Testing and Debugging

Programming for Engineers

Winter 2015

Andreas Zeller, Saarland University

The Problem

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.

Alan Turing

1912–1954
Halting Problem

- Not all problems can be solved by programs
- E.g. the halting problem states that there is no program which can decide for an arbitrary program $P$, whether it will (eventually) return a result or not.

Collatz Conjecture

(Lothar Collatz, 1937)

- Start with an integer $n$
- If $n$ is even, take $n/2$ next
- If $n$ is odd, take $3n+1$ next
- repeat

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

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

- Will collatz() return for every n?
- Solution only by trial (in infinite time)

It is impossible to show correctness automatically for all programs

Halting Problem

To show that a real program fulfils its requirements, we must either

- use mathematical knowledge and assumptions to prove it by hand (which is very hard), or
- we must test it and hope that our tests suffice.

Testing
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.

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

and grows...

Software is Diverse

and grows...

Software is Diverse

and grows...

Software is Diverse

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.

**Manual Testing**

- Manual testing is easy:
  - We execute the program
  - We examine whether it meets our expectations
  - Must be repeated after every change!
Automatic Testing

- A special test function checks another function for correctness:

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

- After every change: simply re-execute the tests

Assertions

- In order to ensure a condition, programs use assertions
- `assert(p)` fails if `p` does not hold

```c
#include <assert.h>
void test_sqrt() {
    assert(sqrt(4) == 2);
    assert(sqrt(9) == 3);
    assert(sqrt(16) == 4);
}
```

Diagnosis

- Usually `assert(p)` halts the program directly ("abort()")
- If defined, the function `__assert()` is called instead, which prints additional information.
- Especially useful on Arduino
```c
#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

---

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

---

**How to Test?**

- **How do we cover as much behaviour as possible?**

---

**What to Test?**

- **Goal:** Cover every aspect of the behaviour
- **Required behaviour:** by specification (functional testing)
- **Implemented behaviour:** by code (structural testing)
**Functional Testing**

- *cgi_decode* takes a string and
  1. replaces every "+" with a space
  2. replaces every "%xx" with a character with hexadecimal value *xx* (returns an error code if *xx* is invalid)
  3. All other characters remain unchanged
- These properties must be tested!

```c
#include <assert.h>

#include <assert.h>
// replaces every "+" with a space
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);
}

#include <assert.h>
// replaces every "%xx" with a character with hexadecimal value *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, "foo 0bar") == 0);
}
```
# Functional Testing

```c
#include <assert.h>

// replaces every "%xx" with a character with hexadecimal value xx
void test_cgi_decode_invalid_hex() {
    char *encoded = "foo%zzbar";
    char decoded[20];
    int result = cgi_decode(encoded, decoded);
    assert(result != 0);
}
```

---

# Test Suite

- A test suite combines multiple tests
- Execute after every change

```c
#include <assert.h>

// All tests
void test_cgi_decode() {
    test_cgi_decode_plus();
    test_cgi_decode_hex();
    test_cgi_decode_invalid_hex();
}
```

---

# Structural Testing

- Based on the structure of the program
- The more parts of the program are covered (executed), the higher the chance to find errors
- "Parts" can be instructions, transition, paths, conditions…

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.
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)
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...
A Test…

- should not show that a program works
- but rather show that a program does not work
- requires creativity in testing!
Who Should Test?

Developer
- understands the system
- will test cautiously
- wants to deliver code

Independent Tester
- must learn the system
- wants to uncover errors
- wants to deliver quality

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

The Best Tester

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

The Developer

The conflict between developers and testers is usually overstated, though.
Weinberg’s Law

A developer is not suited to test their own code.

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”

Sadistic Test

```c
#include <cassert.h>

// replaces every "%xx" // with a character with hexadecimal value xx
void test_cgi_decode_incomplete_hex() {
    char *encoded = "foo%g";
    char decoded[20];
    int result = cgi_decode(encoded, decoded);
    assert(result != 0);
}

• Leads to access outside array bounds
```

Debugging

• Testing is followed by debugging

Nach dem Testing folgt die Fehlersuche
Systematic Debugging

- **T**rack the problem
- **R**eproduce
- **A**utomate
- **F**ind Origins
- **F**ocus
- **I**solate
- **C**orrect

---

Tracking the Problem

- Every problem is entered into the bug database
- The priority determines what problem will be addressed next
- When all problems are solved, the product is finished
Life Cycle of a Problem

Reproducing

Randomness Operating system
Communication Parallelism
Interaction Physics
Data Debugger

Automating

```java
// 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");
}
```
Automating

- Every problem should be automatically reproducible
- This is done by means of unit tests
- The test cases are executed after every change

Finding the Origin

1. The programmer creates a defect – an error in the code
2. The executed defect creates an infection – an error in the program state
3. The infection spreads…
4. …and becomes visible as a malfunction.

We must break this infection chain.
Finding the Origin

1. We start with a known infection (e.g., at the end of the execution)
2. We look for the infection in the previous state

A Program State
When searching for infections, we focus on places in the program state, that are

- probably wrong (e.g. because there were errors here previously)
- explicitly wrong (e.g. because they fail an assertion)

Assertions are the most effective means for finding infections.
Finding Infections

```c
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);
```

Every value from 00:00:00 to 23:59:60 is valid

---

Finding the Origin

```c
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)); // Precondition
    ...
    assert(sane_time(t)); // Postcondition
}
```

---

Finding the Origin

sane() is the **invariant** of a time object:

- **holds before** every time function
- **holds after** every time function
Finding the Origin

- Precondition fails = infection before the function
- Postcondition fails = infection in the function itself
- All assertions ok = no infection

```c
void set_hour(struct Time *t, int h)
{
    assert (sane_time(t)); // precondition
    ...
    assert (sane_time(t)); // postcondition
}
```

Complex Invariants

```c
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;
}
```

Assertions
Focusing

• All possible influences must be checked
• Focusing on most likely candidates
• Assertions help to find infections fast

Isolating

• Error causes are narrowed down systematically – using observations and experiments.

The Scientific Method

1. Observe a part of the universe
2. Formulate a hypothesis that is consistent with the observation
3. Use the hypothesis to make predictions.
4. Test the predictions using experiments or observations and adapt the hypothesis.
5. Repeat 3 and 4 until the hypothesis becomes a theory.
The Scientific Method

- Hypothesis
- Prediction
- Experiment
- Observation
- Conclusion

Hypothesis is confirmed:
- refine the hypothesis

Hypothesis is disproved:
- invent new hypothesis

Diagnosis

Remembering everything is like playing Mastermind with your eyes closed!

Explicit Hypotheses

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>The execution causes $a[0] = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td>At line 37, $a[0]$ should hold.</td>
</tr>
<tr>
<td>Experiment</td>
<td>Observe $a[0]$ at line 37.</td>
</tr>
<tr>
<td>Observation</td>
<td>$a[0] = 0$ holds as predicted.</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Hypothesis is confirmed.</td>
</tr>
</tbody>
</table>
Explicit Hypotheses

Isolating

• We repeat the search for infection origins until we find the defect.
• We proceed systematically — in terms of the scientific method
• We guide the search through explicit steps which we can retrace at any time

Correcting

Before correcting we must ensure that the defect
• is indeed an error and
• it causes the malfunction

Only when both are ensured and understood, we may correct the error.
The Devil’s Guide to Debugging

Find the defect by guessing:

• Spread debugging instructions everywhere
• Change the code until something works
• Don’t make backups of old versions
• Don’t even try to understand what the program is supposed to do

The Devil’s Guide to Debugging

Don’t waste time trying to get to the bottom of the problem

• Most problems are trivial anyway

The Devil’s Guide to Debugging

Use the most obvious repair:

• Repair only what you see:

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

Why deal with compute()?
Successful Correction

Homework

- Is the malfunction no longer present? (it should be a big surprise if it is still there)
- Could the correction introduce new errors?
- Was the same error made elsewhere?
- Is my correction entered into the version control system and bug tracking?

Systematic Debugging

Track the problem
Reproduce
Automate
Find Origins
Focus
Isolate
Correct
What is a Problem?

- Everything is a problem, that is perceived as such by the user
- Developers must be able to take a user perspective

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

$ ssh somehost.foo.com
You don't exist, go away!
$ _
What is a Problem?

- Everything is a problem, that is perceived as such by the user
- Developers must be able to take a user perspective
- Solution: Testing with real users!

Task: Email A Tale of Two Cities to arthur@ximian.com; Subject: 

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