And anything of weight smaller than 1000 kg can be expected to be endured.

\[ \epsilon \]

Termination, Divergence

Semantics

Deterministic Programs

\[ \delta \]

differ only by a small amount

-environments of an input,

For sufficiently small

Testing in the Development Process

Model-Based Testing

Consider a continuous function, e.g. the one to the right:

\[ f(x) = \begin{cases} 
    x^2 & \text{if } x < 0 \\
    0 & \text{if } x = 0 \\
    1-x & \text{if } x > 0 
\end{cases} \]

\[ : \text{multiple "neighbouring" inputs trigger the error.} \]

Point error

Range error

\[ \text{Expected outcomes: Test} \]

Statistical

\[ \text{Approaches:} \]

Generic

\[ \text{some values to others.} \]

Software behaviour is (in general, without extra information)

\[ \text{we may yield} \]

\[ \text{on good test cases} \]

\[ \text{Some more vocabulary} \]

Content

Recall

Dr. Bernd Westphal

Prof. Dr. Andreas Podelski,

2013

Comparison

Information flow development

Information flow examination

Review

Examination result is

\[ \text{difference is detected} \]

Only

\[ \text{test cases (that' s a lot!}, \]

\[ \text{passed the test} \]

\[ \text{Intuition: no error has been discovered.} \]

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: an error has been discovered.

\[ \text{Alternative: test item} \]

Intuition: no error has been discovered.
Specific Testing Notions

How are the test cases chosen?

• Considering only the specification (black-box or function test).
• Considering the structure of the test item (glass-box or structure test).

How much effort is put into testing?

• Execution trial—does the program run at all?
• Throw-away test—invent input and judge output on-the-fly (→ "rumprobieren”),
• Systematic test—somebody (not author!) derives test cases, defines input/soll, documents test execution.

Experience: In the long run, systematic tests are more economic.

• Complexity of the test item:
  • Unit test—a single program unit is tested (function, sub-routine, method, class, etc.)
  • Module test—a component is tested,
  • Integration test—the interplay between components is tested.
  • System test—tests a whole system.

Specific Testing Notions Cont'd

Which property is tested?

• Function test—functionality as specified by the requirements documents,
• Installation test—is it possible to install the software with the provided documentation and tools?
• Recommissioning test—is it possible to bring the system back to operation after operation was stopped?
• Availability test—does the system run for the required amount of time without issues,
• Load and stress test—does the system behave as required under high or highest load?... under overload? "Hey, let's try how many game objects can be handled!"—that's an experiment, not a test.
• Resource tests—response time, minimal hardware (software) requirements, etc.
• Regression test—does the new version of the software behave like the old one on inputs where no behaviour change is expected?

Which roles are involved in testing?

• Inhouse test—only developers (meaning: quality assurance roles),
• Alpha and beta test—selected (potential) customers,
• Acceptance test—the customer tests whether the system (or parts of it, at milestones) is acceptable.

Some more vocabulary

Choosing Test Cases

Generic requirements on good test cases

Approaches:

• Statistical testing
• Expected outcomes: Test Oracle:/-/
• Habitat-based
• Glass-Box Testing
• Statement/Branch term coverage

Conclusions from coverage measures

When To Stop Testing?

• Model-Based Testing
• Testing in the Development Process
• Formal Program Verification
• Deterministic Programs
• Syntax, Semantics, Termination, Divergence

Choosing Test Cases

How to Choose Test Cases?

A first rule-of-thumb: "Everything, which is required, must be examined/checked. Otherwise it is uncertain whether the requirements have been understood and realised."

(1 Ludewig and Lichter, 2013)

In other words:
• Not having at least one (systematic) test case for each (required) feature is (grossly?) negligent. (Dt.: (grob?) fahrlässig).

In even other words:
Without at least one test case for each feature, we can hardly speak of software engineering.

Good project management: document for each test case which feature(s) it tests.
Statistical Testing: Discussion

One Approach: Statistical Tests

The idea of statistical testing is that we should exploit that knowledge to optimise the number of test cases. But it is perfectly reasonable to test representatives for untypical user behaviours. We should aim for good test cases, i.e. representative for specific use cases, such as "C50", "E1" coins in the vending machine.

When To Stop Testing?

Testing needs a method to determine whether a test suite is sufficient, i.e., we should exploit the knowledge that we have gained. Testing usually stops when the tester decides that it has found all the errors or that the software behaves well for all the inputs considered.

What Else Makes a Test Case a Good Test Case?

A test case is a representative of a whole class of inputs, i.e., it does not test what other test cases also test. However, it does test what other test cases also test, i.e., it tests the inputs that other test cases test. Thus: The wish for representative test cases is perfectly reasonable.

Staged Testing

There is a high risk for small-range errors. If they live in their "natural habitat", we certainly know there is an error, good test cases are available, etc. If an error is found, execute test suite for requirement "does not crash", but can be difficult in general.

Conclusions

Coverage measures, on good test cases, are perfect, i.e., they are representative for all classes of inputs. Thus: The wish for representative test cases is perfectly reasonable.
Recall: A test case is a pair $(\text{In}, S_{\text{oll}})$ with proper expected (or “soll”) values.

- In an ideal world, all “soll”-values are defined by the (formal) requirements specification and effectively pre-computable.
- In this world, the formal requirements specification may only reflectively describe acceptable results without giving a procedure to compute the results.
- There may not be a formal requirements specification, e.g.
  - “the game objects should be rendered properly”,
  - “the compiler must translate the program correctly”,
  - “the notification message should appear on a proper screen position”,
  - “the data must be available for at least 10 days”,
  - etc.

Then: need another instance to decide whether the observation is acceptable.

The testing community prefers to call any instance which decides whether results are acceptable a (test) oracle.

I’d prefer not to call automatic derivation of “soll”-values from a formal specification an “oracle”... (-)

("person or agency considered to provide wise and insightful [..] prophetic predictions or precognition of the future, inspired by the gods." says Wikipedia)

Some traditional popular belief on software error habitat:
- Software errors (seem to) enjoy range boundaries, e.g.
  - 0, 1, 27 if software works on inputs from $[0, 27]$,
  - $-1, 2^8$ for error handling,
  - $-2^{31} - 1, 2^{31}$ on 32-bit architectures,
  - boundaries of arrays (first, last element),
  - boundaries of loops (first, last iteration),
  - etc.
- special cases of the problem (empty list, use-case without actor, ...),
- special cases of the programming language semantics,
- complex implementations.

→ Good idea: for each test case, note down why it has been chosen.
For example, “demonstrate that corner-case handling is not completely broken”.

Choosing Test Cases Habitat-based

- Choosing Test Cases Habitat-based
- Choosing Test Cases Habitat-based
- Choosing Test Cases Habitat-based
- Choosing Test Cases Habitat-based
- Choosing Test Cases Habitat-based
- Choosing Test Cases Habitat-based
## Term Coverage

**Definition:** The term coverage is a type of code coverage that measures the percentage of the source code that is covered by the test cases. It includes all the terms in the code, which means all the expressions, functions, variables, and other code elements that are evaluated during the execution of the program.

### Calculation

The term coverage can be calculated using the following formula:

\[
\text{Term Coverage} = \frac{\text{Number of terms covered}}{\text{Total number of terms}} \times 100\%
\]

### Importance

Term coverage is important because it ensures that all the terms in the code are tested, which helps in identifying any potential bugs or issues in the code. It is particularly useful in complex systems where the behavior of individual code terms is critical.

### Challenges

One of the challenges with term coverage is that it can be computationally expensive to calculate, especially for large programs. Additionally, it can be difficult to accurately measure term coverage in languages with dynamic types or in cases where terms are evaluated conditionally.

### Applications

Term coverage is commonly used in software testing, specifically in model-based testing, where it is used to verify that all terms in the specification are covered by the test cases.

### Examples

Consider the following code snippet:

```python
x = 5
y = x + 2
z = y / x
```

- **Terms:** `x`, `y`, `z`, `+`, `/`
- **Covered Terms:** `x`, `y`, `z`, `+`, `/`
- **Term Coverage:** 100%

### Conclusion

Term coverage is a valuable tool in software testing, ensuring that all terms in the code are tested. It helps in identifying potential issues and improves the overall quality of the software.
There need to be defined criteria for when to stop testing; project planning should consider these criteria (and previous experience).

Possible "testing completed" criteria:

- all (previously) specified test cases have been executed with negative result, (Special case: All test cases resulting from a certain strategy, like maximal statement coverage have been executed.)
- testing effort time sums up to \( x \) (hours, days, weeks),
- testing effort sums up to \( y \) (any other useful unit),
- \( n \) errors have been discovered,
- no error has been discovered during the last \( z \) hours (days, weeks) of testing,

Values for \( x \), \( y \), \( n \), \( z \) are fixed based on experience, estimation, budget, etc.

Of course: not all criteria are equally reasonable or compatible with each testing approach.

Another possible "testing completed" criterion:

- The average cost per error discovery exceeds a defined threshold \( c \).

\[
\text{cost per discovered error} = \frac{\text{cost}}{\text{number of discovered errors}}
\]

Value for \( c \) is again fixed based on experience, estimation, budget, etc.

Some more vocabulary

Choosing Test Cases

Generic requirements on good test cases

- Approaches:
  - Statistical testing
  - Expected outcomes: Test Oracle: /-
  - Habitat-based
  - Glass-Box Testing
  - Statement/Branch coverage
  - Conclusions from coverage measures

When To Stop Testing?

Model-Based Testing

Testing in the Development Process

Formal Program Verification

Deterministic Programs

Syntax, Semantics, Termination, Divergence
Testing in The Software Development Process

Role Differences
- Developer and Tester
  - Semantics
  - Syntax

In Formal Program Verification

Model-Based Testing

When To Stop Testing?

Testing in The Development Process

- A skeletal or special-purpose implementation of a software module, used to develop or test a module that calls or is otherwise dependent on it.
- A computer program statement substituting for the body of a software module used to develop or test a module that calls or is otherwise dependent on it.

Testing Approaches:
- Glass-Box Testing
- Habitat
- Statistical Testing

Test Vocab:
- Coverage
- Branch
- Statement

Choosing Test Cases

Some more vocabulary
- Oracle
- Glue Logic

Summary

Testing

Vocabulary

Test Gear

Test Plan

Preparation

Execution

Analysis

Evaluation

Report

Protocol

Test Cases

Choosing Test Cases

Some more vocabulary

Test Gear

Test Plan

Preparation

Execution

Analysis

Evaluation

Report

Protocol

Test Cases
There is a vast amount of literature on how to choose test cases. A good starting point:
- at least one test case per feature,
- corner-cases, extremal values,
- error handling, etc.

Glass-box testing considers the control flow graph, defines coverage measures.

Other approaches:
- statistical testing, model-based testing,
- Define criteria for "testing done" (like coverage, or cost per error).

Process: tester and developer should be different persons.

There are more approaches to code quality assurance than (just) testing.
- For example, program verification.

References: