Software Safety

Dynamic Testing

Prof. Dr.-Ing. Patrick Mäder, M.Sc. Martin Rabe

Contents

Dynamic Testing

Dynamic Testing within V&V

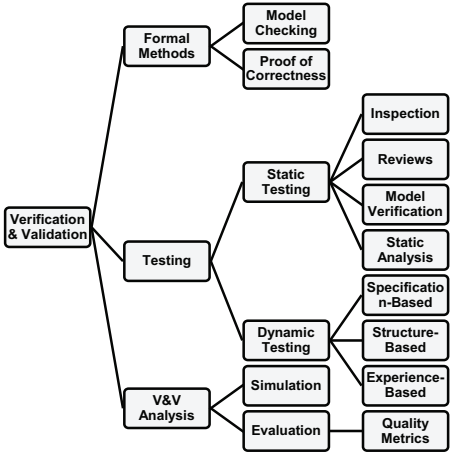


Figure A.1 — Hierarchy of Verification and Validation activities

[ISO/IEC/IEEE 29119-1:2013. Standards catalogue. International Organization for Standardization. September 2013]

Standards Require Testing: Quotations from IEC 62304

- 5.5.2: The manufacturer shall establish **strategies, methods and procedures for verifying each software unit**. Where verification is done by testing, the test procedures shall be evaluated for correctness.

- 5.6.3: The manufacturer shall **test the integrated software items** in accordance with the integration plan and document the results.

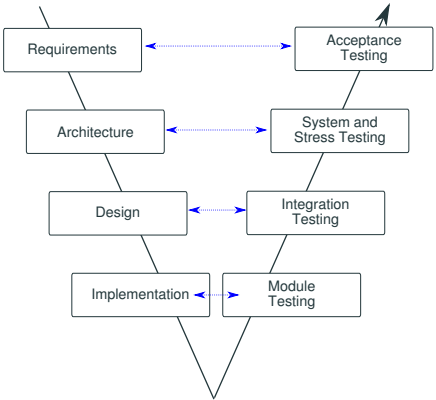
- 5.7.1: The manufacturer shall establish and perform a set of **tests, expressed as input stimuli, expected outcomes, pass/fail criteria and procedures**, for conducting software system testing, such that all software requirements are covered.

Typical Test Phases

Type	Carried out by ...	Purpose
Module	Programmer	Does the module meet its requirements in isolation in an artificial environment?
Integration	Integration Testers	Does the module interact correctly with its peers when run in an artificial environment?
System	System Testers	Does the system operate correctly with these modules present?
Stress	System Testers	Does the system operate correctly under extreme or impulse loads?
Regression	Regression Testers	Does the system still operate to specification after the last set of changes?
Acceptance	Customer	Does the system meet the customer's requirements?

[Hobbs, C. (2019). Embedded Software Development for Safety-Critical Systems (2nd ed.). CRC Press.]

Module Testing



Module testing detects *faults* in the code by looking for the *errors* they produce.

Test quality is often measured by code and branch coverage.

Path coverage is normally impossible.

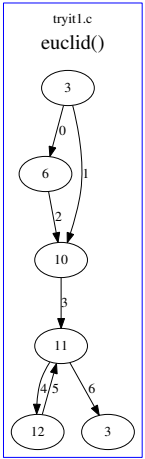
gcov is a neat tool for C programs:

```
Lines executed:100.00% of 20
Branches executed:100.00% of 16
Taken at least once:93.75% of 16
Calls executed:100.00% of 3
```

IEC/ISO 29119 and IEC 61508 call it “unit test”,
ISO 26262 and IEC 62304 call it “module test”,
EN 50128 calls it “component test”

Typical Code Coverage Criteria (1/2)

How many test cases are needed to cover this source code?




```
1  int euclid(int m, int n) {
2      int     r;
3      if (n > m) {  // set m >= n
4          r = n;
5          n = m;
6          m = r;
7      }
8      r = m % n;
9      while (r != 0) {
10         m = n;
11         n = r;
12         r = m % n;
13     }
14     return n;
15 }
```

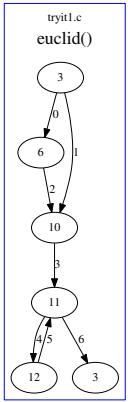
[Hobbs, C. (2019). Embedded Software Development for Safety-Critical Systems (2nd ed.). CRC Press.]

Typical Code Coverage Criteria (2/2)

- **Entry point**: 1 test case
- **Statement**: 1 test case
- **Branch**: 2 test cases
- **Basis path**: 3 test cases
  based on control flow graph → McCabe's cyclomatic complexity
  $M = E - N + 2P = 7 - 6 + 2 \times 1 = 3$ tests
- **Modified condition/decision coverage (MC/DC)**
  - Each entry and exit point is invoked
  - Each decision takes every possible outcome
  - Each condition in a decision takes every possible outcome
  - Each condition in a decision is shown to independently affect the outcome of the decision

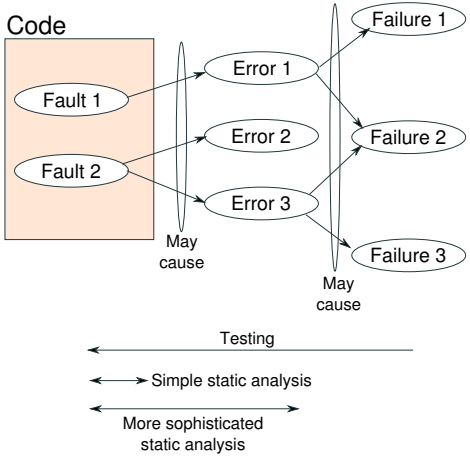
  → DO-178C and ISO 26262 prescribe MC/DC coverage for the highest integrity levels

edges E=7, nodes N=6, components/subgraphs P=1



[Hobbs, C. (2019). **Embedded Software Development for Safety-Critical Systems (2nd ed.)**. CRC Press.]

Testing for Failures – Fixing Faults



[Hobbs, C. (2019). **Embedded Software Development for Safety-Critical Systems (2nd ed.)**. CRC Press.]

- Testing detects **failures** but only **faults** can be fixed. Testing is unlikely to find all failures.
- Once a bug is fixed, testing reproduces the original conditions and checks that the failure no longer occurs.

Test Coverage vs. Fault Coverage

- How does test coverage relate to fault coverage? Does a 95% code coverage mean 95% of the faults are detected?

- [Briand and Pfahl 99] study the impact of test-coverage on defect-coverage and conclude:
  - *"The study outcomes do not support the assumption of a causal dependency between test-coverage and defect-coverage [fault-coverage]."*

→ Test-coverage $\neq$ fault-coverage

[Briand, Lionel, and Dietmar Pfahl: **Using simulation for assessing the real impact of test coverage on defect coverage**. 10th ISSRE, IEEE, 1999.]

Test Coverage vs. Fault Coverage: Example

- Illustrating example

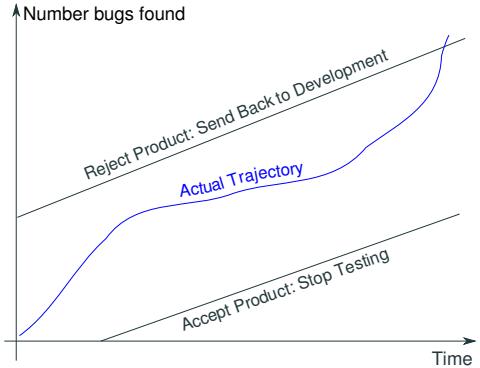
```
int v[100];
void f(int x)
    {
    if (x > 99) x = 99; // check for overflow
    v[x] = 0;
    }
```

  - Two sample test cases with $x = 50$ and $x = 100$ would provide 100% code, branch, basis path, and MC/DC coverage
  - → However, these tests would not detect the problem created by, e.g., $x = -2$

→ The standards prescribe code and branch coverage but do not focus on fault coverage

Stop Testing Based on Failure Rate

- **The BIG testing question**: When should/can I stop testing?
- Is this conventional approach (see figure) acceptable for safety-critical systems?



[Hobbs, C. (2019). **Embedded Software Development for Safety-Critical Systems (2nd ed.)**. CRC Press.]

Stop Testing Based on Confidence [IEC61508] (1/2)

- Running N random tests will give a confidence M that the system will fail at most 1 time in h

  $M = 1(1\frac{1}{h})^N$ [IEC 61508-7, appendix D]

- Tests must be representative of real usage and no failures must occur

Number of Tests N	Confidence Level M
25 000	0.9179
35 000	0.9698
45 000	0.9889
55 000	0.9959
65 000	0.9985
75 000	0.9994
85 000	0.9998
95 000	0.9999

[Hobbs, C. (2019). Embedded Software Development for Safety-Critical Systems (2nd ed.). CRC Press.]

e.g., Confidence that system will perform correctly 9,999 times out of 10,000.

Stop Testing Based on Confidence [IEC61508] (2/2)

- How long do I need to test to be $(1-\alpha)$% confident that I meet SIL n?

$$T = \frac{-\ln(\alpha)}{\lambda}$$

where λ is the probability of failure per hour [IEC 61508-7, appendix D]

- Tests must be representative of real usage and no failures must occur

For SIL	Confidence Level $1-\alpha$	Test Duration T
1	95%	34 years
	98%	45 years
2	95%	342 years
	98%	446 years
3	95%	3,417 years
	98%	4,463 years
4	95%	34,174 years
	98%	44,627 years

[Hobbs, C. (2019). Embedded Software Development for Safety-Critical Systems (2nd ed.). CRC Press.]

Particularly useful for programs that do not terminate

Stop Testing based on Reduced Risk

- If testing is based on IEC 29119's Risk-Based Testing, have the risks been sufficiently mitigated?

- "The Test Completion Process [...] is performed **when agreement has been obtained that the testing activities are complete**. It will be performed to complete the testing carried out at a specific test phase (e.g. system testing) or test type (e.g. performance testing) and to complete the testing for a complete project." [IEC 29119-2]

Risk-based Testing

Testing Limitations

- **Limited Observability**: the tester cannot see how multiple threads of execution are interweaving during a test.
- **Limited Reproducibility**: the tester cannot control the system sufficiently to reproduce the problem.

- Unless exhaustive, testing cannot prove the absence of faults – and exhaustive testing is not practical

- **Testing is a sampling technique**: tests only check a very small percentage of the paths through the code and results must be handled statistically.

[fayrowland, Pixabay, 2005]

IEC/ISO 29119

[**ISO/IEC/IEEE 29119-1:2013**. Standards catalogue. International Organization for Standardization. September 2013]

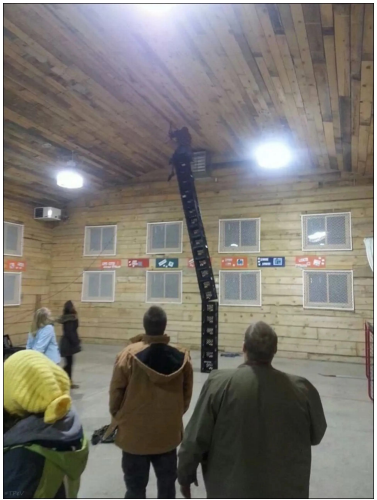
- The oftware test standard IEC/ISO 29119 recognizes that testing cannot be exhaustive and must be risk-based
- Risk examples
  - not satisfying regulatory and/or legal requirements
  - not meeting contractual obligations
  - unsuccessful progress and completion of the project

Quotations from IEC 29119-1

- It is impossible to test a software system exhaustively, thus testing is a sampling activity. A variety of testing concepts [...] aid in choosing an appropriate sample to test [...]. A key premise of this standard is the idea of performing the optimal testing within the given constraints and context using a risk-based approach.
- This is achieved by identifying the relative value of different strategies for testing, in terms of the risks they mitigate for the stakeholders of the completed system, and for the stakeholders developing the system.
- When performing risk-based testing, risk analysis is used to identify and score risks, so that the perceived risks in the delivered system [...] can be scored, prioritized and categorized and subsequently mitigated.

Risk-Based Testing

- “Dynamic test execution is a risk mitigation activity.”
- IEC 29119 gives examples of risks:
  - not satisfying regulatory and/or legal requirements
  - not meeting contractual obligations
  - unsuccessful progress and completion of the project



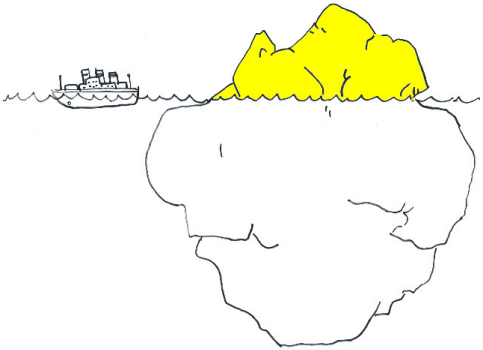
[PIXIMUS: Actions Shots That Were Taken Seconds Before A Disaster, 2015.]

IEC 29119 and Safety Critical Developments

- EN 50128 implies requirements-based testing: paragraph 7.5.4.7
- ISO 26262-6 highly recommends requirements-based testing: table 10

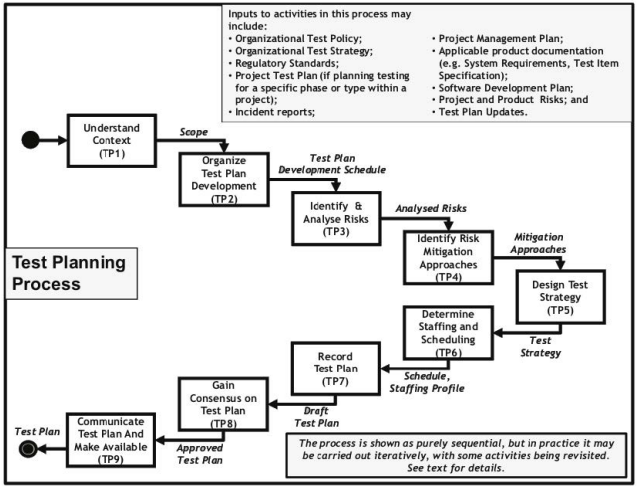
- But what is the safety requirement?
  - **Hazard**: the iceberg
  - **Risk**: ship will run into it
  - **Mitigation**: paint the iceberg yellow

- Do we test that the iceberg has been painted yellow or that we have mitigated the risk?



[Hobbs, C. (2019). **Embedded Software Development for Safety-Critical Systems (2nd ed.)**. CRC Press.]

IEC 29119: Test Planning



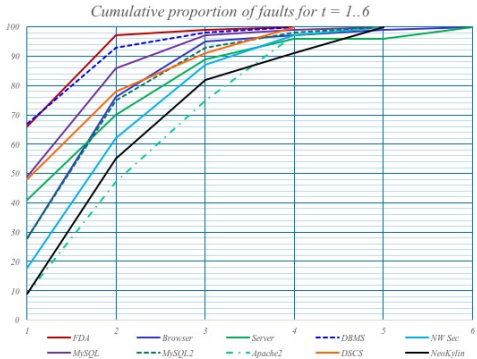
[ISO/IEC/IEEE 29119-1:2013. Standards catalogue. International Organization for Standardization. September 2013]

Combinatorial Testing

Combinatorial Testing (N-wise or t-way Testing)

- tests all possible discrete t-way (N-wise) combinations of input parameters to a system
- using carefully chosen test vectors that "parallelize" the tests of parameter combinations → 20x–700x reduction in test set size

→ **pseudo-exhaustive testing**



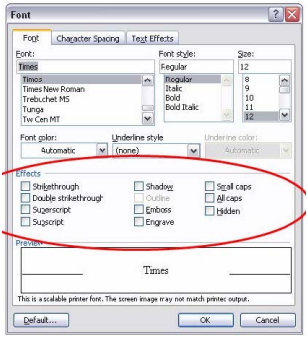
[NIST: **Combinatorial Testing**, project overview, 2021.]

- Percentage of errors triggered by t-way interactions. Most faults tend to be triggered by single values up to three-way interactions.

Combinatorial Testing: Simple Example

- 10 on–off effects → $2^{10} = 1,024$ tests
- **Restriction to 3-way interactions**: $\binom{10}{3} = 120$ interactions × 8 settings = 960 combinations
- all triples can be combined in only 13 tests

  NP hard problem with good algorithmic approximation available



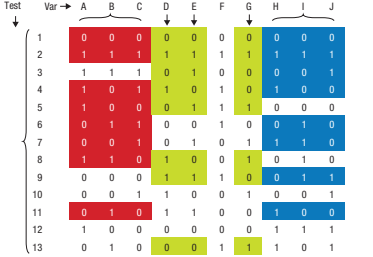


Figure A. Testing with a three-way covering array. The tests must cover 10 binary variables with two values each, which would take 2^{10}, or 1,024, tests to cover exhaustively. However, testing with the three-way covering array takes only 13 tests (rows). Highlighted numbers correspond to the eight possible three-way combinations in each set of three columns.

left: [Rick Kuhn: **Slides: Combinatorial Methods in Software Testing**. CASD, 2016.], right: [Hagar, Jon D., et al: **Introducing combinatorial testing in a large organization**. Computer 48.4 (2015): 64-72.]

Combinatorial Testing: Number of Necessary Tests

- Number of test cases grows proportional to

$$\nu^t log(n)$$

  - where:
    - n is the number of input parameters
    - ν is the number of possible values per parameter
    - t is the number of parameter interactions to cover (t-way)
- Tests increase logarithmically with the number of parameters → good – even very large test problems tractable (e.g., 200 parameters)
- Tests increase exponentially with interaction strength t → bad – select small, but representative value

Combinatorial Testing: More Realistic Example

- Suppose a system with 34 on-off switches as input to a software
- 34 switches $= 2^{34} = 1.7x10^{10}$ possible inputs $=$ **17 billion tests**
- For 3-way interactions: only **33 tests** needed
- For 4-way interactions: only **85 tests** needed

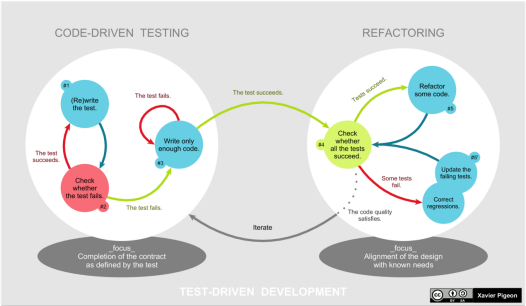
- Find more examples and case studies: [HERE] – and tooling [HERE]

[Rick Kuhn: **Slides: Combinatorial Methods in Software Testing**. CASD, 2016.]

Test-Driven Development

Test-Driven Development (TDD)

- Fast, repeatable and consistent testing
- Acceptance metrics are clearly defined
- Test cycle is fully visible and provides detailed test results
- Test results are easy to interpret
- Substantial tool support and integration into existing workflows



[Wikipedia: Test Driven Development, 2022]

TDD Benefits

- TDD code **takes longer to develop** *...about 16% longer* [George and Williams2004]
- TDD **code is of higher quality** *...it passed 18% more independent black-box tests* [George and Williams2004]
- TDD tests have a **higher code coverage** than tests written after coding
- TDD results in **more predictable development times**
- TDD improves **job satisfaction** for programmers

[George, Boby, and Laurie Williams: **A structured experiment of test-driven development**. Information and software Technology 46.5 (2004): 337-342.]

TDD vs. Inspection: Question Simple Study Results

- A more recent study [Wilkerson et al. 2011] ...compares *"the software* ***defect rates and implementation costs*** *of [...]* ***code inspection and test-driven development****. The [...] group turned in code that had fewer defects than the TDD group, [...], at the p=0.05 level of statistical significance [...]"*
  - → *code inspection is more effective than TDD at reducing defects*
  - → *TDD was no more effective [...] than traditional programming methods*

[Wilkerson, Jerod W., Jay F. Nunamaker, and Rick Mercer: **Comparing the defect reduction benefits of code inspection and test-driven development**. IEEE transactions on software engineering 38.3 (2011): 547-560.]

TDD vs. Inspection: Question Simple Study Results

- A more recent study [Wilkerson et al. 2011] ...compares *"the software* ***defect rates and implementation costs*** *of [...]* ***code inspection and test-driven development****. The [...] group turned in code that had fewer defects than the TDD group, [...], at the p=0.05 level of statistical significance [...]"*
  - → *code inspection is more effective than TDD at reducing defects*
  - → *TDD was no more effective [...] than traditional programming methods*

→ **But wait, that's only the abstract!** – these effects were mainly observed for **"adjusted" defect counts**

- meaning that the "inspection" students get credited not only for the bugs they fix, they also get credited for all the bugs they didn't fix, that were found during inspection

[Adobe Stock]

[Wilkerson, Jerod W., Jay F. Nunamaker, and Rick Mercer: **Comparing the defect reduction benefits of code inspection and test-driven development**. IEEE transactions on software engineering 38.3 (2011): 547-560.]

Automated Testing

Problems in Manual Testing

- How programmers spend their effort
  - Understanding the problem 5%
  - Finding a solution 10%
  - Implementation 20%
  - Testing 5%
  - **Debugging 60%**
- “Fixing a bug is usually pretty quick but finding it is a nightmare.” [Fowler+‘99]

→ Increase test effort

- Testing with “print” statement in code
  - Continuous program changes
  - Time consuming analysis of results

  → takes too long, problems potentially not discovered
- Observations
  - Tests need to be modular
    - External to the tested module
  - Tests need to be „self-analyzing“
    - Test program compares observed results with expected results

Automated Testing

- Reduced effort for
  - Writing tests
  - Maintaining tests
  - Activate and deactivate tests
  - Orchestrating tests
  - Test **execution and analysis**
- Testing becomes possible in shorter intervals
  - Less potential failures between test runs
  - Recent changes are still present → faster fault localization

→ **Faster development**

Continuous Testing Origins

- **Observation**
  - The shorter the interval between change and test, the quicker the fault can be localized
- **Approach** [Saff&Ernst'03]
  - Tool triggers tests in the background preceding every change (inspired by Eclipse's continuous compilation)
  - Programmer focusses on development
    - Only may react to non-passed tests rather than actively testing

Reducing wasted development time via continuous testing

David Saff Michael D. Ernst

MIT Computer Science & Artificial Intelligence Lab
200 Technology Square
Cambridge, MA 02139 USA
{saff,mernst}@csail.mit.edu

Abstract

are broken during maintenance or refactoring. Intuition suggests that a regression error that persists uncaught for a long

eclipse Technology eXchange Preliminary Version

Continuous Testing in Eclipse

David Saff Michael D. Ernst

saff@mit.edu, mernst@mit.edu
MIT Computer Science & Artificial Intelligence Lab
Cambridge, MA, USA

Abstract

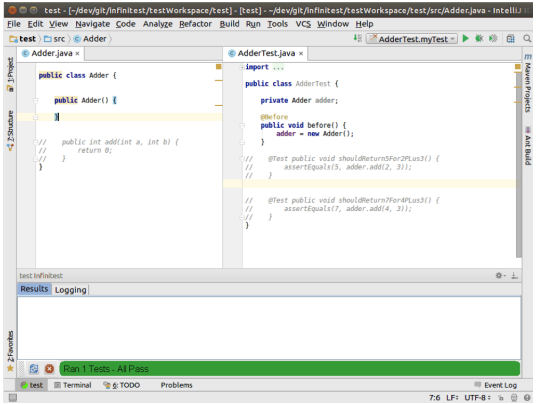
Continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background, providing rapid feedback about test failures as source code is edited. It is intended to reduce the time and energy required to keep code well-tested, and to prevent regression errors from persisting uncaught for long periods of time.

This paper reports on the design and implementation of a continuous testing feature for Java development in the Eclipse development environment. Our challenge was to generate and display a new kind of feedback (asynchronous notification of test failures) in a way that effectively reuses Eclipse's extensible architecture and fits

[Saff, David, and Michael D. Ernst: **Continuous testing in Eclipse**. Electronic Notes in Theoretical Computer Science 107 (2004): 103-117.]

IDE Integration – Infinitest & IntelliJ & jUnit

- **Unintrusive feedback**
  - IDE shows testing status and contains “Problem View”
  - Indicators highlighting failed tests
  - jUnit output only visible on request (e.g., stack trace)
- **Evaluation** [Saff&Ernst’04]
  - **Quantitative** [% correctness]
    - No tool group - 27%
    - Continuous compilation group - 50%
    - Continuous testing group - 78%
  - **Qualitative**: not distracted, would recommend



[Saff, David, and Michael D. Ernst: **An experimental evaluation of continuous testing during development**. ACM SIGSOFT Software Engineering Notes 29.4 (2004): 76-85.]

Potential Test Prioritization Strategies

- Most recent failures first
  - Tests that have failed most recently are ordered first.
- Quickest test first
  - Tests are ordered in increasing runtime; tests that complete the fastest are ordered first.
- No reordering
  - Tests are run in the default JUnit order on every change.
- Round robin
  - Like “No reordering”, but if testing is interrupted, start testing again at the interrupted test and wrap around to the beginning.
- Most frequent failures first
  - Tests that have failed most often (have failed during the greatest number of previous runs) are ordered first. Random: Tests are run in random order, without repetition.

[Saff, D. and M. D. Ernst: **Reducing wasted development time via continuous testing**. 14th ISSRE, 2003, pp. 281–292]

Continuous Testing and Continuous Delivery

- Traditional testing processes are too slow
  - Development iteration months → weeks → days (Agile, DevOps, and Continuous Delivery)

  → Need to extend test automation efforts
- More automation not necessarily gives more insight into associated risks
  - Tests are needed to assess whether risk related with release is business accepted
  - Test shall be designed based on the business's tolerance for risks related to, e.g., security, performance, reliability, and compliance
  - In addition to unit tests there is a need for a broader suite of tests

- "The goal of continuous testing is to provide **fast and continuous feedback** regarding the level of **business risk** in the latest build or release candidate."

Summary

Summary

- System testing cannot be exhaustive, since
  - the set of states cannot be computed
  - the system cannot be put into any particular state to start the test
  - the outcome of a particular test cannot be checked: things happen too fast
- Testing will not find Heisenbugs or, if it does, they will appear as unreproducible failures

→ We must accept testing as a statistical activity and interpret its results in that way

- **But testing is essential** and can also identify areas on which other validation techniques should focus
  - "Beware of bugs in the above code; I have only proved it correct, not tried it." [Donald Knuth]

Questions?