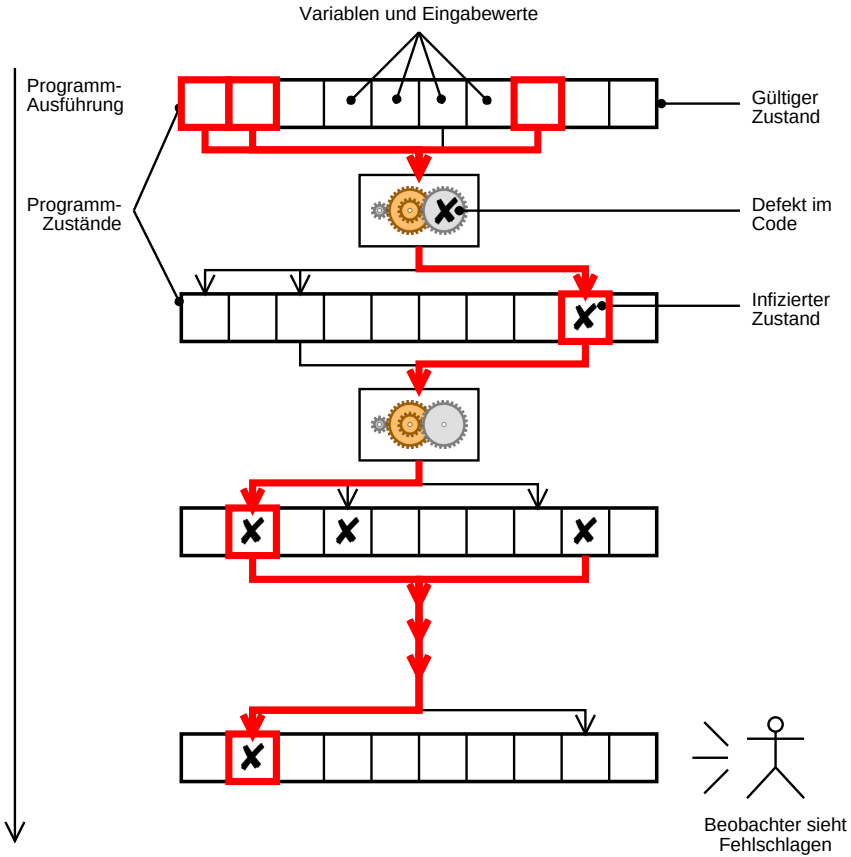
Programmverstehen + Fehlersuche

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Wie ein Fehlverhalten entsteht



Wie ein Fehlverhalten entsteht (2)

Grundlegender Ablauf:

- Ein Codestück ist *fehlerhaft* („der Fehler“) (✘)
- Wird es ausgeführt, *infiziert* es den Programmzustand.
- Die Infektion breitet sich aus und wird als *Fehlverhalten* sichtbar.

Leider ist es nicht so, dass fehlerhafter Code stets eine Infektion oder ein Fehlverhalten bewirkt.
⇒ *Kernproblem des Testens!*

Aber: Jedes Fehlverhalten geht auf eine Infektion zurück und jede Infektion wird von einem fehlerhafter Code verursacht.
⇒ *Grundansatz der Fehlersuche!*

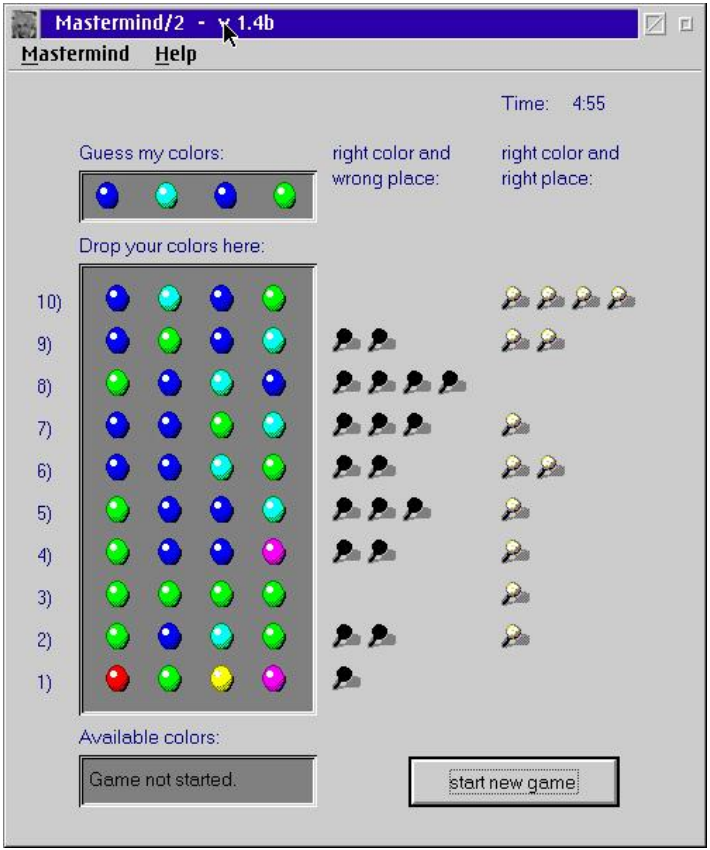
Die Wissenschaftliche Methode

Wissenschaftliche Methode: Allgemeines Muster zum Finden einer Theorie, die einen Aspekt des Universums erklärt und voraussagt.

1. Beobachte einen Aspekt des Universums.
2. Erfinde eine erste Beschreibung, genannt *Hypothese,* die mit den Beobachtungen übereinstimmt.
3. Mache Vorhersagen aufgrund der Hypothese.
4. Teste diese Vorhersagen durch Experimente oder weitere Beobachtungen und verfeinere die Hypothese.
5. Wiederhole Schritte 3 und 4, bis Hypothese und Experiment/Beobachtung übereinstimmen.

Schliesslich wird die Hypothese so zur *Theorie*.

Ein Mastermind-Spiel



Fehlersuche mit Buchführung

Grundidee: *Fehlersuche explizit machen!*

Schreiben Sie auf

1. wie sich das Problem *äussert,*
2. welche *Hypothesen* in Frage kommen
3. welche *Experimenten* die Hypothesen geprüft werden,
4. *Vorhersagen*, was die Experimente angeht
5. *Beobachtungen* aus den Experimenten, und
6. *Schlüsse* aus den Experimenten.

Ohne Buchführung ist Fehlersuche wie *Blindes Mastermind:*
Man muss alles im Kopf behalten – und kann niemals aufhören.

Beispiel Buchführung

Wie äußert sich das Problem?

a = 0 wird ausgegeben, aber a darf nicht Null sein.

Hypothese 1 *Variable* a *ist 0; daher wird* a = 0 *ausgegeben.*

Experiment Setze a auf 1.

Vorhersage a = 1 wird ausgegeben.

Beobachtung a = 0 wird ausgegeben.

Schluss Hypothese 1 wird *verworfen.*

Hypothese 2 *Das Format* "%d" *ist Grund der Ausgabe* a = 0.

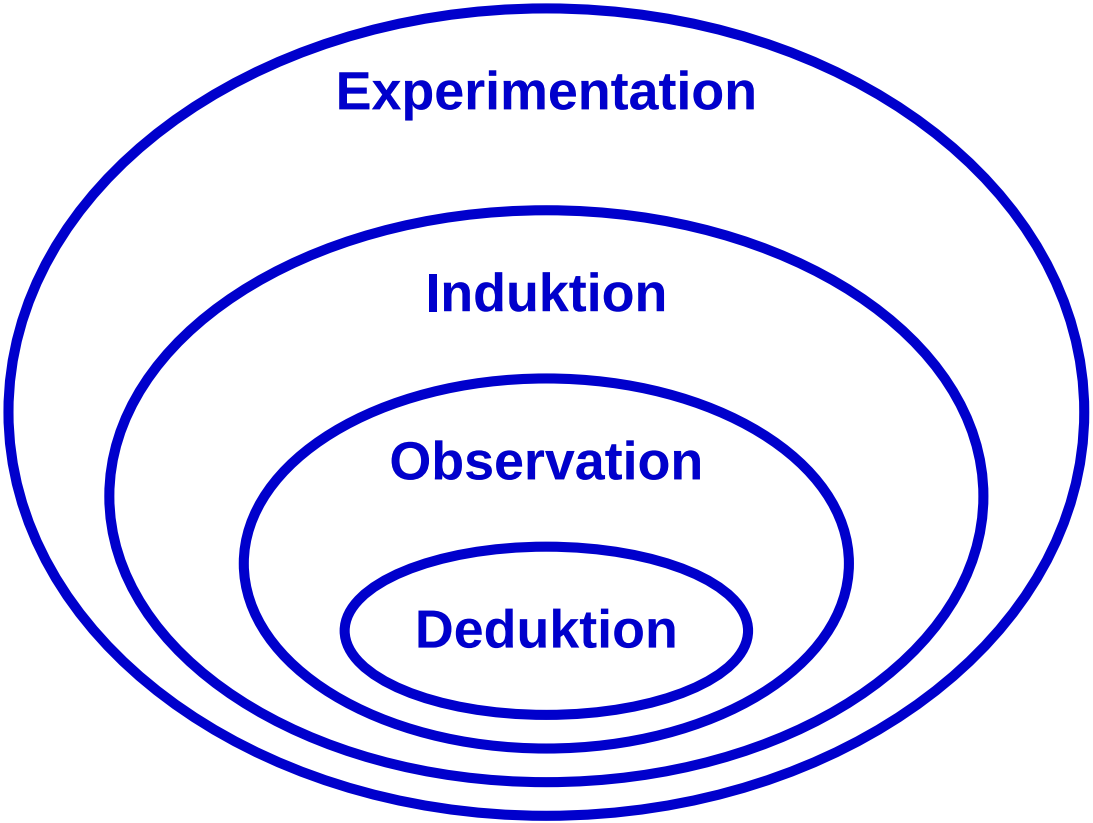
Experiment Ändere das Format auf "%f".

Vorhersage a = ⟨*ein Wert ungleich 0*⟩ wird ausgegeben.

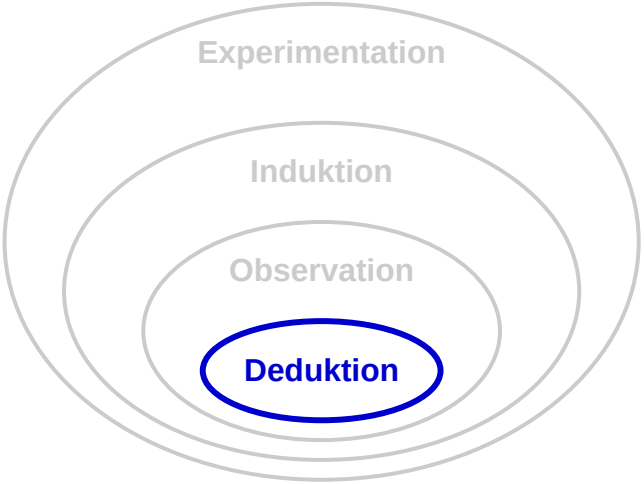
Beobachtung (wie vorhergesagt)

Schluss Hypothese 2 wird *bestätigt.*

Techniken des Programmverstehens



Deduktion



Deduktion: Schließen vom *allgemeinen* auf das *besondere*

- Hier: Schliessen vom Programmcode auf Programmläufe
- Keine Ausführung nötig
- Schlüsse gelten für *alle* Läufe und *alle* Umgebungen

Beispiele: Syntax · Semantik · Datenfluss · Kontrollfluss

Deduktionstechniken

Syntaktische Korrektheit. Der Programmtext entspricht der Syntax der Programmiersprache.

Semantische Korrektheit. Beziehungen zwischen Definition und Benutzung stimmen (etwa: jede Variable ist deklariert).

Typkorrektheit. Jeder Variablenwert entspricht ihrem *Typ*.

Korrekter Datenfluss. Etwa: Jede Variable

- muss vor dem Lesen initialisiert sein.
- muss vor dem Schreiben gelesen werden können.

Korrekter Kontrollfluss. Etwa: Jedes Programmstück muss ausgeführt werden können.

Dies erledigt gewöhnlich der Compiler (z.B. `gcc -Wall`)







Visualisierung

Um das Verständnis des Programms zu vereinfachen, helfen *Visualisierungen*

- der Klassenstruktur
- des Datenflusses
- des Kontrollflusses

Solche Visualisierungen sind entweder

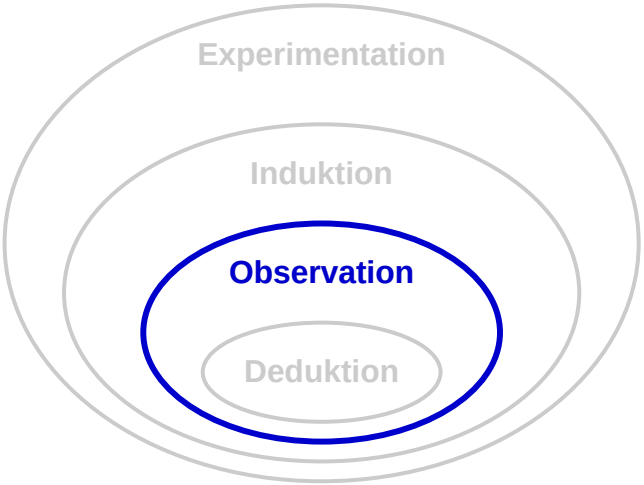
- Teil des Entwurfs (besser!) oder
- müssen aus dem Programmcode generiert werden.







Observation



Observation: Bestimmen von *Fakten*

- Beobachten eines einzelnen Programmlaufes
- Feststellen von *Fakten* über den beobachteten Lauf
- Fakten gelten nur für beobachteten Lauf
- Deduktion kann benutzt werden

Beispiele: Ausgabe-Funktionen · Debugger · Zusicherungen

Ausgabe-Funktionen

Ziel: *Protokollieren des Programmzustands*

```
int main(int argc, char *argv[])
{
    int number_of_primes;
    number_of_primes = atoi(argv[1]);
    printf("main(): number_of_primes = %d\n",
        number_of_primes);
    print_primes(number_of_primes);
    printf("main(): returning\n")
}
```

Ausgabe-Funktionen sind die *einfachste* and *verbreitetste* Technik zur Fehlersuche.







Dezidierte Ausgabe-Funktionen

Wir benutzen

```
LOG(("number_of_primes = %d", number_of_primes))
```

und erhalten

```
main.c:3: number_of_primes = 3
```

Definition:

```
#define LOG(args) \
    printf("%s:%d: ", __FILE__, __LINE__), \
    printf args, \
    printf("\n")
```

Dieses Schema kann leicht erweitert werden, um Datum/Zeit usw. auszugeben.

Ausgabe mit Aspekten

Aspekte sind Code-Stücke, die automatisch in Programme „eingewebt“ werden – etwa zu Beginn jeder Methode.

Aspekte ermöglichen besonders elegante Ausgabe-Funktionen:

```
public aspect Tracer {
   pointcut allMethods():
       call(public * Article.*(..));
   before(): allMethods() {
       System.out.println ("Entering " +
                           thisJoinPoint);
   }
   after(): allMethods() {
       System.out.println ("Leaving "  +
                           thisJoinPoint);
   }
}
```







Debugger-Funktionen

Ein *Debugger* ermöglicht es,

- das Programm zu *starten* (unter Angabe aller Eingabe-Parameter)
- das Programm unter bestimmten Bedingungen *anzuhalten*.
- den *Programmzustand* des angehaltenen Programms zu prüfen.
- den Programmzustand zu *ändern*, um Ursachen und Wirkungen zu studieren.

Quelle: gdb(1) manual page







Ausführung untersuchen

Wie wissen wir, welche Programmteile ausgeführt wurden?

Ein *Haltepunkt (breakpoint)* lässt das Programm an einer bestimmten Stelle anhalten.

```
$ gdb sample
(gdb) break main
Breakpoint 1 in main
(gdb) _
```

Das Programm hält an, sobald `main` erreicht ist
(Formal: Der program counter (PC) ist `main`)







Haltepunkte und Beobachtungspunkte

Einige Debugger stellen weitere Haltebedingungen zur Verfügung – insbesondere Bedingungen über *Daten*

Ein *Beobachtungspunkt (watchpoint)* in GDB hält das Programm an, sobald eine bestimmte Variable ihren Wert ändert:

```
(gdb) watch a
Hardware watchpoint 1: a
(gdb) continue
Old value = (int *) 0xbffff518
New value = (int *) 0x8049850
(gdb) _
```







Beobachtungspunkte im Detail

Beobachtungspunkte können beliebig komplex sein:

(gdb) **`watch f(x) != 42`**

hält an, sobald `f(x)` seinen Wert ändert

Beobachtungspunkte können Haltepunkte simulieren:

(gdb) **`watch $pc != main`**

hält an, sobald der program counter die `main`-Funktion erreicht.

Beobachten von komplexen Ausdrücken ist *teuer* – das Programm läuft wesentlich langsamer.

Beobachten von einfachen Variablen ist billig, sofern vom Prozessor unterstützt.







Bedingte Haltepunkte

Bedingte Haltepunkte ermöglichen es, Bedingungen nur an bestimmten Stellen zu prufen – d.h. wenn der PC einen bestimmten Wert erreicht hat.

```
(gdb) break print_primes if n_primes == 2
Breakpoint 1 at print_primes
(gdb) _
```

Das Programm hält an, wenn
der PC `print_primes` erreicht hat und `n_primes` den Wert 2 hat.

Dies kann billig implementiert werden.







Haltepunkte und Bedingungen

Übersicht:

Typ	GDB-Kommando	Bedingung
Haltepunkt	`break` *location*	$PC = location$
Beobachtungsp.	`watch` *expr*	*expr* changes
Bed. Haltep.	`break` *location* `if` *expr*	$PC = location \wedge expr$

Der Debugger hält das Programm auch *automatisch* an

- bei Unterbrechung durch den Anwender (Ctrl+C)
- wenn es ein fatales Signal empfängt
- wenn eine Ausnahme (exception) nicht behandelt wird

Den Stack untersuchen

Hält ein Programm an, gehört das *Untersuchen des Stacks* (des Stapels der aufrufenden Funktionen) zu den ersten Tätigkeiten.

```
(gdb) run
Starting program: sample

Breakpoint 1, shell_sort (a=0x8049850, size=1)
    at sample.c:9
9           int h = 1;
(gdb) where
#0  shell_sort (a=0x8049850, size=1) at sample.c:9
#1  main (argc=1, argv=0xbffff564) at sample.c:35
#2  __libc_start_main () from /lib/libc.so.6
(gdb) _
```

Programmzustand untersuchen

Hält ein Programm an, können wir seinen Zustand untersuchen – den Zustand des angehaltenen Programms.

Alle Debugger können einzelne Variablen ausgeben:

```
(gdb) print a[0]
$1 = 0
(gdb)
```

Die meisten Debugger unterstützen auch *Ausdrücke:*

```
(gdb) print a[size - 1]
$2 = 0
(gdb)
```







Programmzustand untersuchen (2)

Einige Debuggers unterstützen auch *Funktionsaufrufe:*

```
(gdb) print main(argc, argv)
$3 = 0
(gdb)
```

Auch Aufrufe von Methoden sind möglich:

```
(gdb) print c1.operator==(c2)
$4 = false
(gdb)
```

Hält die Ausführung während eines solchen Aufrufs an, können interessante Dinge geschehen :-)







Programmzustand untersuchen (3)

Um die Variablen einer aufrufenden Funktion zu untersuchen, kann man durch den Stack navigieren:

```
(gdb) frame
#0  shell_sort (a=0x8049850, size=4) at sample.c:9
(gdb) info locals
i = 1073834752
j = 1074077312
h = 1961
(gdb) up
#1  0x8048647 in main (argc=4, argv=0xbffff544)
    at sample.c:35
(gdb) info locals
a = (int *) 0x8049850
i = 3
(gdb)
```

Ausführung fortsetzen

Ist man mit dem aktuellen Programmzustand fertig, kann man die *Ausführung fortsetzen* (bis die nächste Haltebedingung erfüllt ist):

```
(gdb) continue
Program exited normally.
(gdb) _
```

Hoppla – wir hätten wohl noch einen Haltepunkt setzen sollen.







Schrittweise Ausführung

Häufig möchte man das Programm nur ausführen, *bis die nächste Anweisung erreicht ist:*

```
(gdb) run 7 8 9
Breakpoint 1, shell_sort (a=0x8049850, size=4)
    at sample.c:9
9           int h = 1;
(gdb) step
11              h = h * 3 + 1;
(gdb) step
12          } while (h <= size);
(gdb) _
```







Schrittweise Ausführung (2)

Zur schrittweisen Ausführung gibt es viele Varianten – je nachdem, was mit „Schritt“ gemeint ist.

`step` PC erreicht nächste ausgeführte Anweisung, möglicherweise in anderer Funktion

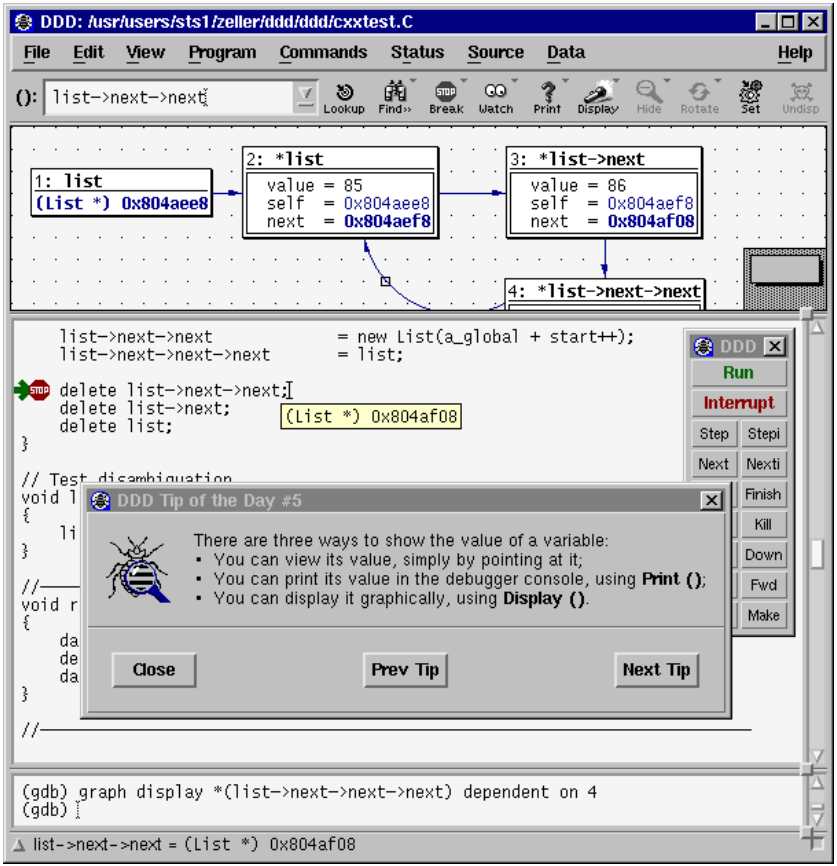
`next` PC erreicht nächste ausgeführte Anweisung in aktueller Funktion, oder Funktion kehrt zurück

`until` PC erreicht Programmzeile hinter der aktuellen, oder Funktion kehrt zurück

`finish` Funktion kehrt zurück

`continue` Unbedingte Fortsetzung

DDD—Graphische Benutzeroberfläche



Flüchtige Beobachtung

Alle Observations-Techniken können allgemein eingesetzt werden, um den Ablauf eines Programms zu *beobachten*.

Bei der Fehlersuche *vergleicht* jedoch der Programmierer den *vorgefundenen Wert* mit dem *erwarteten Wert.*

Ausgabefunktionen und Debugger-Sitzungen sind *flüchtig:*

- alle Vergleiche sind implizit (finden nur im Kopf statt)
- am Ende der Sitzung ist alles vorbei

Alternative: *Zusicherungen*







Zusicherungen

Eine Zusicherung *(assertion)* prüft zur Laufzeit, ob die Erwartungen (des Programmierers) erfüllt sind.

Generell gilt:

- Zusicherungen steigern die Vertrauenswürdigkeit meines Programms
- Zusicherungen helfen, Fehlerquellen schnell einzugrenzen
- Heute eingeführte Zusicherungen helfen auch morgen noch







Vor- und Nachbedingungen

Ziel: Sicherstellen, dass eine Funktion das richtige tut

```
#include <assert.h>

void divide(int dividend, int divisor,
            int& quotient, int& remainder)
{
    assert(divisor != 0);
    // Actual computation
    ...
    assert(quotient * divisor + remainder == dividend);
}
```

Klassen-Invarianten

Ziel: Integrität des Programmzustands sicherstellen

```
 class Game {
    int n_flowers;
    Map<string, int> flower_values;
    // z.B. flower_values["rose"] == 500
    ...
public:
    bool OK() {
        assert(n_flowers >= 1);
        assert(n_flowers == flower_values.size());
        return true;
    }
}
```

Kombinierte Zusicherungen

Eine Klassen-Invariante muss vor und nach jeder öffentlichen Methode gelten:

```
void Game::add_flower(string name, int value)
{
    assert(OK());  // Vorbedingung

    // Berechnung findet hier statt...

    assert(OK());  // Nachbedingung
}
```







System-Zusicherungen

In C- und C++-Programmen sind *Fehler im Freispeicher (Heap)* häufige Fehlerquellen:

- Benutzung von alloziertem Speicher, der wieder freigegeben wurde
- Mehrfaches Freigeben allozierten Speichers...

Die GNU C Laufzeit-Bibliothek bieter *Zusicherungen*, die die Integrität des Freispeichers sicherstellen

```
$ MALLOC_CHECK_=2 myprogram myargs
free() called on area that was already free'd()
Aborted (core dumped)
$ _
```







Feld-Zusicherungen

In C- und C++-Programmen sind *Feldüberläufe* weitere häufige Fehlerquellen.

Die *electric fence library* findet solche Überläufe automatisch:

```
$ gcc -g -o sample-with-efence sample.c -lefence
$ sample-with-efence -11 14
Electric Fence 2.1
Segmentation fault (core dumped)
$ _
```







Allgemeine Speicher-Zusicherungen

Das *Valgrind*-Werkzeug prüft *alle* Speicherzugriffe sowie alle Aufrufe zu malloc/new/free/delete.

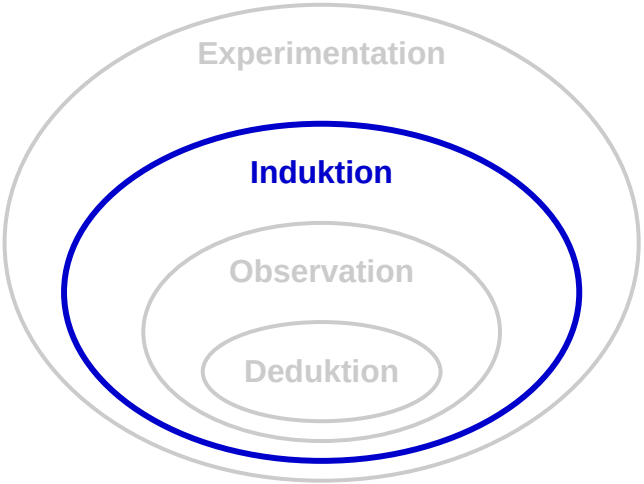
- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks – allocated area is not free'd
- Passing of uninitialised and/or unaddressible memory to system calls
- Mismatched use of malloc/new vs free/delete
- Some misuses of the POSIX pthreads API







Induktion



Induktion: Schließen vom Besonderen auf eine Abstraktion

- Beobachten *mehrerer Läufe*
- Bestimmen von *Gemeinsamkeiten* und *Anomalien*
- Schlüsse gelten für alle beobachteten Läufe
- basiert auf Observation; kann Deduktion benutzen

Beispiele: Vergleich der Abdeckung · Dynamisches Bestimmen von Invarianten

Vergleich der Abdeckung

Grundidee: Unterscheide

- Code, der nur in fehlschlagenden Läufen ausgeführt wurde
  ⇒ „Verdächtiger Code“
- Code, der in fehlschlagenden und funktionierenden Läufen ausgeführt wurde
  ⇒ „Durchwachsener Code“
- Code, der nur in funktionierenden Läufen ausgeführt wurde
  ⇒ „Wahrscheinlich korrekter Code“

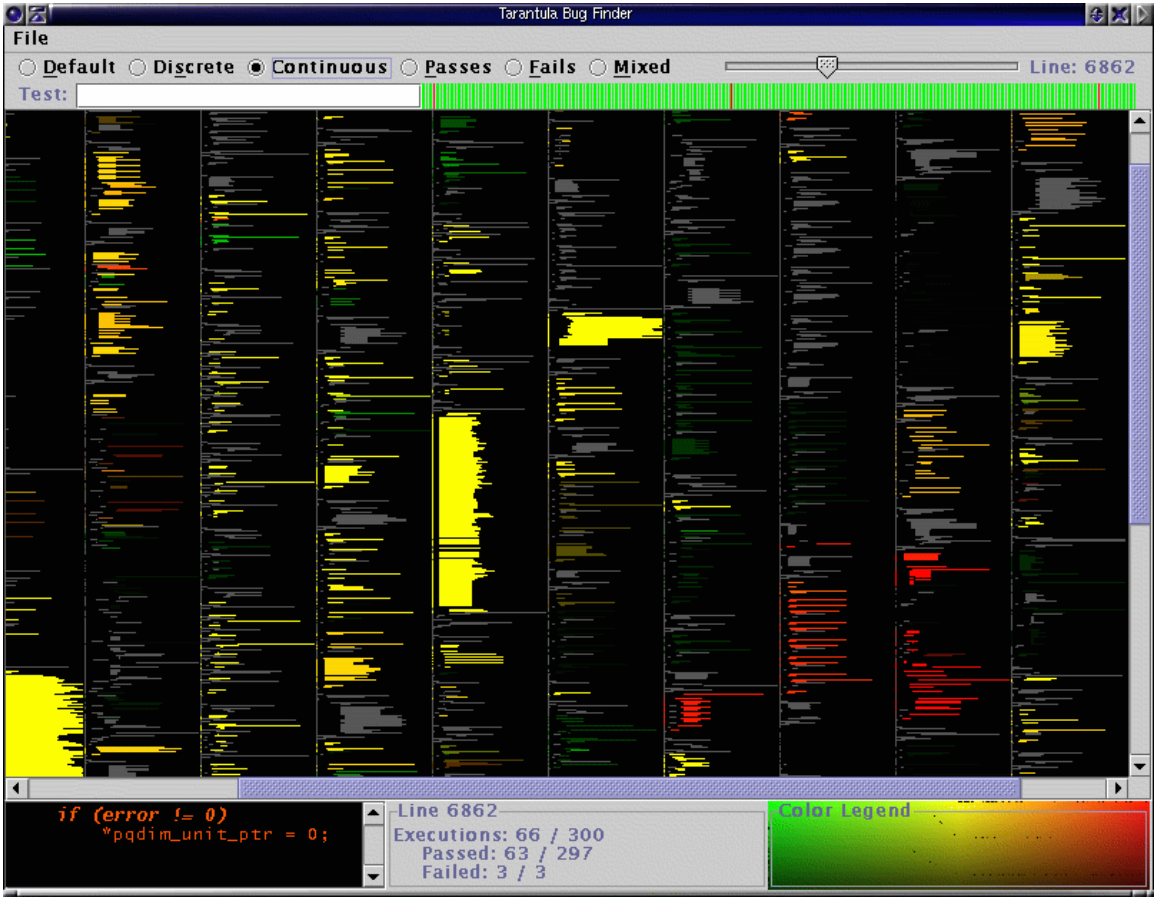
Zur Bestimmung der Abdeckung: vgl. *Testgütemaße*







Vergleich mit TARANTULA



Invarianten bestimmen

Grundidee: *Aus konkreten Läufen Invarianten bestimmen*

Invarianten beschreiben, was über *alle* beobachteten Läufe gilt.

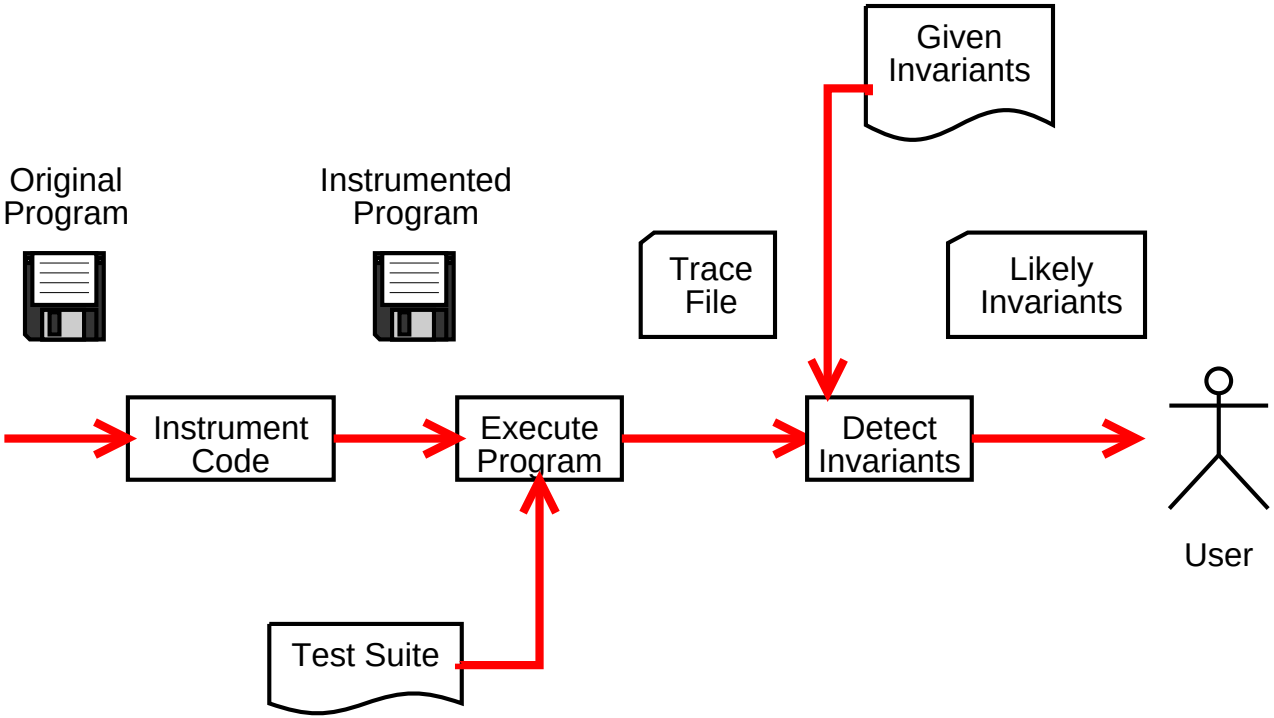
Werden nur funktionierende Läufe betrachtet, kann man die so gewonnenen Invarianten benutzen, um ggf. Verletzungen in fehlschlagenden Läufen zu bestimmen.







Invarianten bestimmen mit DAIKON



Invarianten bestimmen mit DAIKON (2)

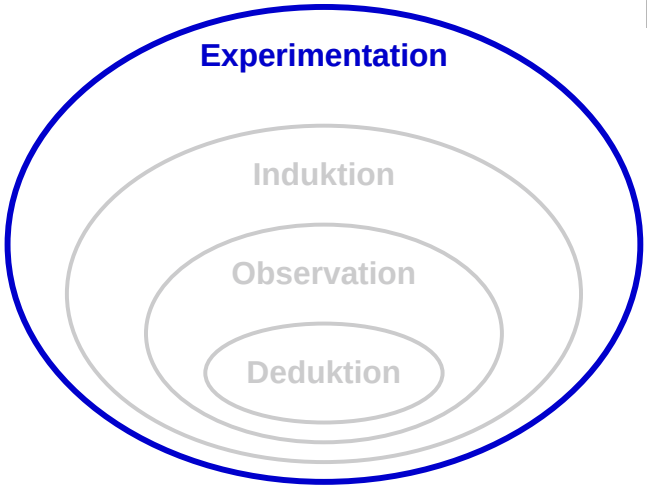

```
$ ./sort 1 2 3
1 2 3
$ ./sort 4 5 6
4 5 6
```

DAIKON erkennt folgende Invarianten:

```
std.shell_sort(int *;int;):::ENTER
size == size(a[])
a[] one of  [1, 2, 3], [4, 5, 6]
size == 3

std.shell_sort(int *;int;):::EXIT1
a[] == orig(a[])
```

Experimentation



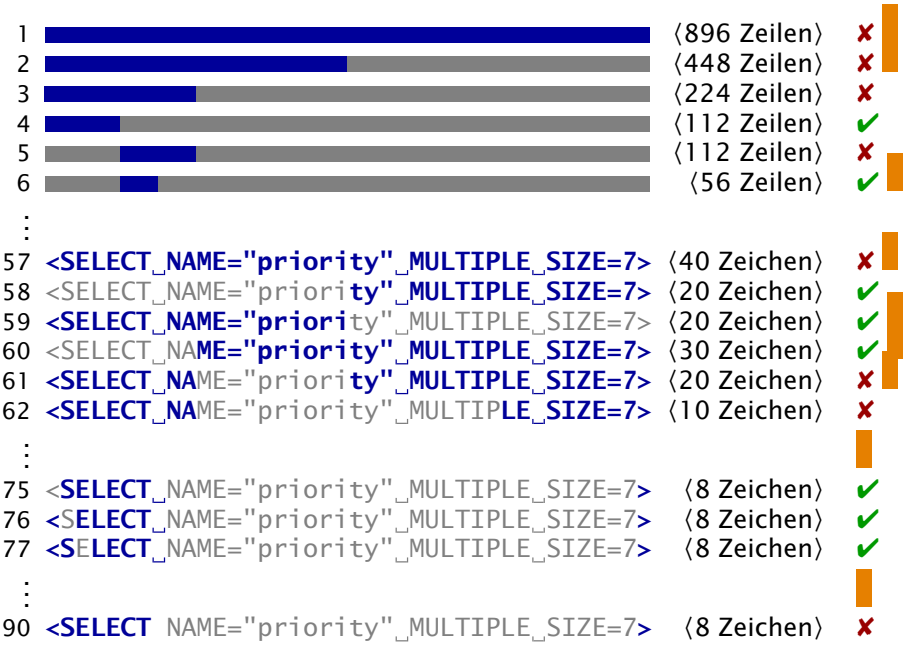
Experimentation: Experimente ausführen, basierend auf früheren Schlüssen

- Gezieltes Ausführen von Programmen
- isoliert *Fehlerursachen*
- basiert auf Observation; kann Deduktion und Induktion nutzen

Beispiele: Experimente durch Menschen (oder Automaten)

Beispiel: Fehlerverursachende Eingaben

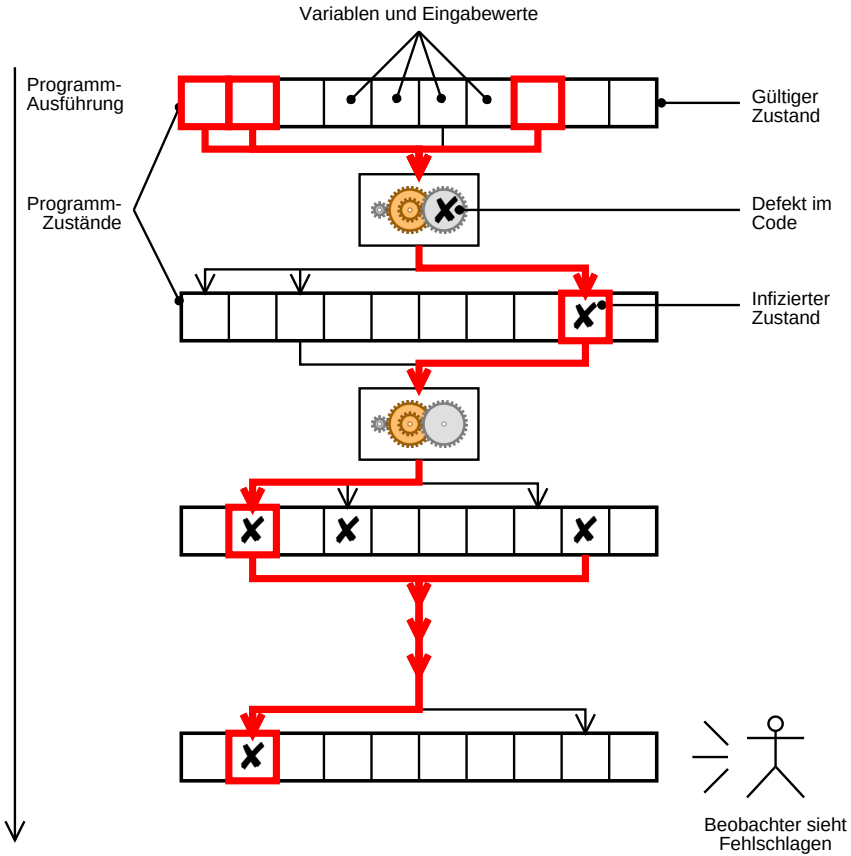
Mozilla stürzt beim Drucken einer Webseite ab.



Vereinfachte Fehlerbeschreibung:

`Mozilla stürzt ab, wenn ich <SELECT> drucke.`

Von der Eingabe zum Code



Von der Eingabe zum Code (2)

Systematische Fehlersuche isoliert Stück für Stück die Ursache-Wirkungs-Kette:

1. Finden der *fehlerverursachenden Eingabe* (durch systematische Experimente)
2. Finden des *fehlerverursachenden Zustands* (durch Beobachtung und Vergleich)
3. Finden des *fehlerverursachenden Codes* (durch Deduktion)

Änderungen am Code sollten erst erfolgen, wenn alle Einzelheiten der Ursache-Wirkungs-Kette bekannt sind.

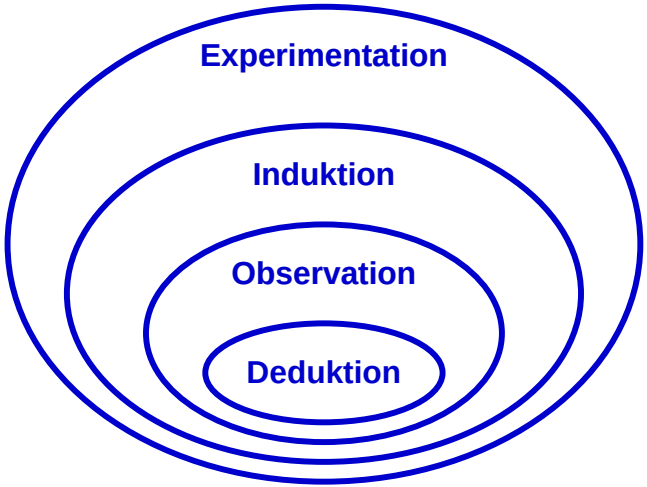
Ziel: *Code genau einmal ändern – zur Reparatur!*







Zusammenfassung



- Die Wissenschaftliche Methode setzt verschiedene Techniken zum Programmverstehen ein
- Jede Technik hat spezifische Stärken und Schwächen, die in der grundlegenden Schlusstechnik begründet sind

Gute Programmierer nutzen *all diese Techniken* zum Programmverstehen – und zur Fehlersuche!