Test und Fehlersuche

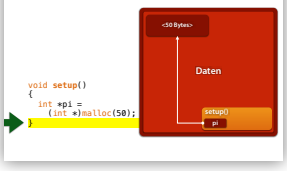
Programmieren für Ingenieure
Sommer 2014

Andreas Zeller, Universität des Saarlandes

Ihr Projekt

- **Bis 27. Juni**: *Projektskizze* abgeben
- Eine A4-Seite mit
  - Was soll gemacht werden?
  - Warum ist das originell?
  - Warum ist das schwer?
- Feedback bis 1. Juli

Freispeicher



Freispeicher

1. Du sollst nicht zu viel Speicher anfordern!
2. Du sollst nicht zu wenig Speicher anfordern!
3. Du sollst angeforderten Speicher wieder freigeben!
4. Niemals sollst Du auf freigegebenen Speicher zugreifen!
5. Du sollst Speicher nicht doppelt freigeben!

Verbünde

- In C können einzelne Daten zu einem *Verbund* ("struct") zusammengefasst werden
- Beispiel: *Komplexe Zahlen*

Typ-Definition

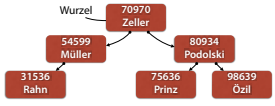
```
struct Complex {
  double real;
  double imag;
};
```

Variablen-Initialisierung

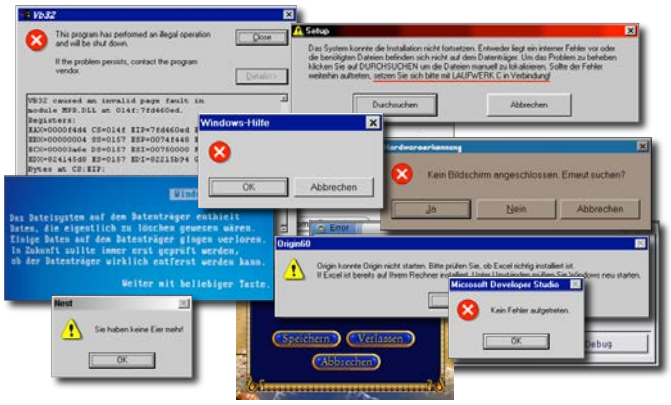
```
struct Complex c = {
    3.0,  // real
    4.0   // imag
};
```

Suchbäume

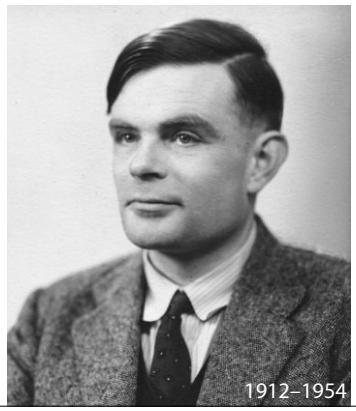
- Jeder *Knoten* hat (bis zu zwei) *Kinder:*
  Im *linken* Teilbaum sind alle kleineren,
  im *rechten* Teilbaum alle größeren Werte



Das Problem



Alan Turing



1936 schließlich führte Turing die Begriffe des Algorithmus und der Berechenbarkeit fassbar, indem er mit seinem Modell die Begriffe des Algorithmus und der Berechenbarkeit als formale, mathematische Begriffe definierte.

Halteproblem

- Nicht alle Probleme können von Programmen gelöst werden
- Das *Halteproblem* etwa besagt, dass es kein Programm gibt, das für ein beliebiges gegebenes Programm *P* entscheidet, ob es ein Ergebnis liefern wird (*hält*) oder nicht.

Collatz-Problem

(Wolfgang Collatz, 1937)

- Beginne mit einer natürlichen Zahl *n*
- Ist *n* gerade, so nimm als nächstes *n*/2
- Ist *n* ungerade, so nimm als nächstes 3*n*+1
- Wiederhole das Ganze

19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26,
13, 40, 20, 10, 5, 16, 8, 4, 2, 1, …

Collatz-Problem

(Wolfgang Collatz, 1937)

- Anscheinend mündet jede so definierte Folge irgendwann in 4, 2, 1, …
- Diese Eigenschaft ist unbewiesen

19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26,
13, 40, 20, 10, 5, 16, 8, 4, 2, 1, …

Halteproblem

```
void collatz(int n) {
  while (n != 1) {
    if (n % 2 == 0)
      n = n / 2;
    else
      n = 3 * n + 1;
  }
}
```

- Wird collatz() für jedes *n* halten (zurückkehren)?
- Lösung nur durch *Ausprobieren* (in unendlicher Zeit)

Es gibt keine Möglichkeit, für alle Programme vollautomatisch die Korrektheit zu beweisen

Halteproblem

Um zu zeigen, dass ein echtes Programm seine Anforderungen erfüllt, müssen wir entweder

- von Hand mathematisches *Wissen und Annahmen* einsetzen, um es zu beweisen, (was sehr aufwändig ist), oder
- das Programm *testen* und hoffen, dass unsere Tests ausreichen.

Edgar Degas: The Rehearsal. With a rehearsal, we want to check whether everything will work as expected. This is a test.

Again, a test. We test whether we can evacuate 500 people from an Airbus A380 in 90 seconds. This is a test.

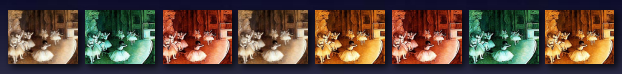
And: We test whether a concrete wall (say, for a nuclear reactor) withstands a plane crash at 900 km/h. Indeed, it does.

We can also test software this way. But software is not a planned linear show – it has a multitude of possibilities. So: if it works once, will it work again? This is the central issue of testing – and of any verification method.

Software ist vielfältig

We can also test software this way. But software is not a planned linear show – it has a multitude of possibilities. So: if it works once, will it work again? This is the central issue of testing – and of any verification method.

Software ist vielfältig

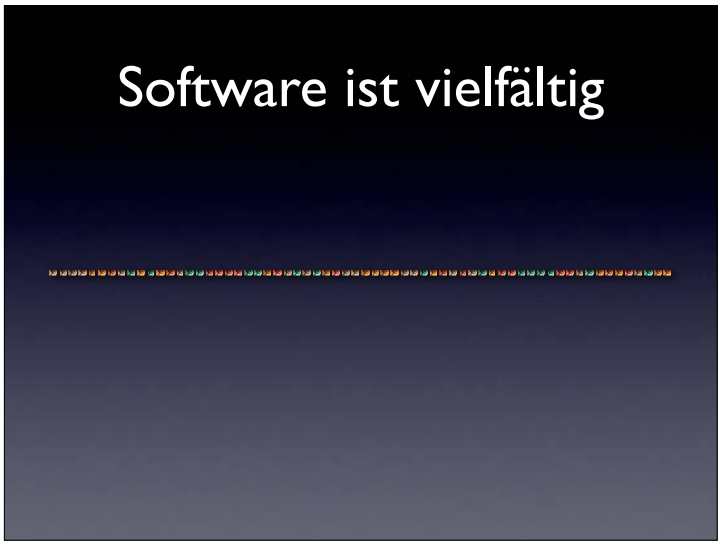


The problem is: There are many possible executions. And as the number grows…

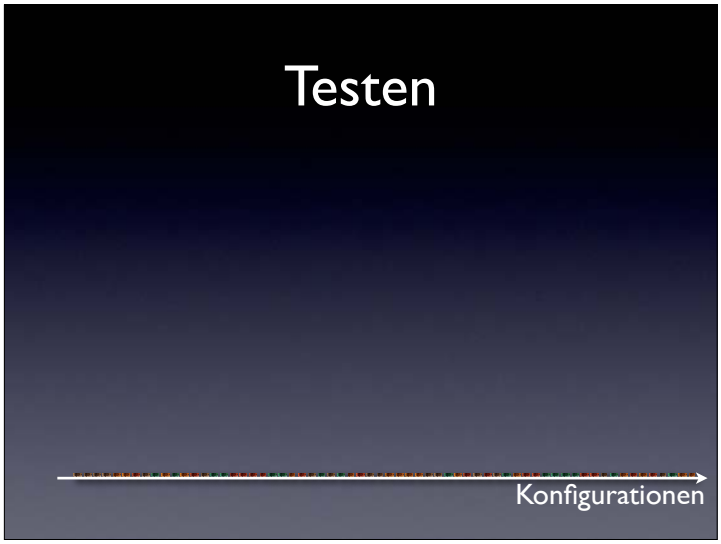
Software ist vielfältig

and grows…

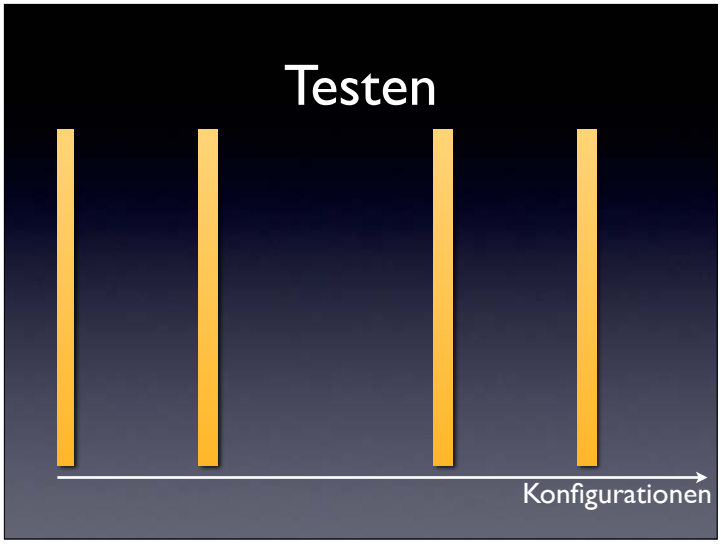
and grows…



and grows…



…you get an infinite number of possible executions, but you can only conduct a finite number of tests.



With testing, you pick a few of these Konfigurationens – and test them.

Testen von Hand

- Testen von Hand ist einfach:
  - Wir führen das Programm aus
  - Wir prüfen, ob es unseren Erwartungen entspricht
- Muss *nach jeder Änderung* wiederholt werden!

Automatischer Test

- Eine spezielle *Testfunktion* prüft eine andere Funktion auf korrekte Ergebnisse:

```
void test_sqrt() {
  if (sqrt(4) != 2)
    error();
  if (sqrt(9) != 3)
    error();
  if (sqrt(16) != 4)
    error();
}
```

- Nach jeder Änderung:
  Tests einfach neu laufen lassen

Zusicherungen

- Um eine Bedingung sicherzustellen, nutzen Programme *Zusicherungen*
- assert(*p*) schlägt fehl, wenn p nicht gilt

```
#include <assert.h>

void test_sqrt() {
  assert(sqrt(4) == 2);
  assert(sqrt(9) == 3);
  assert(sqrt(16) == 4);
}
```

Diagnose

- Normalerweise bricht assert(*p*) das Programm direkt ab (“abort()”)
- Wenn definiert, wird stattdessen die Funktion __assert() aufgerufen, die zusätzliche Informationen ausgibt.
- Auf Arduino/Galileo besonders hilfreich

Diagnose

```
#define __ASSERT_USE_STDERR
#include <assert.h>

void __assert(const char *failedexpr,
              const char *file,
              int line,
              const char *func)
{
     Serial.print(file);
     Serial.print(":");
     Serial.print(line);
     Serial.print(": ");
     Serial.print(func);
     Serial.print(": Assertion failed: ");
     Serial.println(failedexpr);
     abort();
}
```

```
Assert.ino:20: setup(): Assertion failed: 2 + 2 == 5
```

Demo

So, how can we cover as much behavior as possible?

Was testen?

- Ziel: *Jeden Aspekt des Verhaltens* abdecken
- Erfordertes Verhalten: Anhand *Spezifikation (Funktionales Testen)*
- Implementiertes Verhalten: Anhand *Code (Strukturelles Testen)*

Funktionales Testen

- *cgi_decode* nimmt eine Zeichenkette und
  1. ersetzt alle "+" durch Leerzeichen
  2. ersetzt alle "%*xx*" durch ein Zeichen mit dem Hexadezimalwert *xx*
     (gibt Fehlercode, wenn *xx* ungültig)
  3. Alle anderen Zeichen bleiben unverändert
- Diese Eigenschaften müssen getestet werden!

Funktionaler Test

```c
#include <assert.h>

// ersetzt alle "+" durch Leerzeichen
void test_cgi_decode_plus() {
  char *encoded = "foo+bar+";
  char decoded[20];

  int result = cgi_decode(encoded, decoded);
  assert(result == 0);
  assert(strcmp(decoded, "foo bar ") == 0);
}
```

Funktionaler Test

```c
#include <assert.h>

// ersetzt alle "%xx"
// durch ein Zeichen mit dem Hexadezimalwert xx
void test_cgi_decode_hex() {
  char *encoded = "foo%30bar";
  char decoded[20];

  int result = cgi_decode(encoded, decoded);
  assert(result == 0);
  assert(strcmp(decoded, "foo0bar") == 0);
}
```

Funktionaler Test

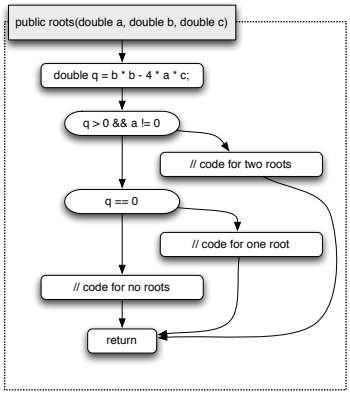
```c
#include <assert.h>

// ersetzt alle “%xx”
// durch ein Zeichen mit dem Hexadezimalwert xx
void test_cgi_decode_invalid_hex() {
  char *encoded = "foo%zzbar";
  char decoded[20];

  int result = cgi_decode(encoded, decoded);
  assert(result != 0);
}
```

Testsuite

- Eine *Testsuite* fasst mehrere Tests zusammen
- Nach jeder Änderung ausführen

```c
#include <assert.h>

// Alle Tests
void test_cgi_decode() {
   test_cgi_decode_plus();
   test_cgi_decode_hex();
   test_cgi_decode_invalid_hex();
}
```

Strukturelles Testen



- Orientiert sich an der *Struktur des Programms*
- Je mehr Teile des Programms *abgedeckt* (ausgeführt) sind, desto höher die Chance, Fehler zu finden
- “Teile” können sein: Anweisungen, Übergänge, Pfade, Bedingungen…

To talk about structure, we turn the program into a *control flow graph*, where statements are represented as nodes, and edges show the possible control flow between statements.

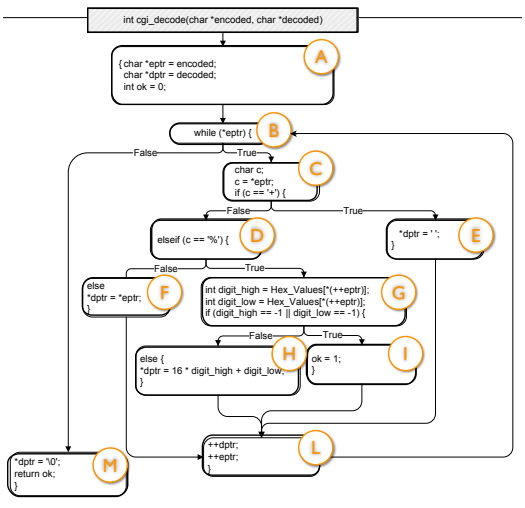
cgi_decode

```
/**
  * @title cgi_decode
  * @desc
  * Translate a string from the CGI encoding to plain ascii text
  * '+' becomes space, %xx becomes byte with hex value xx,
  * other alphanumeric characters map to themselves
  *
  * returns 0 for success, positive for erroneous input
  * 1 = bad hexadecimal digit
  */

int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
```

A

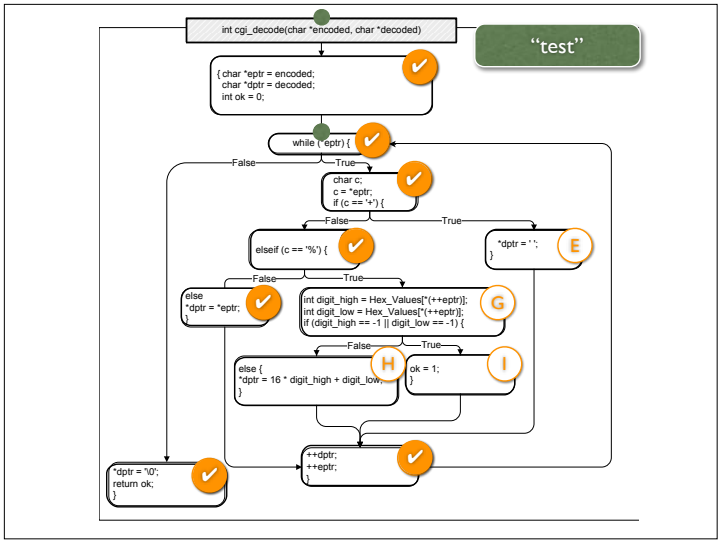
```
    while (*eptr)  /* loop to end of string ('\0' character) */
    {
        char c;
        c = *eptr;
        if (c == '+') {  /* '+' maps to blank */
            *dptr = ' ';
        } else if (c == '%') { /* '%xx' is hex for char xx */
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low  = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1)
                ok = 1; /* Bad return code */
            else
                *dptr = 16 * digit_high + digit_low;
        } else { /* All other characters map to themselves */
           *dptr = *eptr;
        }
        ++dptr; ++eptr;
    }

    *dptr = '\0';   /* Null terminator for string */
    return ok;
}
```

B C E D G I H F L M

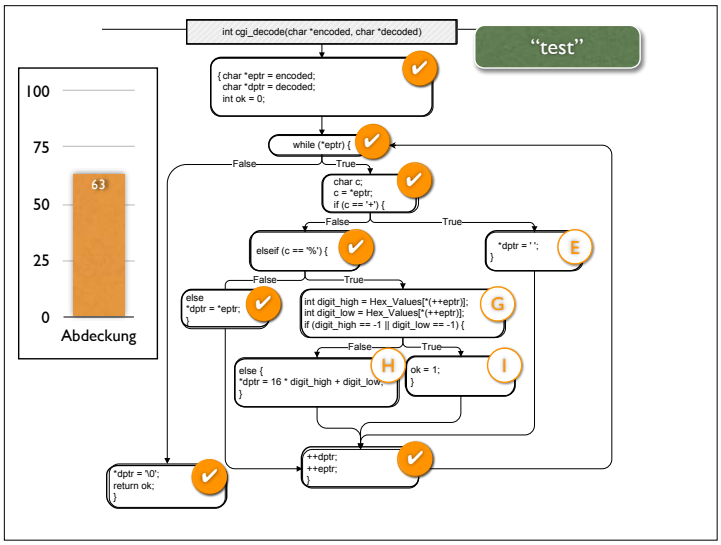


Here's an ongoing example. The function cgi_decode translates a CGI-encoded string (i.e., from a Web form) to a plain ASCII string, reversing the encoding applied by the common gateway interface (CGI) on common Web servers.
(from Pezze + Young, "Software Testing and Analysis", Chapter 12)

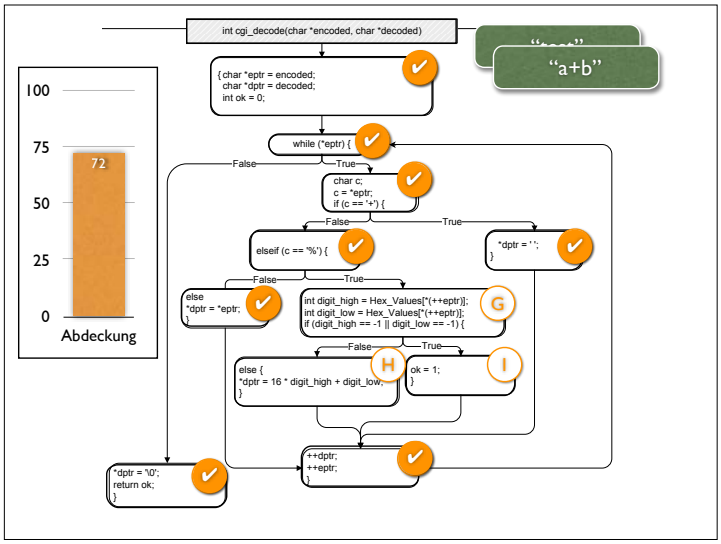
This is what cgi_decode looks as a CFG.
(from Pezze + Young, "Software Testing and Analysis", Chapter 12)



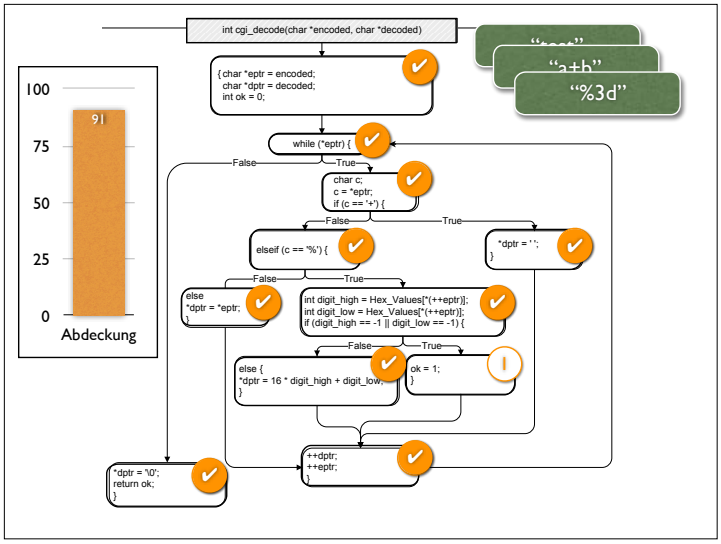
While the program is executed, one statement (or basic block) after the other is covered – i.e., executed at least once – but not all of them. Here, the input is "test"; checkmarks indicate executed blocks.

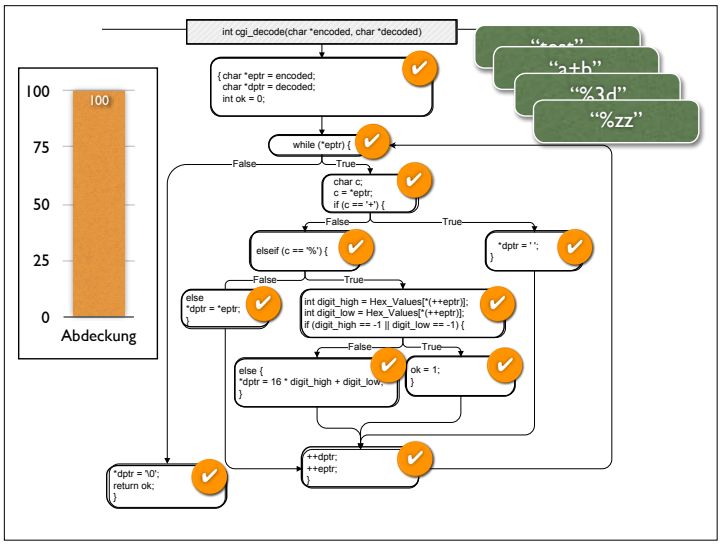


The initial Abdeckung is 7/11 blocks = 63%. We could also count the statements instead (here: 14/20 = 70%), but conceptually, this makes no difference.



and the Abdeckung increases with each additionally executed statement…



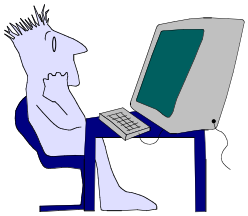


… until we reach 100% block Abdeckung (which is 100% statement Abdeckung, too).

Ein Test…

- soll *nicht* zeigen, dass ein Programm funktioniert
- sondern soll vielmehr zeigen, dass ein Programm *nicht* funktioniert
- Verlangt Kreativität beim Testen!

Wer testet?



Entwickler
- *versteht* das System
- wird *vorsichtig* testen
- will *Code* liefern

Unabhängiger Tester
- muss System *lernen*
- will *Fehler* aufdecken
- will *Qualität* liefern

From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 13

Der beste Tester

A good tester should be creative and destructive – even sadistic in places.
– Gerald Weinberg, “The psychology of computer programming”

Der Entwickler

The conflict between developers and testers is usually overstated, though.

Weinberg's Gesetz

Ein Entwickler ist nicht geeignet, den eigenen Code zu testen.

Theory: As humans want to be honest with themselves, developers are blindfolded with respect to their own mistakes.
Evidence: "seen again and again in every project" (Endres/Rombach)
From Gerald Weinberg, "The psychology of computer programming"

Sadistischer Test

```
#include <assert.h>

// ersetzt alle "%xx"
// durch ein Zeichen mit dem Hexadezimalwert xx
void test_cgi_decode_incomplete_hex() {
  char *encoded = "foo%g";
  char decoded[20];

  int result = cgi_decode(encoded, decoded);
  assert(result != 0);
}
```

- Führt zu Zugriff außerhalb Feldgrenzen

Fehlersuche

- Nach dem Testen folgt die *Fehlersuche*

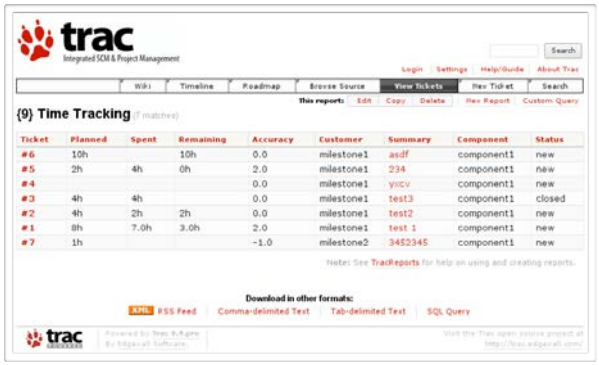
Nach dem Testen folgt die Fehlersuche

Systematische Fehlersuche

T rack the problem	*Problem verfolgen*
R eproduce	*Reproduzieren*
A utomate	*Automatisieren*
F ind Origins	*Ursprünge finden*
F ocus	*Fokussieren*
I solate	*Isolieren*
C orrect	*Korrigieren*

T
R
A
F
F
I
C

Problem verfolgen



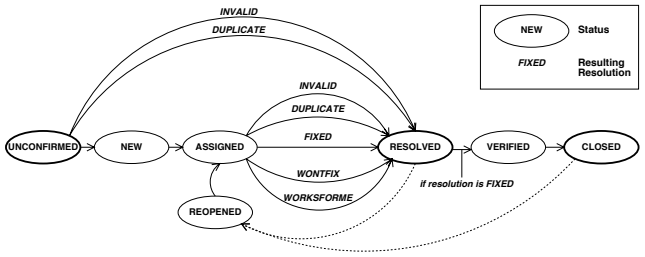
T
R
A
F
F
I
C

Problem verfolgen

- Jedes Problem wird in die Fehler-Datenbank eingetragen
- Die Priorität bestimmt, welches Problem als nächstes bearbeitet wird
- Sind alle Probleme behoben, ist das Produkt fertig

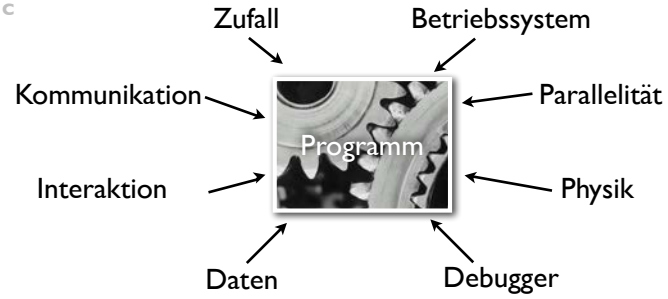
T R A F F I C

Lebenszyklus eines Problems



T R A F F I C

Reproduzieren



T R A F F I C

Automatisieren

```
// Test for host
public void testHost() {
  int noPort = -1;
  assertEquals(askigor_url.getHost(), "www.askigor.org");
  assertEquals(askigor_url.getPort(), noPort);
}

// Test for path
public void testPath() {
  assertEquals(askigor_url.getPath(), "/status.php");
}

// Test for query part
public void testQuery() {
  assertEquals(askigor_url.getQuery(), "id=sample");
}
```

TRAFFIC

Automatisieren

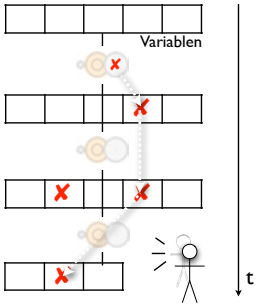
- Jedes Problem sollte automatisch reproduzierbar sein
- Dies geschieht über geeignete JUnit-Testfälle
- Nach jeder Änderung werden die Testfälle ausgeführt

TRAFFIC

Ursprung finden

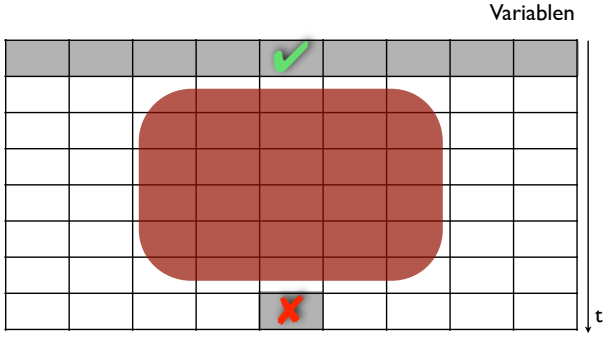
1. Der Programmierer erzeugt einen *Defekt* – einen Fehler im Code
2. Der ausgeführte Defekt erzeugt eine *Infektion* – einen Fehler im Zustand
3. Die Infektion breitet sich aus…
4. …und wird als *Fehlverhalten* sichtbar.

Diese Infektionskette müssen wir *brechen*.



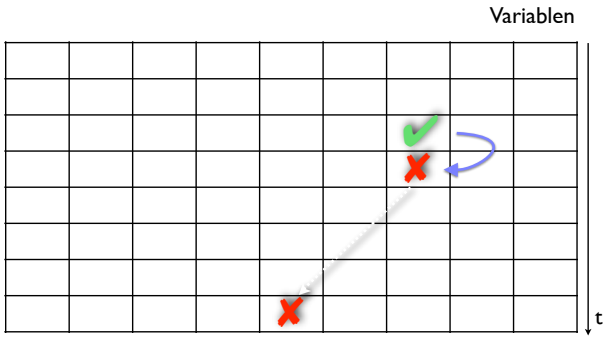
TRAFFIC

Ursprung finden



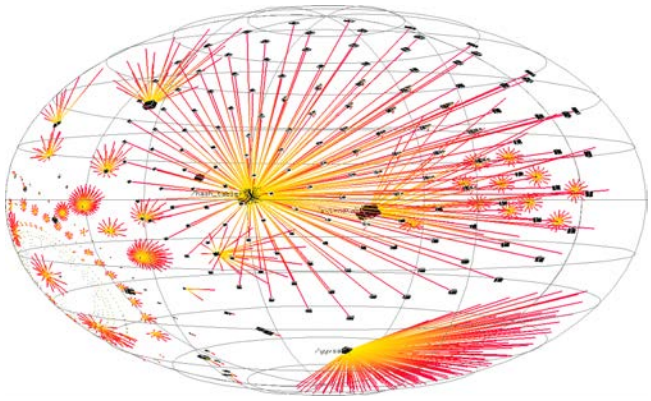
TRAFFIC

Der Defekt



TRAFFIC

Ein Programmzustand

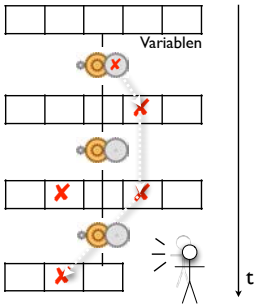


TRAFFIC

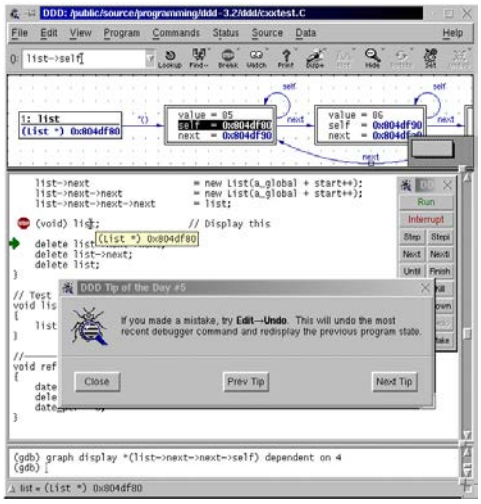
TRAFFIC

Ursprung finden

1. Wir beginnen mit einer *bekannten Infektion* (etwa am *Ende der Ausführung*)
2. Wir suchen die Infektion im *vorherigen Zustand*

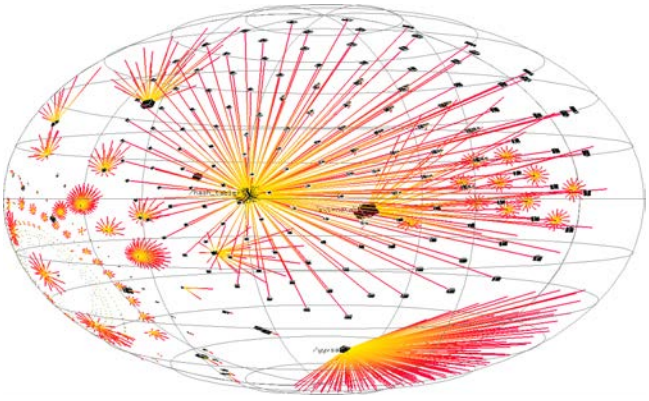


TRAFFIC



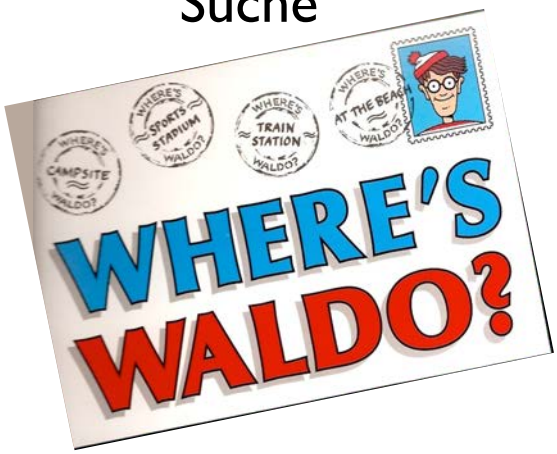
TRAFFIC

Ein Programmzustand



TRAFFIC

Suche



TRAFFIC

Fokussieren

Bei der Suche nach Infektionen konzentrieren uns auf Stellen im Zustand, die

- *wahrscheinlich falsch* sind (z.B. weil hier früher Fehler aufgetreten sind)
- *explizit falsch* sind (z.B. weil sie eine *Zusicherung* verletzen)

Zusicherungen sind das effektivste Mittel, Infektionen zu finden.

TRAFFIC

Infektionen finden

```
struct Time {
    int hour;     // 0..23
    int minutes;  // 0..59
    int seconds;  // 0..60 (incl. leap seconds)
};

void set_hour(struct Time *t, int h);
…
```

Jede Zeit von 00:00:00 bis 23:59:60 ist gültig

TRAFFIC

Ursprung finden

```
int sane_time(struct time *t)
{
    return (0 <= t->hour && t->hour <= 23) &&
           (0 <= t->minutes && t->minutes <= 59) &&
           (0 <= t->seconds && t->seconds <= 60);
}

void set_hour(struct Time *t, int h)
{
    assert (sane_time(t));  // Vorbedingung
    …
    assert (sane_time(t));  // Nachbedingung
}
```

TRAFFIC

Ursprung finden

```
int sane_time(struct time *t)
{
    return (0 <= t->hour && t->hour <= 23) &&
           (0 <= t->minutes && t->minutes <= 59) &&
           (0 <= t->seconds && t->seconds <= 60);
}
```

sane() ist die *Invariante* eines Time-Objekts:

- gilt *vor* jeder öffentlichen Methode
- gilt *nach* jeder öffentlichen Methode

TRAFFIC

Ursprung finden

- Vorbedingung schlägt fehl = Infektion *vor* Methode
- Nachbedingung schlägt fehl = Infektion *nach* Methode
- Alle Zusicherungen ok = keine Infektion

```
void set_hour(struct Time *t, int h)
{
    assert (sane_time(t));  // Vorbedingung
    …
    assert (sane_time(t));  // Nachbedingung
}
```

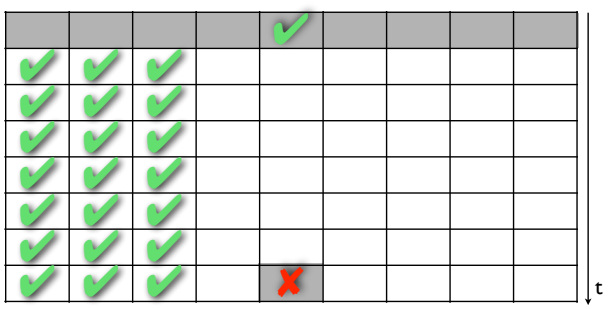
TRAFFIC

Komplexe Invarianten

```
int sane_tree(struct Tree *t) {
      assert (rootHasNoParent(t));
      assert (rootIsBlack(t));
      assert (redNodesHaveOnlyBlackChildren(t));
      assert (equalNumberOfBlackNodesOnSubtrees(t));
      assert (treeIsAcyclic(t));
      assert (parentsAreConsistent(t));

      return 1;
  }
}
```

TRAFFIC

Zusicherungen



T
R
A
F
F
I
C

Fokussieren

- Alle möglichen Einflüsse müssen geprüft werden
- Konzentration auf wahrscheinlichste Kandidaten
- Zusicherungen helfen schnell, Infektionen zu finden

T
R
A
F
F
I
C

Isolieren

- Fehlerursachen sollen *systematisch* eingeengt werden – mit Beobachtungen und Experimenten.

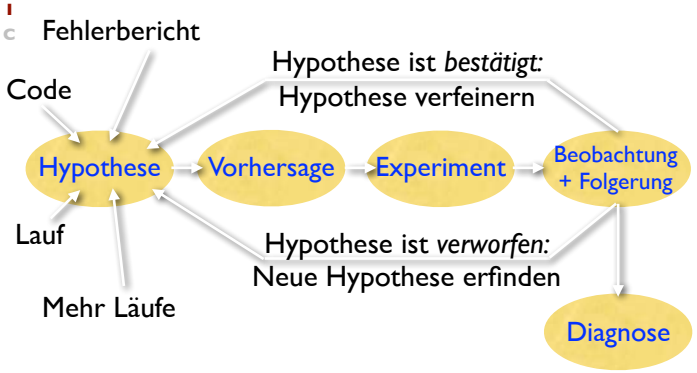
T
R
A
F
F
I
C

Wissenschaftliche Methode

1. Beobachte einen Teil des Universums
2. Erfinde eine *Hypothese*, die mit der Beobachtung übereinstimmt
3. Nutze die Hypothese, um Vorhersagen zu machen.
4. Teste die Vorhersagen durch Experimente oder Beobachtungen und passe die Hypothese an.
5. Wiederhole 3 and 4, bis die Hypothese zur *Theorie* wird.

TRAFFIC

Wissenschaftliche Methode



TRAFFIC

TRAFFIC

Explizite Hypothesen

Hypothesis	The exec… …uses a[0] = 0
Prediction	At l… …hould hold.
Experiment	…me 37.
Observat…	…as as predicted.
Conclusion	…nesis is **confirmed**.

Wer alles im Kopf behält, spielt Mastermind blind!

T R A F F I C

Explizite Hypothesen

T R A F F I C

Isolieren

- Wir wiederholen die Suche nach Infektions-Ursprüngen, bis wir den Defekt gefunden haben.
- Wir gehen *systematisch* vor – im Sinne der wissenschaftlichen Methode
- Durch *explizite* Schritte leiten wir die Suche und können sie jederzeit nachvollziehen

T R A F F I C

Korrektur

Vor der Korrektur müssen wir prüfen, ob der Defekt

- tatsächlich ein *Fehler* ist und
- das Fehlverhalten *verursacht*

Erst wenn beides verstanden ist, dürfen wir den Fehler korrigieren.

TRAFFIC

The Devil's Guide to Debugging

Finde den Defekt durch Raten:

- Verstreue überall Debugging-Anweisungen
- Ändere den Code, bis etwas funktioniert
- Mache keine Kopien von alten Versionen
- Versuche gar nicht erst zu verstehen, was das Programm tun soll

TRAFFIC

The Devil's Guide to Debugging

Verschwende keine Zeit damit, dem Problem auf den Grund zu gehen

- Die meisten Probleme sind ohnehin trivial

TRAFFIC

The Devil's Guide to Debugging

Benutze die offensichtlichste Reparatur:

- Repariere nur das, was Du siehst:

```
x = compute(y);
// compute(17) is wrong – fix it
if (y == 17)
    x = 25.15;
```

Warum sich mit compute() beschäftigen?

T R A F F I C

Erfolgreiche Korrektur

T R A F F I C

Hausaufgaben

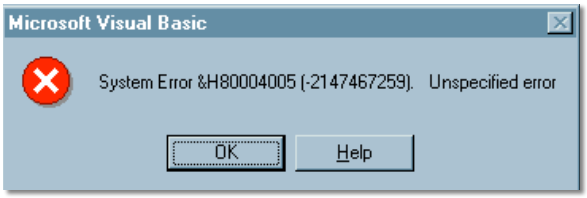
- Tritt das Fehlverhalten nicht mehr auf?
  (Falls doch, sollte dies eine große Überraschung sein)
- Könnte die Korrektur neue Fehler einführen?
- Wurde derselbe Fehler woanders gemacht?
- Ist meine Korrektur ins Versionsmanagement und Problem-Tracking eingespielt?

Vorgehensweise

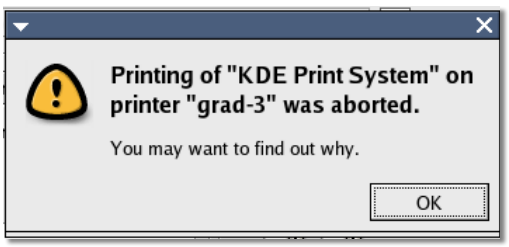
T rack the problem	*Problem verfolgen*
R eproduce	*Reproduzieren*
A utomate	*Automatisieren*
F ind Origins	*Ursprünge finden*
F ocus	*Fokussieren*
I solate	*Isolieren*
C orrect	*Korrigieren*

Was ist ein Problem?

- Ein *Problem* ist alles, was vom Benutzer als solches wahrgenommen wird
- Entwickler müssen dafür eine *Benutzer-Perspektive* einnehmen können



Diese höchst aussagekräftige Fehlermeldung ist Microsoft Visual Basic 5.0 zu entnehmen. Nach dem Klicken auf Help erhalten wir:
Visual Basic encountered an error that was generated by the system or an
external component and no other useful information was returned. The specified error number is returned by the system or external component (usually from an Application Interface call) and is displayed in hexadecimal and decimal format.
Lösung des Problems: Neu booten?

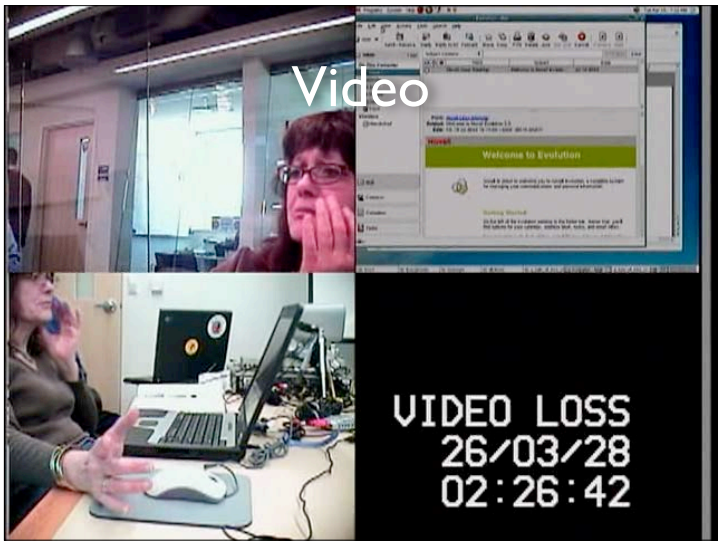



```
$ ssh somehost.foo.com
You don't exist, go away!
$ _
```

Diese Fehlermeldung erscheint etwa, wenn der NIS-Server gerade nicht erreichbar ist. Nicht, daß man den Benutzer darüber aufklären würde...

Was ist ein Problem?

- Ein *Problem* ist alles, was vom Benutzer als solches wahrgenommen wird
- Entwickler müssen dafür eine *Benutzer-Perspektive* einnehmen können
- Lösung: *Test mit echten Benutzern!*



Task: Email A Tale of Two Cities to arthur@ximian.com; Subject14
http://www.betterdesktop.org/wiki/index.php?title=Data

Typische Vorgehensweise: Benutzer sollen mit dem System eine bestimmte Aufgabe erledigen – und halten anschließend fest, was sie gestört hat.

Zusicherungen

- Um eine Bedingung sicherzustellen, nutzen Programme *Zusicherungen*
- assert(*p*) schlägt fehl, wenn p nicht gilt

```
#include <assert.h>

void test_sqrt() {
  assert(sqrt(4) == 2);
  assert(sqrt(9) == 3);
  assert(sqrt(16) == 4);
}
```

Diagnose

```
#define __ASSERT_USE_STDERR
#include <assert.h>

void __assert(const char *failedexpr,
              const char *file,
              int line,
              const char *func)
{
    Serial.print(file);
    Serial.print(":");
    Serial.print(line);
    Serial.print(": ");
    Serial.print(func);
    Serial.print(": Assertion failed: ");
    Serial.println(failedexpr);
    abort();
}
```

```
Assert.ino:20: setup(): Assertion failed: 2 + 2 == 5
```

Was testen?

- Ziel: *Jeden Aspekt des Verhaltens* abdecken
- Erfordertes Verhalten: Anhand *Spezifikation (Funktionales Testen)*
- Implementiertes Verhalten: Anhand *Code (Strukturelles Testen)*

Systematische Fehlersuche

T rack the problem	*Problem verfolgen*
R eproduce	*Reproduzieren*
A utomate	*Automatisieren*
F ind Origins	*Ursprünge finden*
F ocus	*Fokussieren*
I solate	*Isolieren*
C orrect	*Korrigieren*

Handouts

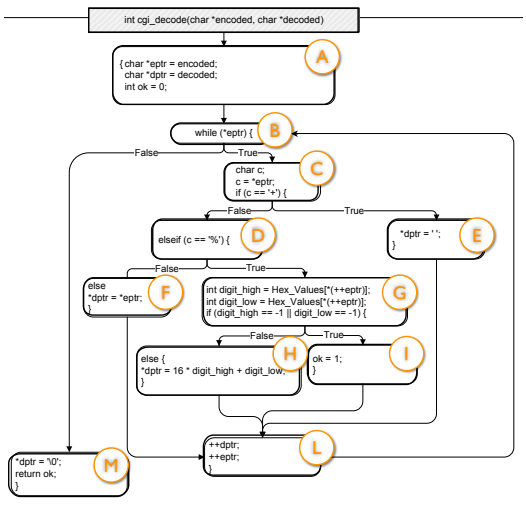
---

cgi_decode

```
/**
  * @title cgi_decode
  * @desc
  * Translate a string from the CGI encoding to plain ascii text
  * '+' becomes space, %xx becomes byte with hex value xx,
  * other alphanumeric characters map to themselves
  *
  * returns 0 for success, positive for erroneous input
  * 1 = bad hexadecimal digit
  */

int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
```

A

Here's an ongoing example. The function cgi_decode translates a CGI-encoded string (i.e., from a Web form) to a plain ASCII string, reversing the encoding applied by the common gateway interface (CGI) on common Web servers.
(from Pezze + Young, "Software Testing and Analysis", Chapter 12)



This is what cgi_decode looks as a CFG.
(from Pezze + Young, "Software Testing and Analysis", Chapter 12)

```
    while (*eptr)  /* loop to end of string ('\0' character) */ B
    {
        char c;
        c = *eptr; C
        if (c == '+') {  /* '+' maps to blank */
            *dptr = ' '; E
        } else if (c == '%') { /* '%xx' is hex for char xx */ D
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low  = Hex_Values[*(++eptr)]; G
            if (digit_high == -1 || digit_low == -1)
                ok = 1; /* Bad return code */ I
            else
                *dptr = 16 * digit_high + digit_low; H
        } else { /* All other characters map to themselves */
           *dptr = *eptr; F
        }
        ++dptr; ++eptr; L
    }

    *dptr = '\0';   /* Null terminator for string */ M
    return ok;
}
```

Funktionaler Test

```c
#include <assert.h>

// ersetzt alle "+" durch Leerzeichen
void test_cgi_decode_plus() {
  char *encoded = "foo+bar+";
  char decoded[20];

  int result = cgi_decode(encoded, decoded);
  assert(result == 0);
  assert(strcmp(decoded, "foo bar ") == 0);
}
```