Jeder Programmierer kennt die Situation: Ein Programm läuft nicht so, wie es soll. Ich stelle Techniken vor, die automatisch
(a) die Ursachen eines Fehlverhaltens finden - indem wir genau die Aspekte isolieren, die das Zustandekommen eines Fehlers verursachen;
(b) Programmfehler finden - indem wir Anwendungen systematisch und vollautomatisch testen; und
(c) vorhersagen, wo in Zukunft Fehler auftreten werden - indem wir maschinell lernen, welche Code- und Prozesseigenschaften bisher mit

An F-16 fighter plane on the northern hemisphere.
Why the northern hemisphere, you ask?

Because this is what an F-16 on the southern hemisphere would look like. (BTW, interesting effect if you drop a bomb :-)

From risks.digest, volume 3, issue 44:
- Since the F-16 is a fly-by-wire aircraft, the computer keeps the pilot from doing dumb things to himself. So if the pilot jerks hard over on the joystick, the computer will instruct the flight surfaces to make a nice and easy 4 or 5 G flip. But the plane can withstand a much higher flip than that. So when they were 'flying' the F-16 in simulation over the equator, the computer got confused and instantly flipped the plane over. killing the...
One of the first things the Air Force test pilots tried on an early F-16 was to tell the computer to raise the landing gear while standing still on the runway. Guess what happened? Scratch one F-16. (my friend says there is a new subroutine in the code called 'wait_on_wheels' now...) [weight?]

(Folklore has it that the programmer checked the height above sea level rather than the height above ground - AZ)

What camera crews depicted was truly a disaster; carts jammed together, damaged luggage everywhere, some bags literally split in half, and the tattered remains of clothing strewn about causing subsequent carts to derail. Finally, adding insult to injury, half the luggage that survived the ordeal ended up at the wrong terminal.
Get #years, #days since 1980

days = ...;
year = 1980;
while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 366) {
            days -= 366; year += 1;
        }
    }
    else {
        days -= 365; year += 1;
    }
}

http://www.aeroxp.org/2009/01/lesson-on-infinite-loops/
http://www.youtube.com/watch?v=fYTJ9v2vsAE
Der erste Bug
9. September 1947

Wo sind die Fehler?

Prozess
frühere Fehlerorte
und deren Eigenschaften

Programm
Programmtests
Programmanalysen

Retrieved by a technician from the Harvard Mark II machine on September 9, 1947.
Now on display at the Smithsonian, Washington
Wo sind die Fehler?

Prozess
frühere Fehlerorte
und deren Eigenschaften

Such software archives are being used in practice all the time. If you file a bug, for instance, the report is stored in a bug database, and the resulting fix is stored in the version archive.
Fehler auf Code abbilden

Sind es die Entwickler?

Macht Erfahrung einen Unterschied?

Je mehr Erfahrung, desto mehr Fehler!

Woher kommen diese Fehler?

These databases can then be mined to extract interesting information. From bugs and changes, for instance, we can tell how many bugs were fixed in a particular location.
Oder die Geschichte?
Wir haben hier viele Fehler gefunden…
Dann sind dort noch mehr!

Wie steht es mit Metriken?
Korrelieren Metriken mit Fehlerdichte?
Manchmal!

Programmiersprache?
Sind gotos schädlich?
Keine Korrelation!
Ok. Problembereich?

Welche Tokens sind relevant?

import • extends • implements

Vorhersage | Komponente | Tatsache
--- | --- | ---
1 | nsDOMClassInfo | 3
2 | SGGridRowLayout | 95
3 | xpcprivate | 6
4 | jxml | 2
5 | nsGenericHTMLElement | 8
6 | jgc | 3
7 | nsISEnvironment | 12
8 | jsfun | 1
9 | nsHTMLLabelElement | 18
10 | nsHttpTransaction | 35
Wissen, wo die Fehler sind
Kalibrieren der Fehlervorhersage
Wissen, wo die nächsten Fehler sind
Vollautomatisch!

Sogar das heute journal hat einen Bericht gebracht – seitdem bin ich bekannt aus Funk und Fernsehen :-)

Wo sind die Fehler?
Prozess
frühere Fehlerorte und deren Eigenschaften
Wo sind die Fehler?

Prozess
frühere Fehlerorte und deren Eigenschaften

Programm
Programmtests Programmanalysen

Es gibt viele Werkzeuge, die generische Fehler finden – aber wir wollen spezifische Fehler finden!

Testen

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
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.

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

and grows…
Software ist vielfältig and grows...

Software ist vielfältig and grows...

Testen …you get an infinite number of possible executions, but you can only conduct a finite number of tests.
...and this was something the first testers also needed to realize.

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

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

- Fehlschlag (wertvoll)!
- Kein Fehlschlag

...aber in einigen Bereichen dicht

Ziel: Aus jedem Bereich wenigstens eine Ausführung erhalten

Funktionales Testen = Eingabe in Äquivalenzbereiche aufteilen

Struktur-Testen

Path testing
Boundary interior testing
Compound condition testing
MC/DC testing
LCSAJ testing
Branch and condition testing
Basic condition testing
Loop boundary testing
Statement testing

Figure 12.2: The control flow graph of function cgi

test

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

and the coverage increases with each additionally executed statement...
... until we reach 100% block coverage (which is 100% statement coverage, too).

All these techniques attempt to find the needle in the haystack…

Können wir den Test automatisieren?
Automatisierung

- Automatisches Ausführen
- Automatisches Generieren
- Automatisches Prüfen

Capture + Replay

- Wir können Tastatur und Maus aufzeichnen
- … und nach Belieben wieder abspielen!

As an example, here’s the Addressbook program: a simple Java application which manages a set of contacts that can be entered, searched, and grouped into.
Wer definiert all die Tests?

Zufallstesten

```java
public class RandoopTest0 extends TestCase {
    ...

    public void test8() throws Throwable {
        AddressBook var0 = new AddressBook();
        EventHandler var1 = var0.getEventHandler();
        Category var2 = var0.getRootCategory();
        Contact var3 = new Contact();
        AddressBook var4 = new AddressBook();
        EventHandler var5 = var4.getEventHandler();
        Category var6 = var4.getRootCategory();
        String var7 = var6.getName();
        var0.addCategory(var3, var6);
        SelectionHandler var9 = new SelectionHandler();
        AddressBook var10 = new AddressBook();
        EventHandler var11 = var10.getEventHandler();
    }
}
```

As an example, here’s the Addressbook program: a simple Java application which manages a set of contacts that can be entered, searched, and grouped into categories.

Here’s a test case generated by Randoop. It’s >200 lines long...

... and in the end, it fails. What do you do now?
Zufallstesten

- Einfach zu realisieren…
- …aber erzeugt viele unsinnige Tests!

The catch is: There’s never more than one addressbook! So the Randoop test makes little sense, because it violates an implicit precondition. When testing the Addressbook classes, Randoop detects * 112 failures. However, all of them are false, pointing to an error in the generated test case rather than the application itself, which has *0 problems.

Ein Fehlalarm

A simplified version of the above. If you use two address book objects and make one’s category depend on one the other, it’ll crash.
System-Tests

- Erzeuge Tests für die Bedienoberfläche
- Jede Eingabe ist korrekt
- Keine Fehlalarme

Genetische Algorithmen

Population initialisieren → Population bewerten → Solange nicht fertig → Beste Lösung

- Eltern auswählen
- Eltern rekombinieren

Eingaben

Abdeckung

Population initialisieren → Population bewerten → Solange nicht fertig → Beste Lösung

- Eltern auswählen
- Eltern rekombinieren

- Zweig-Abstand
- Mutation der Eingabe
What I'm going to demo you now is our prototype called EXSYST, for Explorative SYStem Testing. EXSYST takes a Java program with a graphical user interface, such as our Addressbook example. It then generates user inputs such as mouse clicks or keystrokes and feeds them into the program. What you see here is EXSYST clicking and typing into the address book program; at the top, you see the statement coverage achieved so far. (Normally, all of this takes place in the background, so you don't see it, and it is also much much faster).

At first, these inputs are completely random, as you can see in these initial
Erzielte Abdeckung

<table>
<thead>
<tr>
<th></th>
<th>Randoop</th>
<th>Exsyst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressbook</td>
<td>25</td>
<td>75</td>
</tr>
<tr>
<td>Calculator</td>
<td>50</td>
<td>75</td>
</tr>
<tr>
<td>TerpSpreadSheet</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>TerpWord</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>TerpPresent</td>
<td>75</td>
<td>75</td>
</tr>
</tbody>
</table>

The results are clear. Although it’s going through the GUI, EXSYST achieves a far higher coverage than Randoop.

Here are the results for *Addressbook* and ***three more*** programs which can be invoked with a few simple inputs. In AddressBook, for instance, if you press the Apply button without having any contact,

EXSYST found failures in all five programs which can be invoked with a few simple inputs. In AddressBook, for instance, if you press the Apply button without having any contact,

But still, testing suffers from what I call Dijkstra’s curse – a double meaning, as it applies both to testing as to his famous quote. Is there something that can find the absence of errors?
Areas missing might be: the operating system, the hardware, all of the world the system is embedded in (including humans!)
Außer Kontrolle

Moderne Programme sind nicht mehr zu beweisen:

- Mehrere Sprachen
- Obskurer oder nicht verfügbarer Code
- Verteilte Aufrufe

Well, everyone has. You start a browser, you have it all. None of this is what program analysis can handle these days. We’re talking scripts, we’re talking distributed, we’re talking amateurs, we’re talking security.
We might not be able to cover all abstraction levels in all configurations, but we can do our best to cover as much as possible.