Today, we’ll talk about testing – how to test software. The question is: How do we design tests? And we’ll start with functional testing.

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.

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

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.
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…

and grows…
Software is manifold and grows…

Software is manifold and grows…

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

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

Dijkstra’s Curse

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?

Formal Verification
Formal Verification

Areas missing might be: the operating system, the hardware, all of the world the system is embedded in (including humans!)
Verification can only find the absence of errors, but never their presence.

Areas missing might be: the operating system, the hardware, all of the world the system is embedded in (including humans!)

We might not be able to cover all Abstraction levels in all Konfigurationens, but we can do our best to cover as much as possible.

So, how can we cover as much behavior as possible?
Today, we’ll talk about testing – how to test software. The question is: How do we design tests? And we’ll start with functional testing.

Functional testing is also called “black-box” testing, because we see the program as a black box – that is, we ignore how it is being written in contrast to structural or “white-box” testing, where the program is the base.
Testing Tactics

- Tests based on spec
- Test covers as much specified behavior as possible

- Tests based on code
- Test covers as much implemented behavior as possible

Why Functional?

- Program code not necessary
- Early functional test design has benefits
  reveals spec problems • assesses testability • gives additional explanation of spec • may even serve as spec, as in XP
Why Functional?

- **Best for missing logic defects**
  Common problem: Some program logic was simply forgotten.
  Structural testing would not focus on code that is not there.

- **Applies at all granularity levels**
  unit tests • integration tests • system tests • regression tests

Functional

“black box”

Structural

“white box”

Structural testing can not detect that some required feature is missing in the code.
Functional testing applies at all granularity levels (in contrast to structural testing, which only applies to unit and integration testing).

Random Testing

- Pick possible inputs uniformly
- Avoids designer bias
  A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)
- But treats all inputs as equally valuable

One might think that picking random samples might be a good idea.

Abstrakt gesehen, ist Angry Birds dasselbe wie die Ariane: bei beiden geht es darum, ballistisch ein Ziel zu treffen – in unserem Fall zwei Schweine. (Sie ahnen nicht, wie lange ich gespielt habe, bis ich das hinbekommen habe – alles im Dienste der Wissenschaft!)
Wenn wir bei Angry Birds wieder abstrahieren, ist das Spiel eigentlich ganz einfach. Sie müssen nur zwei Dinge auswählen: Den * Winkel und die Kraft.

Diese beiden legen die Flugbahn fest. Die Frage ist: Können wir alle Flugbahnen testen?

Infinite Monkey Theorem
In unserem Fall sieht das so aus: Der Affe klickt wahllos durch die Gegend.

Wie lange dauert es, bis der Affe alle Flugbahnen durch hat? Für Winkel und Kraft gibt es jeweils $2^{32}$ different values.

Das sind dann 18 Trillionen verschiedene mögliche Abläufe...

Wenn ein Affe die alle ausprobieren soll, sagen wir 1 Minute pro Spiel, dann ist der Affe längst tot, bevor er fertig ist. Das Universum übrigens auch.

Ich kann aber…

Source: http://www.gadgets-club.com/happy-ipad-user

zwei Affen nehme, dann geht’s doppelt so schnell…

mit vier nochmal doppelt so schnell
Und wenn Sie 18 Trillionen Affen nehme, bekommen Sie in * einer Minute alle Flugbahnen.


Wir nehmen also einfach alles Wasser der Ozeane und machen daraus Affen (Affen bestehen größtenteils aus Wasser). Planet der Affen, sozusagen.

Wobei nun aber auf den untersten Affen bis zu 10 Kilometern Affen lasten, also 5 Tonnen –

Die Alternative zum Affen ist der Informatiker. Informatiker sind smart, und die können Programme sehr systematisch testen und analysieren.

Systematic Functional Testing

![Diagram of Systematic Functional Testing](image)

The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

- **Decompose system into independently testable features (ITF)**
- An ITF need not correspond to units or subsystems of the software
- For system testing, ITFs are exposed through user interfaces or APIs

Testable Features

![Diagram of Testable Features](image)
Testable Features

class Roots {
    // Solve \( ax^2 + bx + c = 0 \)
    public roots(double a, double b, double c) {
        // Result: values for x
        double root_one, root_two;
    }

    • What are the independently testable features?

Testable Features

• Consider a multi-function calculator
• What are the independently testable features?

Testable Features

The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)
Representative Values

- Try to select inputs that are especially valuable
- Usually by choosing representatives of equivalence classes that are apt to fail often or not at all

Needles in a Haystack

- To find needles, look systematically
- We need to find out what makes needles special

Systematic Partition Testing

We can think of all the possible input values to a program as little boxes ... white boxes that the program processes correctly, and colored boxes on which the program fails. Our problem is that there are a lot of boxes ... a huge number, and the colored boxes are just an infinitesimal fraction of the whole set. If we reach in and pull out boxes at random, we are unlikely to find the colored ones.

Systematic testing says: Let’s not pull them out at random. Let’s first subdivide the big bag of boxes into smaller groups (the pink lines), and do it in a way that tends to concentrate the colored boxes in a few of the groups. The number of groups needs to be much smaller than the number of boxes, so that we can systematically reach into each group to pick one or a few boxes. Functional testing is one way of partition testing, a way of drawing the orange lines so that, when one of the boxes within a orange group is a failure, many of the other boxes in that group may also be failures. Functional testing means using the program specification to draw pink lines.

(from Pezze + Young, “Software Testing and Analysis”, Chapter 10)
**Equivalence Partitioning**

<table>
<thead>
<tr>
<th>Input condition</th>
<th>Equivalence classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>one valid, two invalid</td>
</tr>
<tr>
<td></td>
<td>(larger and smaller)</td>
</tr>
<tr>
<td>specific value</td>
<td>one valid, two invalid</td>
</tr>
<tr>
<td></td>
<td>(larger and smaller)</td>
</tr>
<tr>
<td>member of a set</td>
<td>one valid, one invalid</td>
</tr>
<tr>
<td>boolean</td>
<td>one valid, one invalid</td>
</tr>
</tbody>
</table>

How do we choose equivalence classes? The key is to examine input conditions from the spec. Each input condition induces an equivalence class – valid and invalid inputs.

---

**Boundary Analysis**

- **Possible test case**
  - Test at lower range (valid and invalid), at higher range (valid and invalid), and at center

How do we choose representatives from equivalence classes? A greater number of errors occurs at the boundaries of an equivalence class rather than at the “center”. Therefore, we specifically look for values that are at the boundaries – both of the input domain as well as at the output.

---

**Example: ZIP Code**

- **Input:** 5-digit ZIP code
- **Output:** list of cities
- **What are representative values to test?**

(from Pezze + Young, “Software Testing and Analysis”, Chapter 10)
Valid ZIP Codes

1. with 0 cities as output
   (0 is boundary value)
2. with 1 city as output
3. with many cities as output

Invalid ZIP Codes

4. empty input
5. 1–4 characters
   (4 is boundary value)
6. 6 characters
   (6 is boundary value)
7. very long input
8. no digits
9. non-character data

“Special” ZIP Codes

- How about a ZIP code that reads
  
  `12345'; DROP TABLE orders; SELECT * FROM zipcodes WHERE 'zip' = '`
- Or a ZIP code with 65536 characters...
- This is security testing
Gutjahr’s Hypothesis

Partition testing is more effective than random testing.

---

Generally, random inputs are easier to generate, but less likely to cover parts of the specification or the code. See Gutjahr (1999) in IEEE Transactions on Software Engineering 25, 5 (1999), 661-667

---

Representative Values

The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

---

Model-Based Testing

The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)
Finite State Machine

As an example, consider these steps modeling a product maintenance process…
(from Pezze + Young, “Software Testing and Analysis”, Chapter 14)

Coverage Criteria

- **Path coverage**: Tests cover every path
  Not feasible in practice due to infinite number of paths

- **State coverage**: Every node is executed
  A minimum testing criterion

- **Transition coverage**: Every edge is executed
  Typically, a good coverage criterion to aim for
With five test cases (one color each), we can achieve transition coverage (from Pezze + Young, “Software Testing and Analysis”, Chapter 14)

Finite state machines can be used to model for a large variety of behaviors – and thus serve as a base for testing.

Here’s an example of a finite state machine representing an Account class going through a number of states. Transition coverage means testing each Account method once. (From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 14)

State-based Testing

- Protocols (e.g., network communication)
- GUIs (sequences of interactions)
- Objects (methods and states)
A decision table describes under which conditions a specific outcome comes to be. This decision table, for instance, determines the discount for a purchase, depending on specific thresholds for the amount purchased.

(from Pezze + Young, “Software Testing and Analysis”, Chapter 14)
This also applies to changing the other values, so adding additional test cases is not necessary in this case.
(from Pezze + Young, “Software Testing and Analysis”, Chapter 14)
Weyuker’s Hypothesis

The adequacy of a coverage criterion can only be intuitively defined.

Established by a number of studies done by E. Weyuker at AT&T. “Any explicit relationship between coverage and error detection would mean that we have a fixed distribution of errors over all statements and paths, which is clearly not the case”.

Learning from the past

To decide where to put most effort in testing, one can also examine the past – i.e., where did most defects occur in the past. The above picture shows the distribution of security vulnerabilities in Firefox – the redder a rectangle, the more vulnerabilities, and therefore a likely candidate for intensive testing. The group of Andreas Zeller at Saarland University researches how to mine such information automatically and how to predict future defects.

Pareto’s Law

Approximately 80% of defects come from 20% of modules

Evidence: several studies, including Zeller’s own evidence :-)

Model-Based Testing

The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

Deriving Test Case Specs

- Input values enumerated in previous step
- Now: need to take care of combinations
- Typically, one uses models and representative values to generate test cases

Combinatorial Testing

Many domains come as a combination of individual inputs. We therefore need to cope with a combinatorial explosion.
Combinatorial Testing

- Eliminate invalid combinations
  - IIS only runs on Windows, for example
- Cover all pairs of combinations
  - such as MySQL on Windows and Linux
- Combinations typically generated automatically
  - and – hopefully – tested automatically, too

Pairwise testing means to cover every single pair of configurations

In practice, such testing needs hundreds and hundreds of PCs in every possible configuration – Microsoft, for instance, has entire buildings filled with every hardware imaginable.

Source: http://www.ci.newton.ma.us/MIS/Network.htm
The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

Deriving Test Case Specs

- Function specification → Independently testable feature
- Independently testable feature → Model
- Model → Representative values
- Representative values → Test case specifications
- Test case specifications → Test case
- Test case → Test case specifications

Deriving Test Cases

- Implement test cases in code
- Requires building scaffolding – i.e., drivers and stubs
- Test case specifications → Test case
- Test case → Test case specifications

Unit Tests

- Directly access units (= classes, modules, components…) at their programming interfaces
- Encapsulate a set of tests as a single syntactical unit
- Available for all programming languages (JUnit for Java, CPPUNIT for C++, etc.)

Here’s an example for automated unit tests – the well-known JUnit
Running a Test

A test case…

1. sets up an environment for the test
2. tests the unit
3. tears down the environment again.

The environment provides the stubs such that a JUnit test case can work. The JUnit test case is the driver.

Testing a URL Class

As an example, consider parsing a URL

http://www.askigor.org/status.php?id=sample

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Host</th>
<th>Path</th>
<th>Query</th>
</tr>
</thead>
</table>

The setUp() and tearDown() functions set up the environment…

```java
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class URLTest extends TestCase {
    private URL askigor_url;

    // Create new test
    public URLTest(String name) { super(name); }

    // Assign a name to this test case
    public String toString() { return getName(); }

    // Setup environment
    protected void setUp() {
        askigor_url = new URL("http://www.askigor.org/" + "status.php?id=sample");
    }

    // Release environment
    protected void tearDown() { askigor_url = null; }
```
This functional test can be used as a specification!

public void testProtocol() {
    assertEquals(askigor_url.getProtocol(), "http");
}

// 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");
}

// Set up a suite of tests
public static Test suite() {
    TestSuite suite = new TestSuite(URLTest.class);
    return suite;
}

// Main method: Invokes GUI
public static void main(String args[]) {
    String[] testCaseName = {
        URLTest.class.getName() 
    };
    // junit.textui.TestRunner.main(testCaseName);
    junit.swingui.TestRunner.main(testCaseName);
    // junit.awtui.TestRunner.main(testCaseName);
}

JUnit comes with a GUI – and is frequently integrated in programming environments
The main steps of a systematic approach to functional program testing (from Pezze + Young, "Software Testing and Analysis", Chapter 10)