Universität Tübingen
Wilhelm Schickard Institut für Informatik
Technische Informatik
Seminar "Pleiten, Pech und Pannen "
Betreuer: Gerald Heim
SS 2001

# Informelle Methode und Compilertechniken

zur Aufdeckung von Programmierfehlern

Christoph Probst
Schleifmühleweg 30
72070 Tübingen

Informatik
Diplom
4. Semester

Inhaltsverzeichnis


1. Typisierung

1.1. Statische Typisierung
1.2. Strenge Typisierung
1.3. Dynamische Typisierung
1.4. Wählbare Typisierung/Zusammenfassung

2. Statische Analyse

2.1. Bedingungen für eine Statische Analyse
2.2. Ziele der Statische Analyse – moderne Compiler
2.3. Ziele der Statischen Analyse – Analyse Tools / manuelle Analyse
2.4. Zusammenfassung

3. Code Review

3.1. Walkthrough
3.2. Code-Inspektion
3.3. Zusammenfassung

4. Dokumentation

4.1. Quellcode-Dokumentation
4.2. Programm-Dokumentation
4.3. Anwender-Dokumentation

5. Beispiel

6. Quellen

# Einleitung

„Houston ... we have a problem“ – wenn diese Worte gefallen sind, dann ist es vermutlich schon zu spät. Ein Computerfehler, ein unvorhergesehenes Hard- oder Softwareproblem, jede kleinste Abweichung von der Spezifikation kann Millionen kosten – Menschenleben oder Euro. Ein Ziel bei der Entwicklung von Hard- und Software muss es also sein, alle Fehler bereits bei der Entwicklung zu entdecken und zu beseitigen. Für fast jeden Entwicklungsschritt gibt es heutzutage Testverfahren, Simulationsmöglichkeiten und Analysen.

Dieser Vortrag beschäftigt sich mit der Fehlervermeidung und –erkennung bei der Softwareentwicklung. Beginnend bei der Programmierung gibt es verschiedene Techniken, mit denen versucht wird, Programmierfehler zu finden. Aller hier vorgestellten Techniken haben den Vorteil, dass sie bereits vor der Fertigstellung des Programms auf den Programmcode bzw. einen Ausschnitt daraus angewendet werden können.

# 1. Typisierung

Die meisten höheren Programmiersprachen sind streng typisiert. Das bedeutet, dass jeder innerhalb des Programms gespeicherte Wert zur Laufzeit einen Typ (z. B. String, Integer, etc.) bekommt. Beim Compilieren wird nun überprüft, ob jeder Wert im Programm dem Typ nach zu den Funktionen, Methoden und Variablen passt, die ihn benutzen. Gibt es bei dieser Prüfung bereits Fehler, so bricht die Compilierung mit einem Fehler ab.

Eine Typisierung ist letztendlich unumgänglich. Eine arithmetische Operation erwartet z. B. im Normalfall einen als Zahl interpretierbaren Wert. Es stellt sich letztendlich nur die Fragen, **wann** die Typenüberprüfung stattfinden soll.

## 1.1. Statische Typisierung

Programmiersprachen wie z. B. Turbo Pascal fordern eine statische Typisierung. Bereits bei der Programmierung wird jedem Wert ein fester Typ zugewiesen. Unterschiedliche Typen sind nicht kompatibel. Es gibt statt dessen spezielle Funktionen, die einen Typ in einen anderen überführen (z. B. Val, Str).

Vorteile:

- Effizienzgewinn zur Laufzeit, da keine Typüberprüfungen mehr statt finden müssen.
- Verhinderung von Programmabbrüchen durch inkompartibele Typen. (Sicherer Code durch Typkonsistenz)
- Aufdeckung von Programmierfehlern zum Zeitpunkt der Compilierung.
- Optimierung durch den Compiler möglich. (Konstante Variablen werden zu Konstanten, u. v. m.)
- Statische Analyse möglich (siehe 2.)

Nachteile:

- Mehrfachnutzung von Code für verschiedene Typen ist nicht möglich. „Ad hoc polymorphe Sprachen“[7] bieten darum Techniken wie „Überladen“ bzw. „Typenanpassung“ an, um verschiedene Typen zu vereinen.
- Geringe Flexibilität: z. B. können vorhandene Schnittstellen zu Funktionen können nicht mit neuen Objekten/Typen benutzt werden. Besonders, wenn eine Funktion nicht im Quellcode vorliegt, sind solche Erweiterungen relativ umständlich.

*Beispiel 1.1.1 (Turbo Pascal): Fehlerhaftes Programm*

```
var i:integer;

begin
  i:=1;

  writeln('Die Zahl lautet ' + i);
end.
```

Statische Typisierung verhindert, dass der Text „Die Zahl lautet“ mit der Zahl i auf diese Art ausgegeben wird. Bereits das Kompilieren schlägt fehlt.

*Beispiel 1.1.2 (Turbo Pascal): Mögliche Behebung des Fehlers*

```
var i:integer;
    s:string;
begin
  i:=1;
  str(i, s);
  writeln('Die Zahl lautet ' + s);
end.
```

Wandelt man die Zahl mit str() in einem String um, so läßt sie sich mit dem Text kombinieren. str() ist dabei die Funktion, die die Typumwandlung durchführt.

*Beispiel 1.1.3 (Java): Beispiele*

Nicht existierende Methoden:
```
Matrix.flieg();
```
Falsche Anzahl von Parametern:
```
Vektor.add();
```
Falsche Werte:
```
int erg = "hallo" + 3;
```

Diese Funktionen würden vom Compiler auf Grund statischer Typisierung als fehlerhaft erkannt. Weder besitzt das Objekt Matrix die Methode flieg() noch kann man add() bei einem Vektor ohne Parameter aufrufen. Ebenso die Addition von Text und Zahl kann nicht in einem int gespeichert werden.

## 1.2. Strenge Typisierung

Die streng typisierten Sprachen sind eine Erweiterung der statisch typisierten Sprachen. Es wurde, um die Programmierung flexibler zu gestalten, der sogenannte „Typecast“, hinzugefügt. Dabei macht der Programmierer eine Zusicherung, dass zwei Typen zueinander passen – die Typumwandlung wird jedoch erst zur Laufzeit durchgeführt. (z. B. Java oder C++).

Vorteile:

- Weil die Strenge Typisierung die Statische erweitert, gelten weiterhin einige Vorteile der Statischen Typisierung
- Es können allgemeinere Typen benutzt werden (z. B. „Object“).
- Unterschiedliche Typen können relativ einfach vermischt werden (Typecast).

Nachteile:

- Siehe auch statische Typisierung.
- Teilweise Typüberprüfung zur Laufzeit => Keine Typkonsistenz mehr.

*Beispiel 1.2.1 (Java): Fehler durch Typ-Cast*

```
public class test {
    public static void main(String args[])
    {
       Object a = "test";
       Integer i = (Integer)a;
    }
}
```

Diese Klasse wird anstandslos vom Java-Compiler kompiliert. Erst zur Laufzeit bricht das Programm durch folgenden Fehler bei der Typumwandlung ab:

*Beispiel 1.2.2 (Java): Fehlermeldung zum Beispiel 1.2.1*

```
java.lang.ClassCastException: java.lang.String
        at test.main(Compiled Code)
```

## 1.3. Dynamische Typisierung

Werden bei der Compilierung keinerlei Typüberprüfungen mehr durchgeführt, so spricht man von einer dynamischen Typisierung zur Laufzeit. Die Sprachen Scheme (Lisp), Basic, sowie viele Skript- und Interpretersprachen benutzen diese Art von Typisierung.

Vorteile:

- Man kann bei der Programmierung vom Typ der Werte abstrahieren.
- Funktionen können für beliebige Typen geschrieben werden. (=> Mehrfachnutzung von Programmcode)
- Flexibler: Code kann zur Laufzeit manipuliert werden.

Nachteile:

- Inkompatibele Typen können zu Programmabstürzen führen.
- Typüberprüfung während der Laufzeit
  → extra Code nötig zur Typüberprüfung
  → extra Speicherverbrauch, dadurch evtl. Caching-Nachteile
  → längere Laufzeiten durch Typüberprüfung
- Code kann bei der Compilierung nicht auf die Datentypen hin optimiert werden.

*Beispiel 1.3.1 (Scheme): Add-Funktion*

```
(define add
  (lambda (a b)
    (+ a b)))

(add 1 2)
(add 3.45 1.23)
(add (expt 2 100000) 1)
```

Die Funktion `add` addiert beliebige Zahlen – selbst Dezimal- oder große Zahlen ($2^{100000}$). Veränderungen an der Funktion `add` sind dazu nicht nötig.

*Beispiel 1.3.2 (Scheme): Probleme mit der dynamisch getypten Add-Funktion*

```
(add "a" 4)

+: expects type <number> as 1st argument, given: "a"; other
arguments were:
```

Will man allerdings einen Buchstaben mit einer Zahl addieren, so kommt es zur Laufzeit zum Programmabbruch. Beachtenswert ist in diesem Beispiel die Tatsache, dass die Funktion `add` keine Probleme mit der Addition von Text und Zahl hat. Erst die Funktion + erzeugt die Ausnahme.

### 1.4. Wählbare Typisierung/Zusammenfassung

Um die Vorteile aller Typisierungen zu vereinen, wird in modernen Programmiersprachen versucht, Strenge oder Dynamische Typisierung zu vereinen (z. B. Java oder Dylan[1]). Es soll Programmierer freigestellt bleiben, ob er statische Typisierung (mit den entsprechenden Optimierungen), oder dynamische Typisierung (mit den entsprechenden Freiheiten) nutzen will.

Die Wahl der Typisierung ist im allgemeinen von der Art der Anwendung abhängig. Je mehr Sicherheit in einem Programm nötig ist, desto weniger Freiheiten hat man bei der Typisierung.

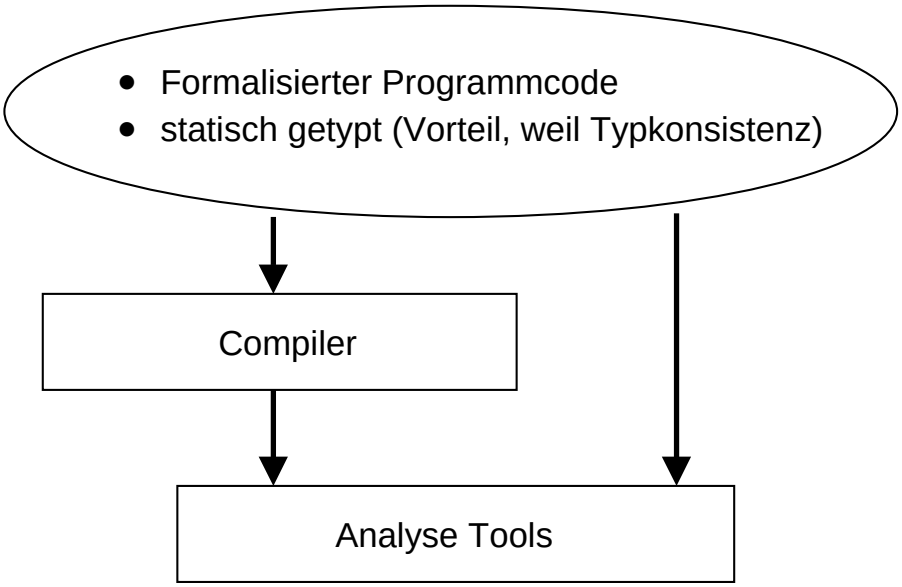
## 2. Statische Analyse

Die Statische Analyse ist eine Elementarmethode zur Analyse von Programmen oder Programmteilen. Sie beschäftigt sich mit einem Quelltext und versucht schematisch Informationen über das Programm zu ermitteln. Ziel ist es Fehler und Widersprüche aufzudecken oder Optimierungen zu ermöglichen.

Ihre Hauptanwendung findet die Statische Analyse innerhalb von modernen Compilern. Diese versuchen automatisch Programmfehler zu finden, um letztendlich die Compilierung und das Ergebnis zu optimieren. Darüber hinaus gibt es spezielle Analyse Tools, die den Programmcode noch eingehender analysieren und bewerten. Moderne Compiler gehen allerdings immer mehr dazu über, diese speziellen Tools zu integrieren und entsprechende Warnungen oder Fehler auszugeben.

### 2.1. Bedingungen für eine Statische Analyse

- Eine Statische Analyse auf formalisiertem Programmcode am effizientesten. (Code liegt im Quelltext einer bestimmten Sprache vor). Analyse von Binärdateien ist aber ebenfalls möglich[4].
- Besonders geeignet sind statisch getypte Programmiersprachen. Hier kann z. B. die Typkonsistenz nachgewiesen werden => Typ-Fehler-Freiheit
- Reicht die statische Analyse eines modernen Compilers aus, oder werden zusätzliche Werkzeuge benötigt?




## 2.2. Ziele der Statische Analyse – moderne Compiler

- Erkennung von syntaktischen Programmierfehlern.
- Überprüfung der Typisierung.
- Entfernung überflüssiger Programmteile
- Optimierung von Laufzeitverhalten und Speicherbedarf (zur Erklärung siehe [3])

  - Constant Folding and Propagation
  - Common Subexpression Elimination (CSE)
  - Loop Invariant Code Motion
  - ...

**Beispiel 2.2.1 (Turbo Pascal) – Mögliche Optimierungen**

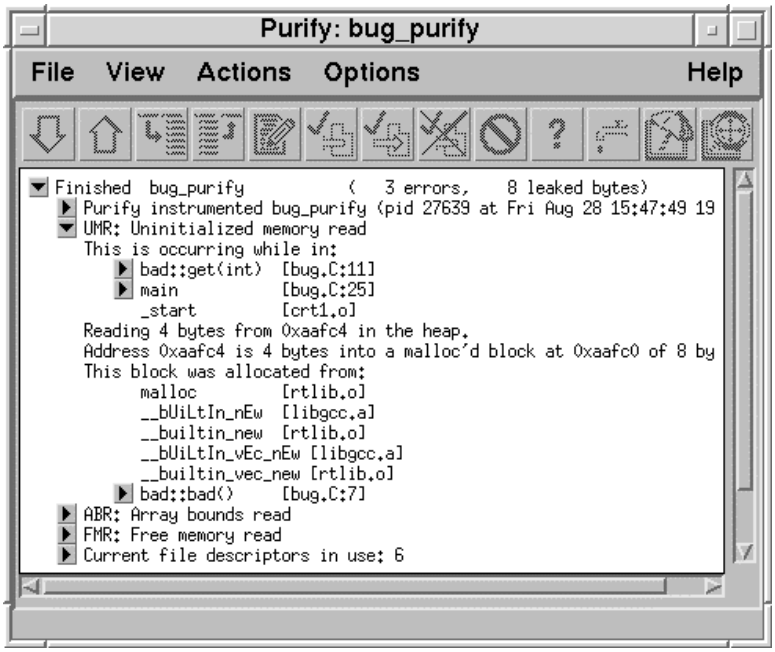
```
procedure ueberfluessig;
  begin
    writeln('Ich will auch was sagen!');
  end;

begin
  writeln(1+2+3+4);
end.
```

Der Compiler kann erkennen, dass die Funktion ueberfluessig niemals erreicht wird und bei der Comilierung dieses Codefragment überspringen. Ebenso kann der Ausdruck „1+2+3+4“ vorberechnet werden, da er nur aus Konstanten besteht.

## 2.3. Ziele der Statischen Analyse – Analyse Tools / manuelle Analyse

*Beispielprogramm zur Statischen Analyse (Purify): Quelle [5].*




- **Variablen (Datenfluß)**

  - Findet für alle benötigten Variablen eine Initialisierung statt?

*Beispiel 2.3.1 (Turbo Pascal) – Wert ist nicht initialisiert*

```
var i:integer;
begin
  writeln(3/i);
  writeln(i+1);
end.

Ausgabe:    1.52006485610053E-0004
            1
```

Der Wert in i kann nicht 0 gewesen sein, sonst hätte die Rechnung 3/i eine Div-by-Zero-Ausnahme erzeugt. Allerdings ist der Wert „fast“ Null denn i+1 ist 1. Manche Compiler (z. B. gcc 2.95) finden solche Probleme.

  - Ist die Dimension von Arrays gross genug?

*Beispiel 2.3.2 (C) – Out-Of-Bound-Access*

```
int a[10];
int f()
{
  int k;
  k = 10;
  return a[k];
}
```

Erst zur Laufzeit wird festgestellt, dass es im Array a[] keine Element a[10] gibt. Analyse Tools hätten solche fehler durch den Vergleich von Variablen-Inhalten finden können.

- Werden Überläufe abgefangen?
- Kann ein „Division by Zero“ auftreten?
- ...

- **Konstrukte (Kontrollfluß)**

  - Funktionieren alle Konstrukte einwandfrei? Sind z. B. Endlosschleifen, etc. möglich?
  - Werden bedenkliche Konstrukte bei der Programmierung verwendet? Z. B. Goto, ...
  - Ist der Code portabel?
  - Existiert redundanter/überflüssiger Code?

- **Unvollständigkeit, Abweichung oder Widersprüche zur Spezifikation**

Über die automatischen Überprüfungen hinaus, kann man mit manuellen Tests, Analysen und Auswertungen von z. B. Ablaufprotokollen weitere Fehler und Probleme aufdecken. Siehe dazu auch 3. Codereview.

*Beispiel 2.2.2 (Turbo Pascal) – Abweichung von der Spezifikation:*

**Spezifikation:**

```
Die Funktion add kann beliebige Zahlen addieren.
```

**Implementation:**

```
function add(a, b:integer):integer;
  begin
    add:=a+b;
  end;
```

**Analyse:**

```
add() verarbeitet nur Integer-Zahlen
```

## 2.4. Zusammenfassung

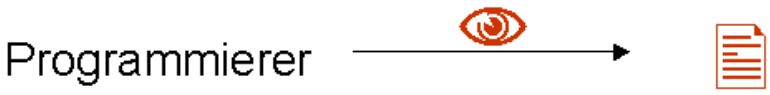
- Moderne Compiler leisten immer mehr.
- Umfangreiche Analysen und Empfehlungen liefern verschiedene Tools[4].
- Grundlegende Statische Analysen sind unumgänglich um funktionierenden Code zu erzeugen.

# 3. Code Review

Mit Code Review wird eine Technik bezeichnet, bei der fertige Programmteile erneut von Programmierern durchgesehen, analysiert und auf Fehler untersucht werden. Abhängig davon, wie ein Code Review abläuft, wird es unterschiedlich bezeichnet:

## 3.1. Walkthrough[2]

Bei einem Walkthrough prüft der Entwickler selbst seinen Code. Diese Analysetechnik findet meist automatisch auch parallel zur Programmierung, statt. Ein richtiger Walkthrough bedeutet jedoch, dass der Entwickler nach Fertigstellung eines Programmteils diesen Zeile für Zeile analysiert und verifiziert.




Vorteile:

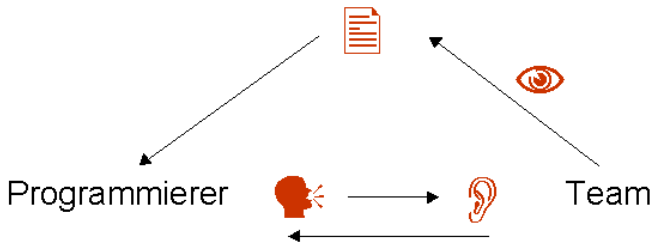
- Der Entwickler kennt die Spezifikation und das Programm und kann bei einem Walkthrough „von aussen“ Fehler finden. Es ist keine Einarbeitung in den Code erforderlich.
- Man-Power: Nur eine Person ist an diesem Code Review beteiligt.

Nachteil:

- Ein genereller Nachteil eines Code Reviews ist es, das er manuell durchgeführt werden muss. Längere Programme eignen sich daher nur bedingt für einen Code Review.

## 3.2. Code-Inspektion[2]

Diese Analyse läuft im Team ab. Der Entwickler stellt seinen Code vor und erklärt, wie er die Spezifikation implementiert und warum er bestimmte Techniken gewählt hat. Das Team bewertet und analysiert den Code und gibt Tipps und Verbesserungshinweise.




Vorteile:

- Das Team muss sich nicht selbst in die Spezifikation einarbeiten, bekommt aber dennoch ein Bild, wie dieser Programmteil abläuft: → Zeitersparnis / Vorteil wenn ein Entwickler ausfällt
- Das Team muss den Code nicht komplett verstehen, weil es jederzeit beim Programmierer nachfragen kann: → Zeitersparnis
- Viele Personen sehen und bewerten den Code → Wissensvereinigung → Fehlerverringerung
- Das Team lernt den Stil der anderen Programmierer kennen. Jeder Programmierer muss seinen Stil rechtfertigen können.

## 3.3. Zusammenfassung

Ein Code Review ist eine relativ einfache Analyse-Methode. Sie vereinigt das Wissen von allen Beteiligten, ohne, dass diese sich selbstständig in den Code einarbeiten müssen. Dazu ist es eine gute Möglichkeit, einen Programmierer zu kontrollieren und eine Bewertung seiner Arbeit zu bekommen. Nachteilig wirkt sich der Zeit- und Personenbedarf aus.

# 4. Dokumentation

Dieser Bereich untergliedert sich in mehrere Techniken und Arten von Dokumentationen. Beginnend beim Quelltext bis hin zum Handbuch gibt es je nach Zielgruppe verschiedene Anforderungen und Notwendigkeiten.

## 4.1. Quellcode-Dokumenation (Kommentierung)

Für jede Schnittstelle (Funktion/Methode) sollte dokumentiert sein, welche möglichen Eingaben sie verarbeiten kann und welche Ausgaben zurückkommen können. Auf diese Weise erleichtert man die Wiederverwendung in anderen Programmteilen und hilft Fehler zu vermeiden.

Bei Teamarbeit ist es grundsätzlich sinnvoll, sich ein einheitliches, gemeinsames Kommentierungskonzept zu überlegen. Grundsätzlich sollte man sich an folgendes halten:

- Kommentar zu Beginn einer Datei:
  *Was beinhaltet diese Datei?*

- Kommentar zu Beginn jeder Funktion:
  *Was macht diese Funktion?*

- Kommentar bei komplizierten Konstrukten und eigenwilligen Implementationen

*Beispiel 4.1.1 (Turbo Pascal) - Beispielkommentar*

```
(*Die Funktion add addiert zwei integer-Werte
  pre : a und b sind zwei beliebige Integer-Werte
        ihre Summe darf den Zahlenbereich
        -32768 ... 32767 nicht verlassen.
  post: die Summe a+b wird zurueckgegeben
*)
function add(a, b:integer):integer;
  begin
    test := a + b;
  end;
```

Pre steht für die Parameter, die der Funktion übergeben werden. Post für die Werte, die die Funktion wieder zurück gibt.

Für einige Programmiersprachen gibt es automatische Dokumentations-Tools. Beispielsweise Weave und Tangle von D. E. Knuth erzeugt für Pascal- oder C-Programme eine Dokumentation im Tex-Format. Für Java gibt es das Programm javadoc, welches speziell formatierte Quelltext-Kommentare zu einer externen HTML-Dokumentation verarbeiten kann.

*Beispiel 4.1.2 (Javadoc) – Java-Kommentar*

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *    Window win = new Window(parent);
 *    win.show();
 * </pre>
 *

 * @author  Christoph Probst
 * @see     java.awt.BaseWindow
 * @see     java.awt.Button
 */
class Window extends BaseWindow {
   ...
}
```

Schreibt man solche Kommentare in seinem Java-Quellcode, so kann javadoc automatisch eine HTML-Dokumentation erzeugen.

*Beispiel 4.1.3 (Javadoc) – Automatisch erstellte HTML-Dokumentation:*

---

class **Window**
A class representing a window on the screen.
For example:

```
Window win = new Window(parent);
win.show();
```

**Author**: Christoph Probst

**See Also:** BaseWindow, Button

---

Diese HTML-Dokumentatinon erzeugt javadoc automatisch aus den Kommentaren aus Beispiel 4.1.2

Im Internet finden sich auch verschiedene Richtlinien sowie Tipps und Hinweise zum richtigen Kommentieren. Siehe z. B. Technisch Physikalische Bundesanstalt[6].

## 4.2. Programm-Dokumenation (Spezifikation)

Eine Programm-Dokumentation beschreibt die internen Abläufe, Strukturen und Datenformate eines Programms.

- Funktionalität und Fehler
- Architektur des Programms
- Aufruf- und Klassenhierachie, Jackson-Diagramme
- Datenformate/Datenfluß, Interne Formate, Schnittstellen (E/A-Formate)
- Platformabhängigkeiten

*Beispiel 4.2.1 – Programmdokumentation [6]*

**1. Programmkenndaten**
**1.1. Programmidentifizierung**
**1.1.1. Programmname**
**Programm HOEPPLER. Die Quelltexte umfassen die Files HOEPPLER.C und GRFDEF.H.**

**1.1.4. Aktuelle Programmversion**
**Die Dokumentation basiert auf der Version 1.1 vom 9. März 1995.**
**[...]**
**1.8. Dokumentationsumfang**
**Die Dokumentation zum Programm HOEPPLER besteht aus der hier vorliegenden Programmdokumentation.**

**1.9. Zuständigkeiten**
**Bemerkungen, Hinweise und Vorschläge können an den Autor des Programms, Herrn Dr. N. Böse, Labor 3.12 gerichtet werden.**

**2. Programmfunktion**
**2.1. Aufgabenstellung**
**2.1.1. Aufgabenbeschreibung**
**[...]**

## 4.3. Anwender-Dokumentation / Handbuch / Manual

Die Anwender-Dokumentation bildet ebenfalls einen der wichtigsten Teile der Dokumentation. Letztendlich hängt es vom Benutzer ab, ob die Software auch richtig angewendet wird. Wichtig ist es, diese Dokumentation für die Zielgruppe und nicht für die Entwickler zu schreiben. Eine zu technische Bedienungsanleitung ist für den End-Anwender u. U. nutzlos.

Aus diesen Gründen wird die Anwender.Dokumentation häufig von professonellen Dokumentationsschreibern verfasst. Diese haben die entsprechende Erfahrung und Fähigkeiten, eine anwendergerechte Dokumentation zu erstellen.

*Beispiel: In einem Artikel der Zeitschrift mot (8/97) wird es dieser Umstand relativ treffend beschrieben:*

Juristen mißbrauchen [die Bedienungsanleitung des Autos] allzugern für ängstliche Absicherungen. Dann strotzt das Büchlein wie bei Suzuki und Subaru nur so von Warnhinweisen. Und wenn sich Ingenieure als Schriftsteller versuchen, wird, wie bei Opel, die Motorleistung nur noch in Kilowatt angegeben. Das aber geht am Verständnis der meisten Menschen vorbei.

*Beispiel: Aus der Bedienungsanleitung eines Reglers - Ladung der Sollwerte*

```
Bei der Erarbeitung der Programmierungsfolge wurde darauf
geachtet, jeden möglichen Einstellfehler seitens der
Bedienungsperson zu vermeiden und in die Programmierungsseiten
sind Kontrollen der einstellbaren Werte vorgestanzt. Es könnte
trotzdem vorkommen, daß die Bedienungsperson von vorgegebenen
Bedingungen und insbesondere von untereinander kongruenten, wenn
```

```
auch nicht unbedingt für die jeweilige den Regler enthaltene
Anlage optimalen Daten wieder ausgehen muß. Zu diesem Zweck ist
eine besondere durch die simultane Betätigung von drei Tasten
aktivierbare Funktion vorgesehen, die die Rücksetzung aller
Daten gemäß den im Werk eingespeicherten Werten durchführt.
```

Eines Anderdokumentation sollte auf jeden Fall beinhalten:

- Erläuterungen zu Parametern und der Konfiguration des Programms
- Funktionsweise und Ablauf des Programms
- Hilfestellung bei Problemen

# 5. Beispiel-Problem[8]

Es geht um eine Fax-Software, die selbststaendig eine Modemverbindung aufbaut, um Faxe zu verschicken.

## Software-Version A: - unselbständig

Benutzer gibt einfach die Empfänger-Rufnummer an:

„1-408-xxx-xxxx“
1-<Vorwahl>-<Rufnummer>

## Software-Version B: - etwas intelligenter

- Software versucht Vorwahl zu erkennen
- Hängt bei fremden Vorwahlen „1“ vor die Rufnummer
- Kann selbständig Telefonanlagen verlassen.

| | | |
|---|---|---|
| | 1-408-xxx-xxxx | Automatische Erkennung der Vorwahl |
| 1- | 1-408-xxx-xxxx | Anfügung einer 1 damit 140 eine richtige Vorwahl wird. Allerdings stand diese 1 ja schon längst davor. |
| 9- 1- | 1-408-xxx-xxxx | Weil weiterhin kein Verbindungsaufbau möglich war: Anfügung einer 9 (zum Verlassen der Telefonanlage). |

Etwa 5 Minuten später traf die Polizei ein.

## Analyse des Beispiels

Wären folgende Punkte beachtet worden, hätte man den Softwarefehler vermutlich verhindern können. Meist scheitern größere Analysen allerdings an der Notwendigkeit der Fehlerfreiheit bzw. am Entwicklungsdruck. Besonders kleinere Firmen können es sich kaum Leisten, langfristige Test durchzuführen.

| | | |
|---|---|---|
| Typisierung | - | Unterschiedliche Variablen für Vorwahl und Rufnummer |
| | - | Regeln für Vorwahl und Rufnummer implementieren („Typ-Spezifikation“) |
| Code Review | - | Standard-Situation: „Rufnummer beginnt bereits mit 1“ |
| | - | Keine Interaktion mit dem Benutzer |
| Dokumentation | - | Warnhinweis bei wesentlichen Änderungen |

# Quellen und weiterführende Literatur


**Typisierung:**

[1] Andreas Bogk / Einführung in Dylan / http://www.ccc.de/events/congress98/doku/281298-dylan.text

- Heiner Grill (1997) / Das Typkonzept
  http://file.inf.bi.ruhr-uni-bochum.de/~heinerg/diplom/diplom_arbeit/node3.html
- Mark Fink (1998) / Konzeption und Implementation eines Funktionstabellen-Editors
  http://www.e-applications.de/dok/diplomarbeit.pdf

[7] Prof. Schönefeld (1996) / Polymorphismusprinzip /
http://remus.prakinf.tu-ilmenau.de/wetter/swt/bkoop/kap4.html

**Statische Analyse:**

- FH Fulda / Glossar / http://www.fh-fulda.de/medoc/konvert/vdi/ad2.htm#Statische_Analyse0
- FU Berlin / Seminarbeschreibung Statische Analyse / http://www.inf.fu-berlin.de/lehre/SS98/statana/

[2] Softwarepraktikum Informatikstechnik und Logistik; Testen von Programmen /
http://stl-www.cs.uni-dortmund.de/Lehre/Sopra/Testen.pdf

[3] Matthias Ernst, Daniel Schneider (2001) / Konzepte und Implementierungen moderner virtueller Maschinen; Nebenläufigkeit, automatische Speicherverwaltung und dynamische, optimierende Compiler für Java / http://www.virtualmachine.de/2000-ernst-schneider/node91.html

[4] Tools für die Statische Analyse:
- Physikalisch- Technische Bundesanstalt / Softwaretechnische Prüfungen; Grundlagen + Hilfsmittel /
  http://www.berlin.ptb.de/8/83/831/swq/ansqs/j_tools8302.html
- Purify (Rational Software)
- jtest (Parasoft)

[5] Beispielbild Purify / http://www.cs.wisc.edu/~lederman/purify/umr.gif

**Code Review**

[3] Matthias Ernst, Daniel Schneider (2001) / Konzepte und Implementierungen moderner virtueller Maschinen; Nebenläufigkeit, automatische Speicherverwaltung und dynamische, optimierende Compiler für Java / http://www.virtualmachine.de/2000-ernst-schneider/node91.html

- Iterative Software Ltd.; Professional Code Review / http://www.iterative-software.com/codereview.html
- Adam Shostack (2000) / Security Code Review Guidelines / http://www.homeport.org/~adam/review.html

**Dokumentation**

- Adam Shostack (2000) / Security Code Review Guidelines / http://www.homeport.org/~adam/review.html

[6] Richtlinien für die Softwareentwicklung / Physikalisch- Technische Bundesanstalt /
http://www.berlin.ptb.de/8/83/831/swq/ansqs/j_rili.html

**Beispiel**

http://ciac.llnl.gov/news/RISKS/9612/msg00002.html