prüfmethoden (Inspektion, Reviews)
    - statistische Analysatoren
- **Verifizierende Verfahren**
  - Verifikation, Symbolische Ausführung
- **Analysierende Verfahren**
  - Metriken, Grafiken, Tabellen, Anomalienanalyse

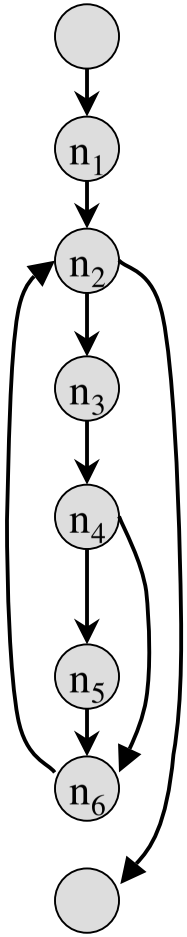
# *Kontrollflussgraph*




```
cin >> c;

while((c >= ‚A‘) && (c <= ‚Z‘)
      && gesamt < INT_MAX))

gesamt++;

if ((c == ‚A‘) || (c == ‚E‘) ||
    (c == ‚I‘) || (c == ‚O‘) ||
    (c == ‚U‘))

vokale++;

cin >> c;
```

- **Kontrollflussgraph**:
  - gerichteter Graph G = (N, E, $n_{start}$, $n_{final}$)
  - N endliche Menge der Knoten
  - E $\subseteq$ N x N: Menge der gerichteten Kanten
  - $n_{start} \in$ Startknoten, $n_{final} \in$ Endknoten
- jeder Knoten = ausführbare Anweisung
- **Zweig**: gerichtete Kante von Knoten i zu j, die möglichen Kontrollfluss von i nach j beschreibt
- **Pfad:** abwechselnde Folge von Kanten und Knoten die mit Startknoten beginnt und mit Endknoten endet

# *Zweigüberdeckungstest(1)*




```
cin >> c;

while((c >= ‚A‘) && (c <= ‚Z‘)
      && gesamt < INT_MAX))

gesamt++;

if ((c == ‚A‘) || (c == ‚E‘) ||
    (c == ‚I‘) || (c == ‚O‘) ||
    (c == ‚U‘))

vokale++;

cin >> c;
```

- kontrollflussorientiertes Testverfahren
- **Ziel:** Überdeckung aller Zweige
- **Metrik:** $C_{Zweig}$ = # ausgeführter Zweige/ Gesamtanzahl der vorhandenen Zweige
- **Eigenschaften**:
  - 100% prozentige Zweigüberdeckung stellt sicher, dass alle Zweige laufen wurden
  - weder Kombination von Zweigen noch komplexe Bedingungen berücksichtigt
  - Schleifen nicht ausreichend getestet
  - fehlende Zweige nicht zu entdecken
- **Beispiel:**
  eingelesene Zeichen: ‘A‘,‘B‘,‘1‘, Pfad ($n_{start}$ , $n_1$,$n_2$,$n_3$ ,$n_4$ ,$n_5$ ,$n_6$ ,$n_2$ ,$n_3$ ,$n_4$ ,$n_6$ ,$n_2$ ,$n_{final}$)

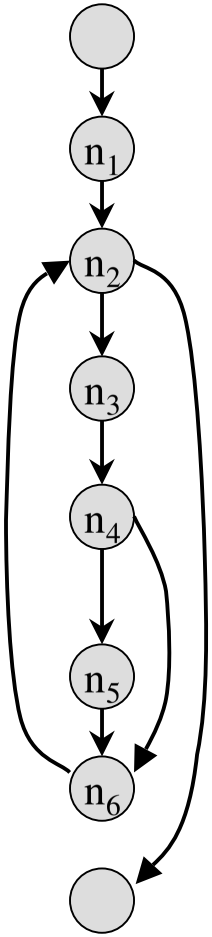
# *Zweigüberdeckungstest(2)*

- **Leistungsfähigkeit**:
  - findet 79% Kontrollflussfehler und 20% Berechnungsfehler
  - Leistungsfähigkeit schwankt zwischen 25% - 75% oder 67% - 100% (Infotech Vol.I 79)
  - Erfolgsquote höher als bei statischen Analyse (Gannon 79)
  - gute Leistungsfähigkeit bei Behandlung von logischen Fehlern und Fehlern bei Berechnungen
  - Erkennung von Datenfehlern fast unmöglich
- **Bewertung**:
  - das *minimale* Testkriterium
  - nicht ausführbare Programmzweige können gefunden werden
  - Korrektheit des Kontrollflusses an Verzweigungsstellen überprüfbar
  - oft durchlaufene Programmteile erkennbar und gezielt optimierbar

# *boundary interior-Pfadtest(1)*

- Pfadüberdeckungstest: ausreichender Test von Schleifen → Ausführung aller unterschiedlichen Pfade des zu testenden Programmes
- **boundary interior-Pfadtest**: schwächere Version des Pfadüberdeckungstests
- **Ziel:** beim Test von Schleifen auf Überprüfung von Pfaden verzichtet, die durch mehr als einmalige Schleifenwiederholung erzeugt werden
- **Eigenschaften:**
  existieren zwei Pfadgruppen:
  - *Grenztest Gruppe* (boundary test): alle Pfade, die Schleife betreten, aber nicht wiederholen, alle Pfade ausgeführt, die unterschiedliche Pfade innerhalb des Schleifenkörpers verfolgen
  - *Gruppe zum Test des Schleifeninneren* (interior test): alle Pfade mit mindestens einer Schleifenwiederholung., Testfälle verfolgen alle unterschiedlichen Pfade während der ersten beiden Ausführungen des Schleifenkörpers

# *boundary interior-Pfadtest(2)*




```
cin >> c;

while((c >= ‚A‘) && (c <= ‚Z‘)
      && gesamt < INT_MAX))

gesamt++;

if ((c == ‚A‘) || (c == ‚E‘) ||
    (c == ‚I‘) || (c == ‚O‘) ||
    (c == ‚U‘))

vokale++;

cin >> c;
```

- **Bewertung:**
  - gezielte Überprüfung von Schleifen
  - gegenüber Pfadüberdeckungstest praktikabel
- **Beispiel-Testfälle:**
  - Testfall für Pfad ausserhalb der Schleife: gesamt = INT_MAX
  - Grenztest: gesamt = 0, c = ‚A‘, ‚1‘, Schleife einmal durchlaufen + THEN Zweig
    gesamt = 0, c = ‚B‘, ‚1‘ Schleife einmal durchlaufen + nicht THEN Zweig
  - Testfälle für Schleifenkörper: z.B.: c = ‘E‘,‘I‘,‘N‘,‘*‘