# Mutation Testing at SAAB

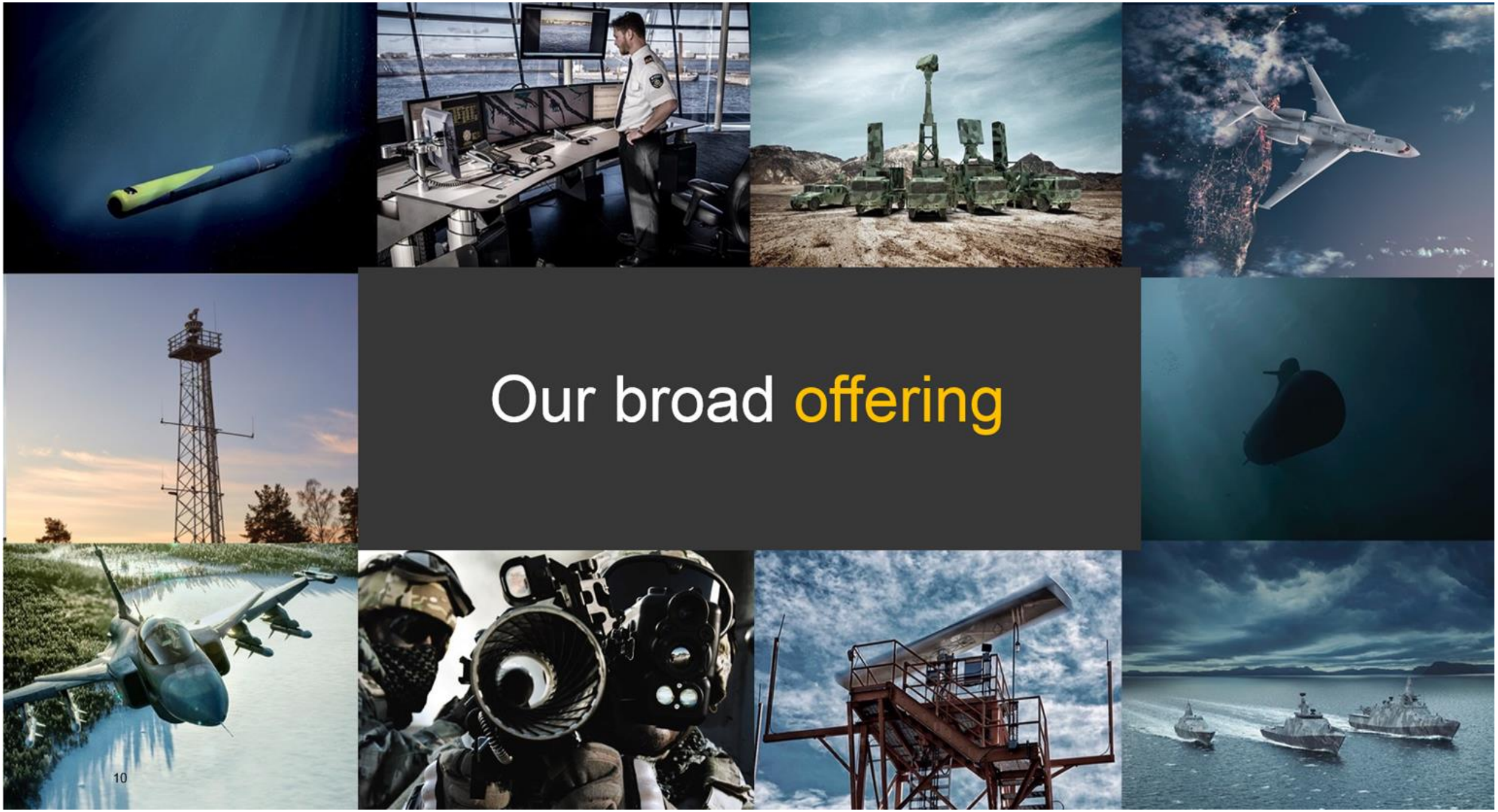
**Niklas Pettersson** niklas.pettersson2@saabgroup.com
**Joakim Brännström** joakim.k.brannstrom@saabgroup.com

# Agenda

- What is SAAB? (~25 min)
  - Some PR-videos, our products and more.
  - Safety-critical software.
- Why use mutation testing? (~20 min)
  - Practical examples.
  - Problem with code coverage.
  - What is mutation testing?
- Pitfalls of Mutation testing (~30 min)
  - From academic to industry.
- Short tool introduction (~10 min)
  - Dextool Mutate (will be used in the laboration).
- Questions and wrap-up (~ 5min)

# In 1937 we took off

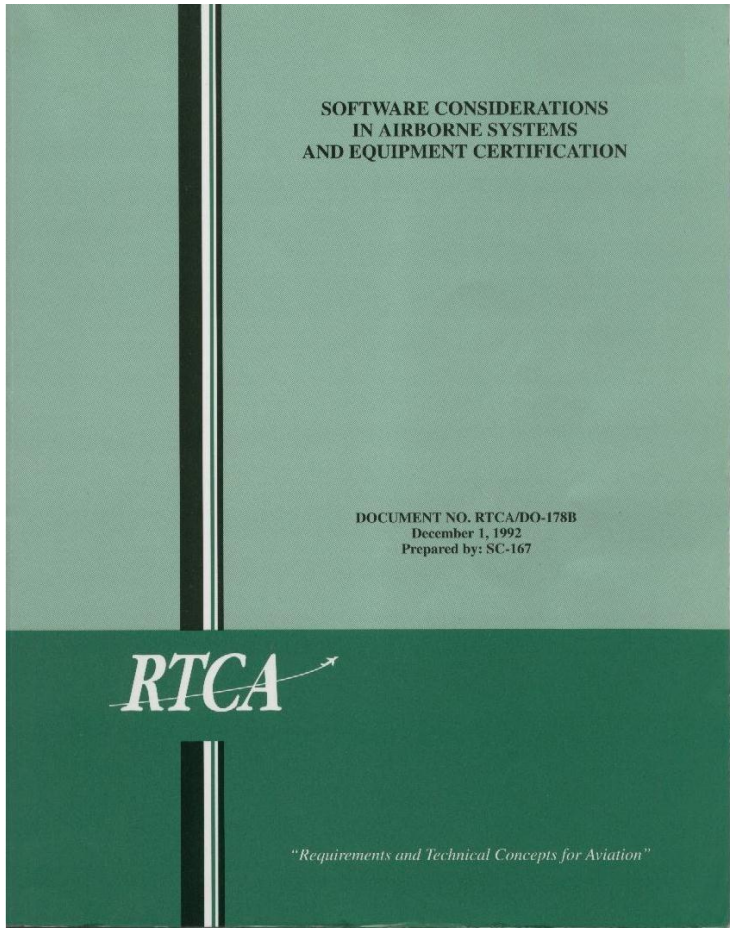
6:07


Our broad offering

# FAA RTCA/DO-178C

Provides detailed guidelines for the production of software for airborne systems:

- Objectives for the life cycle processes.
- Activities and design considerations for achieving those objectives.
- Descriptions of the evidence indicating the objectives have been satisfied.



# Responsibility

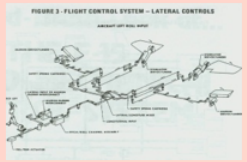
- Saab has a **certificate** issued by the Military Aviation Safety Inspectorate that allows us to have an organization to design and build military aircrafts.
- Saab is responsible for the **safety** of the aircraft, including the software.
- Other companies are not trusted to be responsible for the safety (e.g. external authority signs the software development plan).
- In case of an accident, Saab will be **reviewed:**
  - Saab must be able to show that we did enough to avoid the failure.
  - Saab will be measured against the best practices of the aviation industry. RTCA/DO-178B/C is **best practice** for software.

# Design Assurance Levels (DAL)

| Category | Failure Condition | DAL | Objectives |
|---|---|---|---|
| Catastrophic | Prevented flight and landing  | A | 67 |
| Hazardous | Injury (potentially fatal)  | B | 65 |
| Major | Discomfort to the occupants  | C | 57 |
| Minor | Slight reduction in safety margins | D | 28 |
| No Safety Effect | No effect on safety | E | |

# Design Assurance Levels - Examples

**A**



Flight Control System

Engine Control System

Display System

**B**

Warning System

**C**

Radio System

**D**

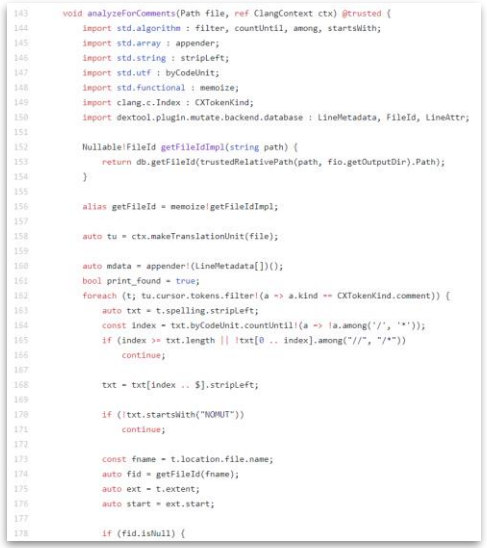
Flight Planning System

**E**

Entertainment system

Coffee delivery system

# How can we effectively test software?



```
void analyzeForComments(Path file, ref ClangContext ctx) @trusted {
    import std.algorithm : filter, countUntil, among, startsWith;
    import std.array : appender;
    import std.string : stripLeft;
    import std.utf : byCodeUnit;
    import std.functional : memoize;
    import clang.c.Index : CXTokenKind;
    import dextool.plugin.mutate.backend.database : LineMetadata, FileId, LineAttr;

    Nullable!FileId getFileIdImpl(string path) {
        return db.getFileId(trustedRelativePath(path, fio.getOutputDir).Path);
    }

    alias getFileId = memoize!getFileIdImpl;

    auto tu = ctx.makeTranslationUnit(file);

    auto mdata = appender!(LineMetadata[])();
    bool print_found = true;
    foreach (t; tu.cursor.tokens.filter!(a => a.kind == CXTokenKind.comment)) {
        auto txt = t.spelling.stripLeft;
        const index = txt.byCodeUnit.countUntil!(a => !a.among('/', '*'));
        if (index >= txt.length || !txt[0 .. index].among("//", "/*"))
            continue;

        txt = txt[index .. $].stripLeft;

        if (!txt.startsWith("NOMUT"))
            continue;

        const fname = t.location.file.name;
        auto fid = getFileId(fname);
        auto ext = t.extent;
        auto start = ext.start;

        if (fid.isNull) {
```

Program

?

```
******* TESTs *******
[==========] Running 11 tests from 3 test cases.
[----------] Global test environment set-up.
[----------] 5 tests from WahrFalschTest
[ RUN      ] WahrFalschTest.BeideWahrBeideWahr
[       OK ] WahrFalschTest.BeideWahrBeideWahr (0 ms)
[ RUN      ] WahrFalschTest.BeideWahrEinerFalsch
[       OK ] WahrFalschTest.BeideWahrEinerFalsch (0 ms)
[ RUN      ] WahrFalschTest.OderBeideWahr
[       OK ] WahrFalschTest.OderBeideWahr (0 ms)
[ RUN      ] WahrFalschTest.OderEinerFalsch
[       OK ] WahrFalschTest.OderEinerFalsch (0 ms)
[ RUN      ] WahrFalschTest.OderBeideFalsch
[       OK ] WahrFalschTest.OderBeideFalsch (0 ms)
[----------] 5 tests from WahrFalschTest (17 ms total)

[----------] 2 tests from FakultaetTest
[ RUN      ] FakultaetTest.Iterativ
[       OK ] FakultaetTest.Iterativ (7 ms)
[ RUN      ] FakultaetTest.Rekursiv
[       OK ] FakultaetTest.Rekursiv (0 ms)
[----------] 2 tests from FakultaetTest (25 ms total)

[----------] 4 tests from TestSequence/FakultaetParamsTest
[ RUN      ] TestSequence/FakultaetParamsTest.TestMitParametern/0
[       OK ] TestSequence/FakultaetParamsTest.TestMitParametern/0 (0 ms)
[ RUN      ] TestSequence/FakultaetParamsTest.TestMitParametern/1
[       OK ] TestSequence/FakultaetParamsTest.TestMitParametern/1 (0 ms)
[ RUN      ] TestSequence/FakultaetParamsTest.TestMitParametern/2
[       OK ] TestSequence/FakultaetParamsTest.TestMitParametern/2 (0 ms)
[ RUN      ] TestSequence/FakultaetParamsTest.TestMitParametern/3
[       OK ] TestSequence/FakultaetParamsTest.TestMitParametern/3 (0 ms)
[----------] 4 tests from TestSequence/FakultaetParamsTest (44 ms total)

[----------] Global test environment tear-down
[==========] 11 tests from 3 test cases ran. (126 ms total)
[  PASSED  ] 11 tests.
```

Tests

- What to test?
- How to test?
- When to stop?

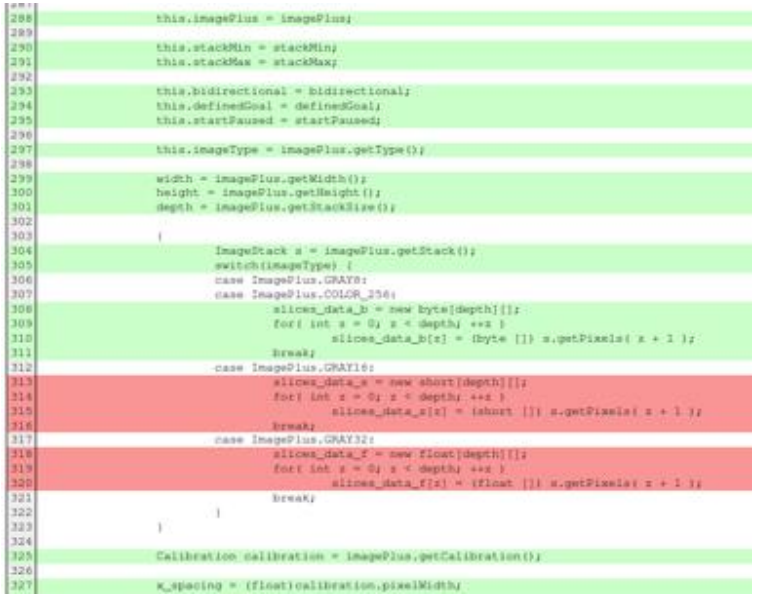
- How effective are the tests?
- What is missing?

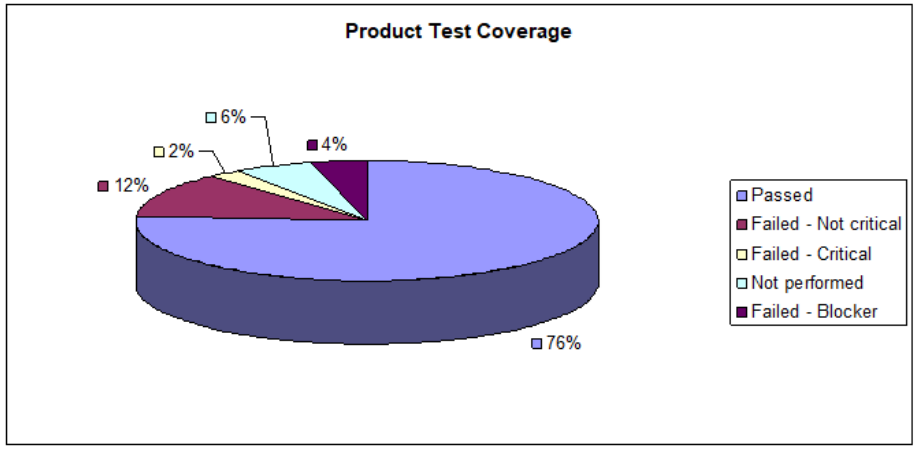
*Source: G. Petrovic, M Ivanokovic, et. al.*

# Coverage metrics

Many types may be interesting:

- Coverage of requirements
- Coverage of functions
- Coverage of use cases
- Coverage of code structure
  - Function coverage
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - MCDC






# RTCA/DO-178C: Table A-7 - Verification of Verification Process Results

| Objective | | Applicability by Software Level | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1 | Test procedures are correct | ● | ○ | ○ | |
| 2 | Test results are correct and discrepancies explained. | ● | ○ | ○ | |
| 3 | Test coverage of high-level requirements is achieved. | ● | ○ | ○ | ○ |
| 4 | Test coverage of low-level requirements is achieved | ● | ○ | ○ | |
| 5 | Test coverage of software structure (modified condition/decision coverage) is achieved | ● | | | |
| 6 | Test coverage of software structure (decision coverage) is achieved. | ● | ○ | | |
| 7 | Test coverage of software structure (statement coverage) is achieved. | ● | ○ | ○ | |
| 8 | Test coverage of software structure (data coupling and control coupling) is achieved | ● | ○ | ○ | |

● The objective should be satisfied with independence.

○ The objective should be satisfied.

**Blank** Satisfaction of the objective is at application's discretion.

# RTCA/DO-178C: Table A-7 - Verification of Verification Process Results

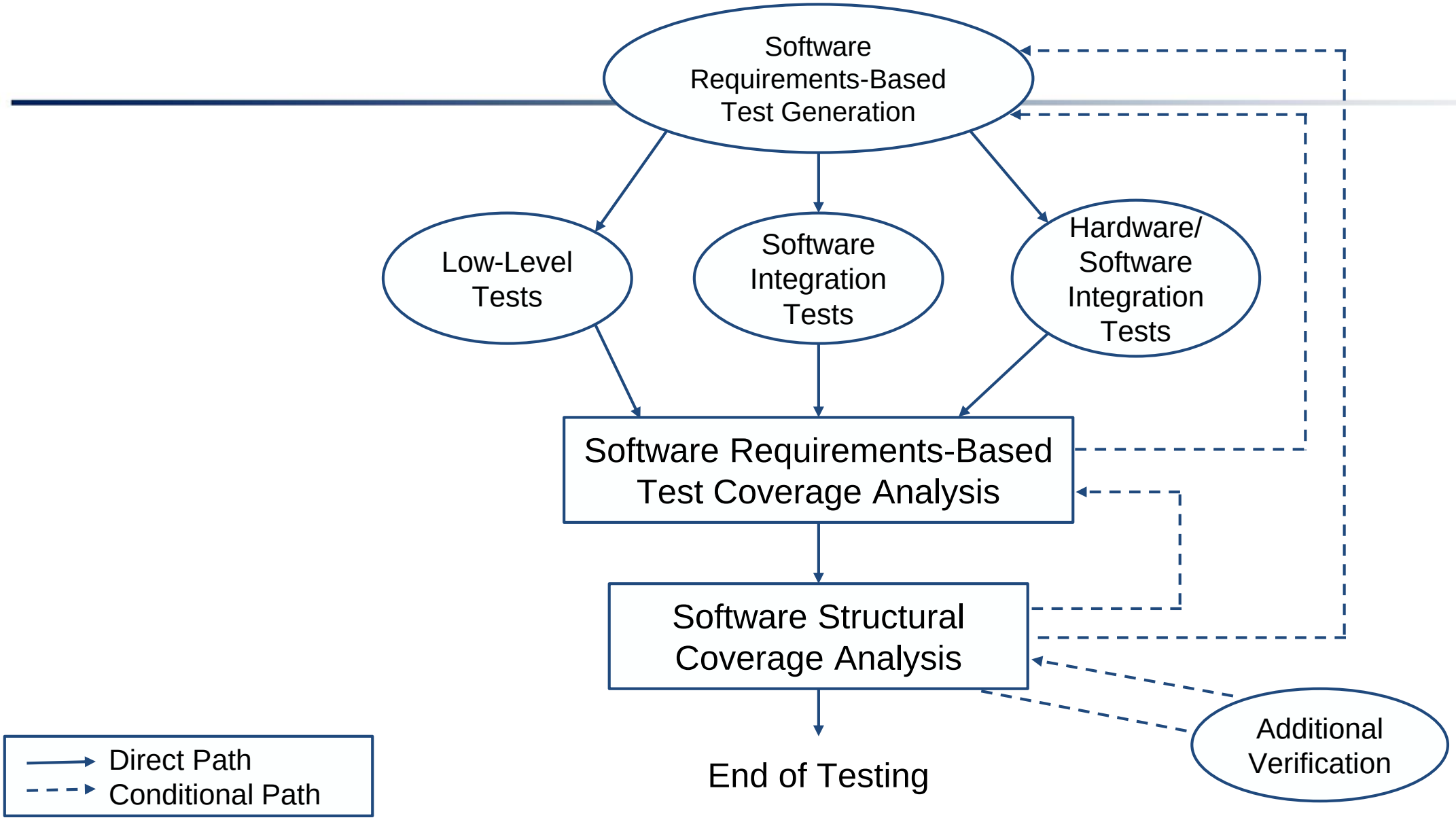
| Objective | | Applicability by Software Level | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1 | Test procedures are correct | ● | ○ | ○ | |
| 2 | Test results are correct and discrepancies explained. | ● | ○ | ○ | |
| 3 | Test coverage of high-level requirements is achieved. | ● | ○ | ○ | ○ |
| 4 | Test coverage of low-level requirements is achieved | ● | ○ | ○ | |
| 5 | Test coverage of software structure (modified condition/decision coverage) is achieved | ● | | | |
| 6 | Test coverage of software structure (decision coverage) is achieved. | ● | ○ | | |
| 7 | Test coverage of software structure (statement coverage) is achieved. | ● | ○ | ○ | |
| 8 | Test coverage of software structure (data coupling and control coupling) is achieved | ● | ○ | ○ | |

● The objective should be satisfied with independence.

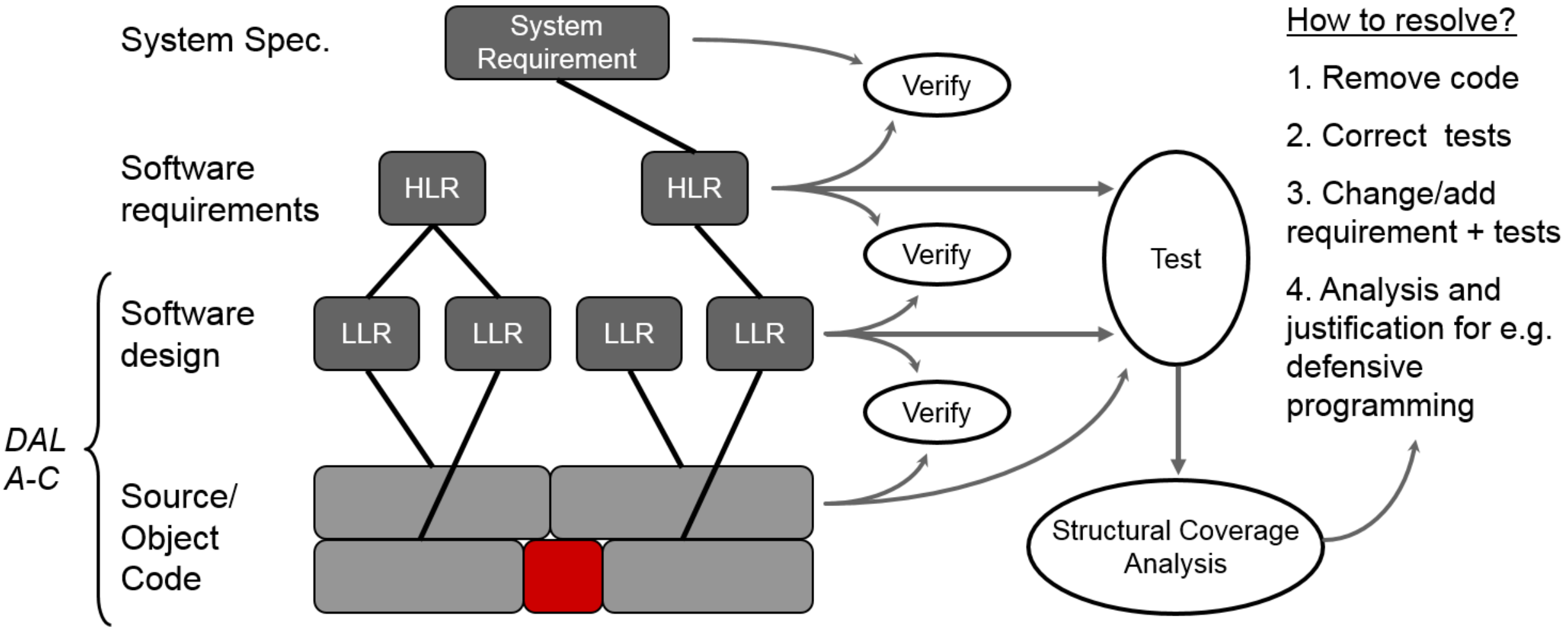
○ The objective should be satisfied.

**Blank** Satisfaction of the objective is at application's discretion.

# RTCA/DO-178C: Figure 6-1 - Software Testing Process



# Working with code coverage



# Practical example

# Practical example

- **Small program**

```
#include "fibonacci_example.hpp"

int fibonacci(int x){
    if(x < 0){
        return -1;
    }
    if (x == 0){
        return 0;
    }
    if (x == 1){
        return 1;
    }

    return fibonacci(x-1)+fibonacci(x-2);
}
```

# Practical example

- Small program
- **Test suite**

```
1  #define BOOST_TEST_DYN_LINK
2  #define BOOST_TEST_MAIN
3  #include "fibonacci_example.hpp"
4  #include <boost/test/unit_test.hpp>
5  #include <boost/test/output_test_stream.hpp>
6
7  BOOST_AUTO_TEST_CASE(point_accessor_test) {
8      BOOST_CHECK_EQUAL(fibonacci(0), 0);
9      BOOST_CHECK_EQUAL(fibonacci(1), 1);
10     BOOST_CHECK_EQUAL(fibonacci(2), 1);
11     BOOST_CHECK_EQUAL(fibonacci(5), 5);
12 }
```

# Practical example

- Small program
- Test suite
- **Test result**

```
Test project /home/niklas/Projects/liu-lab/mutantninjas/test-project/build
    Start 1: decrement_test
1/7 Test #1: decrement_test ...................   Passed    0.01 sec
    Start 2: enum_test
2/7 Test #2: enum_test ........................   Passed    0.00 sec
    Start 3: fibonacci_test
3/7 Test #3: fibonacci_test ...................   Passed    0.00 sec
    Start 4: lottery_test
4/7 Test #4: lottery_test .....................   Passed    0.00 sec
    Start 5: struct_test
5/7 Test #5: struct_test ......................   Passed    0.00 sec
    Start 6: triangle_test
6/7 Test #6: triangle_test ....................   Passed    0.00 sec
    Start 7: void_test
7/7 Test #7: void_test ........................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 7
```

# Practical example

- Small program
- Test suite
- Test result
- **Are we testing everything?**

```
1  #include "fibonacci_example.hpp"
2
3  int fibonacci(int x){
4      if(x < 0){
5          //return -1;
6      }
7      if (x == 0){
8          return 0;
9      }
10     if (x == 1){
11         return 1;
12     }
13
14     return fibonacci(x-1)+fibonacci(x-2);
15 }
```

# Practical example

- Small program
- Test suite
- Test result
- **Are we testing everything?**

```
1  #include "fibonacci_example.hpp"
2
3  int fibonacci(int x){
4      if(false){
5          return -1;
6      }
7      if (x == 0){
8          return 0;
9      }
10     if (x == 1){
11         return 1;
12     }
13
14     return fibonacci(x-1)+fibonacci(x-2);
15 }
```

# Practical example

- Small program
- Test suite
- Test result
- **Are we testing everything?**
  - **Obviously not …**

```
Test project /home/niklas/Projects/liu-lab/mutantninjas/test-project/build
    Start 1: decrement_test
1/7 Test #1: decrement_test ...................   Passed    0.01 sec
    Start 2: enum_test
2/7 Test #2: enum_test ........................   Passed    0.00 sec
    Start 3: fibonacci_test
3/7 Test #3: fibonacci_test ...................   Passed    0.00 sec
    Start 4: lottery_test
4/7 Test #4: lottery_test .....................   Passed    0.00 sec
    Start 5: struct_test
5/7 Test #5: struct_test ......................   Passed    0.00 sec
    Start 6: triangle_test
6/7 Test #6: triangle_test ....................   Passed    0.00 sec
    Start 7: void_test
7/7 Test #7: void_test ........................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 7
```

# Practical example

- Small program
- Test suite
- Test result
- Are we testing everything?
  - Obviously not …
- **Add test case**

```
1  #define BOOST_TEST_DYN_LINK
2  #define BOOST_TEST_MAIN
3  #include "fibonacci_example.hpp"
4  #include <boost/test/unit_test.hpp>
5  #include <boost/test/output_test_stream.hpp>
6
7  BOOST_AUTO_TEST_CASE(point_accessor_test) {
8      BOOST_CHECK_EQUAL(fibonacci(-10), -1);
9      BOOST_CHECK_EQUAL(fibonacci(0), 0);
10     BOOST_CHECK_EQUAL(fibonacci(1), 1);
11     BOOST_CHECK_EQUAL(fibonacci(2), 1);
12     BOOST_CHECK_EQUAL(fibonacci(5), 5);
13 }
```

# Practical example

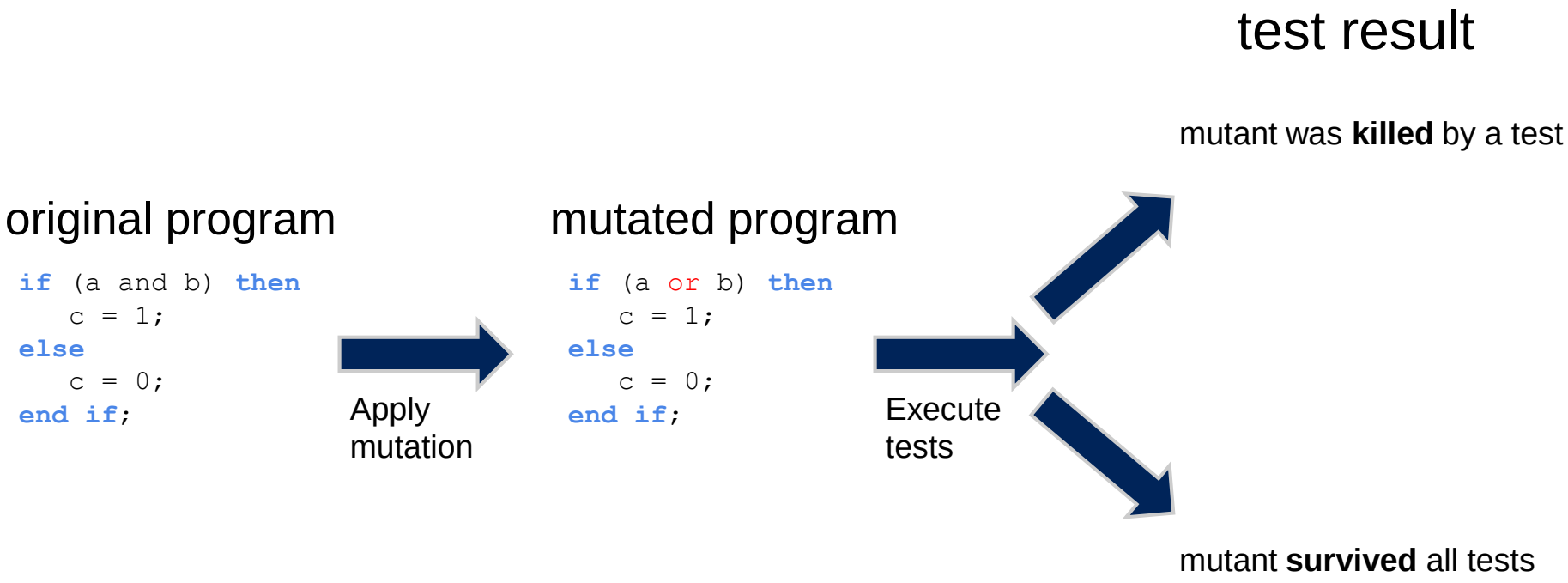
- Small program
- Test suite
- Test result
- Are we testing everything?
  - Obviously not …
- Add test case
- **We detect the injected fault**

```
Test project /home/niklas/Projects/liu-lab/mutantninjas/test-project/build
    Start 1: decrement_test
1/7 Test #1: decrement_test ...................   Passed    0.01 sec
    Start 2: enum_test
2/7 Test #2: enum_test ........................   Passed    0.00 sec
    Start 3: fibonacci_test
3/7 Test #3: fibonacci_test ...................***Failed    0.01 sec
    Start 4: lottery_test
4/7 Test #4: lottery_test .....................   Passed    0.00 sec
    Start 5: struct_test
5/7 Test #5: struct_test ......................   Passed    0.00 sec
    Start 6: triangle_test
6/7 Test #6: triangle_test ....................   Passed    0.00 sec
    Start 7: void_test
7/7 Test #7: void_test ........................   Passed    0.00 sec

86% tests passed, 1 tests failed out of 7

Total Test time (real) =   0.04 sec

The following tests FAILED:
          3 - fibonacci_test (Failed)
```

# Practical example

- Small program
- Test suite
- Test result
- Are we testing everything?
  - Obviously not …
- Add test case
- We detect the injected fault
- **What did we learn?**
  - **Detection of flaws**
  - **Tedious**

# Mutation Testing



# Mutation Testing

Focuses on determining the adequacy of a test suite.

- It is *fault based testing* directed towards typical syntactical faults that occurs when constructing a program.

- It relies on two hypothesis:
  - *The competent programmer*
    - Given a specification, a programmer develops a program that is either correct or differs from the correct program by a combination of simple errors.
  - *The coupling effect*
    - Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors

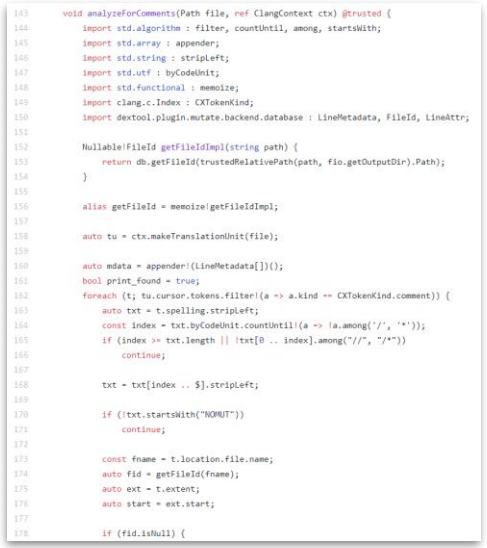
# Mutation Score

The mutation score $M_s$ of a test set T, designed to test P, is computed as the number of killed mutants divided by the total amount of mutants, with the equivalent mutants subtracted.

$$M_s(P,T) = \frac{M_k}{M_t - M_q}$$

- $M_t$ = total number of mutants
- $M_k$ = mutants killed
- $M_q$ = equivalent mutants

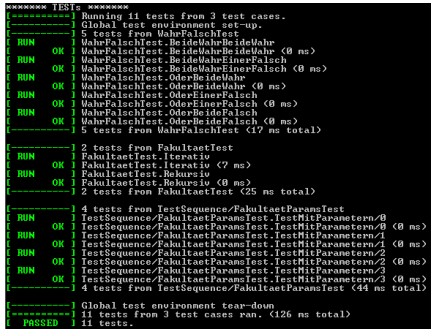
# How can we effectively test software?



```
void analyzeForComments(Path file, ref ClangContext ctx) @trusted {
    import std.algorithm : filter, countUntil, among, startsWith;
    import std.array : appender;
    import std.string : stripLeft;
    import std.utf : byCodeUnit;
    import std.functional : memoize;
    import clang.c.Index : CXTokenKind;
    import dextool.plugin.mutate.backend.database : LineMetadata, FileId, LineAttr;

    Nullable!FileId getFileIdImpl(string path) {
        return db.getFileId(trustedRelativePath(path, fio.getOutputDir).Path);
    }

    alias getFileId = memoize!getFileIdImpl;

    auto tu = ctx.makeTranslationUnit(file);

    auto mdata = appender!(LineMetadata[])();
    bool print_found = true;
    foreach (t; tu.cursor.tokens.filter!(a => a.kind == CXTokenKind.comment)) {
        auto txt = t.spelling.stripLeft;
        const index = txt.byCodeUnit.countUntil!(a => !a.among('/', '*'));
        if (index >= txt.length || !txt[0 .. index].among("//", "/*"))
            continue;

        txt = txt[index .. $].stripLeft;

        if (!txt.startsWith("NOMUT"))
            continue;

        const fname = t.location.file.name;
        auto fid = getFileId(fname);
        auto ext = t.extent;
        auto start = ext.start;

        if (fid.isNull) {
```

Program

Mutation Testing



```
******* TESTs *******
[==========] Running 11 tests from 3 test cases.
[----------] Global test environment set-up.
[----------] 5 tests from WahrFalschTest
[ RUN      ] WahrFalschTest.BeideWahrBeideWahr
[       OK ] WahrFalschTest.BeideWahrBeideWahr (0 ms)
[ RUN      ] WahrFalschTest.BeideWahrEinerFalsch
[       OK ] WahrFalschTest.BeideWahrEinerFalsch (0 ms)
[ RUN      ] WahrFalschTest.OderBeideWahr
[       OK ] WahrFalschTest.OderBeideWahr (0 ms)
[ RUN      ] WahrFalschTest.OderEinerFalsch
[       OK ] WahrFalschTest.OderEinerFalsch (0 ms)
[ RUN      ] WahrFalschTest.OderBeideFalsch
[       OK ] WahrFalschTest.OderBeideFalsch (0 ms)
[----------] 5 tests from WahrFalschTest (17 ms total)

[----------] 2 tests from FakultaetTest
[ RUN      ] FakultaetTest.Iterativ
[       OK ] FakultaetTest.Iterativ (7 ms)
[ RUN      ] FakultaetTest.Rekursiv
[       OK ] FakultaetTest.Rekursiv (0 ms)
[----------] 2 tests from FakultaetTest (25 ms total)

[----------] 4 tests from TestSequence/FakultaetParamsTest
[ RUN      ] TestSequence/FakultaetParamsTest.TestMitParametern/0
[       OK ] TestSequence/FakultaetParamsTest.TestMitParametern/0 (0 ms)
[ RUN      ] TestSequence/FakultaetParamsTest.TestMitParametern/1
[       OK ] TestSequence/FakultaetParamsTest.TestMitParametern/1 (0 ms)
[ RUN      ] TestSequence/FakultaetParamsTest.TestMitParametern/2
[       OK ] TestSequence/FakultaetParamsTest.TestMitParametern/2 (0 ms)
[ RUN      ] TestSequence/FakultaetParamsTest.TestMitParametern/3
[       OK ] TestSequence/FakultaetParamsTest.TestMitParametern/3 (0 ms)
[----------] 4 tests from TestSequence/FakultaetParamsTest (44 ms total)

[----------] Global test environment tear-down
[==========] 11 tests from 3 test cases ran. (126 ms total)
[  PASSED  ] 11 tests.
```

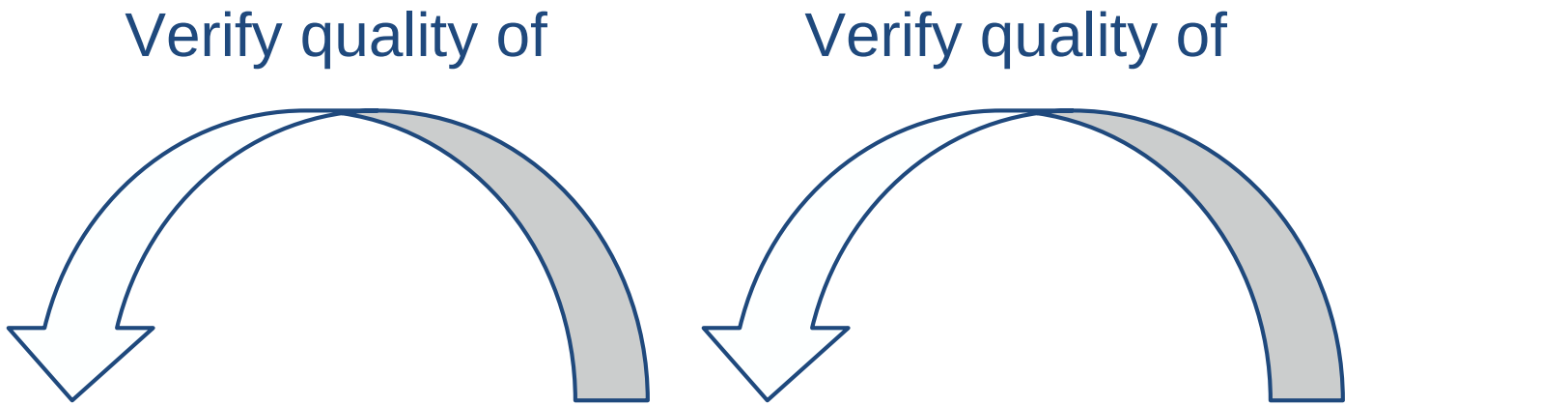
Tests

- What to test?
- How to test?
- When to stop?

- How effective are the tests?
- What is missing?

*Source: G. Petrovic, M Ivanokovic, et. al.*

# Mutation Testing



# Mutation Operators

- Which mutations do we apply?
  - Are there mutations we do not want to perform?
- How are mutations applied?
  - Can for example type information help us?
- When are mutations applicable?
- The answer is **Mutation operators!**
  - Template schemes for implementing mutations.

**Examples**

Statement Deletion (SDL):
// return x;

Boolean Subexpression Replacement (BSR):
True, False

Arithmetic Operator Replacement (AOR):
+, *, -, /

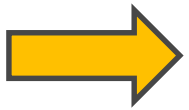
Logical Operator Replacement (LOR):
and, or, xor

Relational Operator Replacement (ROR):
<,>,==,!=,<=,>=

# Equivalent mutants

Consider the following example

```
int find (Array a, int value) {
     int res = -1;
     bool found = false;

     for (int i = 0; i < a.length(); ++i) {
          if (a[i] == value && !found) {
                    res = i;
                    found = true;
                    break;
          }
     }
     // Do something more
}
```



```
int find (Array a, int value) {
     int res = -1;
     bool found = false;

     for (int i = 0; i < a.length(); ++i) {
          if (a[i] == value && !found) {
                    res = i;
                    found = true;
                    // break;
          }
     }
     // Do something more
}
```

# Equivalent mutants

Assume we have:

- Function f
- Mutated version f’
- Input x

$$\forall x \; f(x) == f'(x)$$

- Impossible to kill
- Requires manual intervention
- Limits the usability of mutation testing

# Classification of mutations

There are problems with classifying mutations.

- The different aspects from “Academia vs. Industry”
- Are mutations that are *killable* (not equivalent) desirable?
- Are *killable* mutations always productive for developers?

# Productive mutations

Petrovic et al, “**An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions”,** 2018:

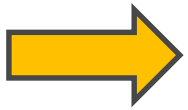
*”A mutant is productive if:*

*1) the mutant is killable and elicits an effective test*
*2) the mutant is equivalent but its analysis advances knowledge and code quality*

- **Note:** the notion of productive vs. unproductive mutants is inherently qualitative → Different developers may reach different conclusion

# Unproductive Mutants - Example

```
int find (Array a, int value) {
     int res = -1;
     bool found = false;

     for (int i = 0; i < a.length(); ++i) {
          if (a[i] == value && !found) {
                    res = i;
                    found = true;
                    break;
          }
     }
     // Do something more
     Log(”Found at index ”, i);
}
```



```
int find (Array a, int value) {
     int res = -1;
     bool found = false;

     for (int i = 0; i < a.length(); ++i) {
          if (a[i] == value && !found) {
                    res = i;
                    found = true;
                    break;
          }
     }
     // Do something more
     // Log(”Found at index ”, i);
}
```

**Note:** the notion of productive vs. unproductive mutants is inherently qualitative → Different developers may reach different conclusion

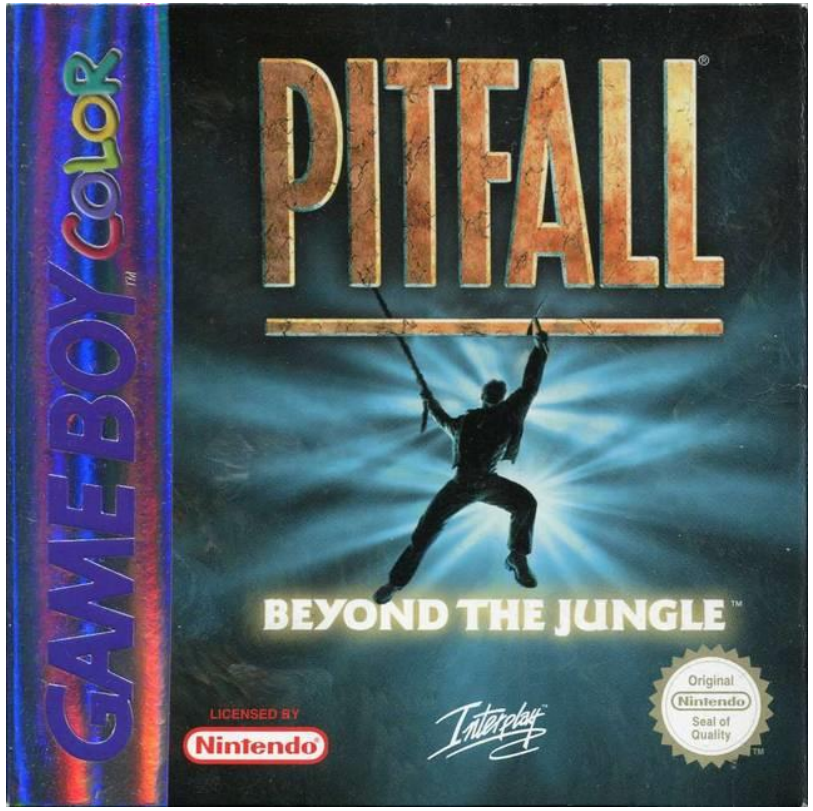
# Challenges and Pitfalls of Mutation testing

**Niklas Pettersson** niklas.pettersson2@saabgroup.com
**Joakim Brännström** joakim.k.brannstrom@saabgroup.com

# Challenges With Mutation Testing

Introducing a state-of-the-art test technique

- Project focused on delivering functionality
  - Cost and time
  - Talk the manager language
- Process capture how to develop projects
  - Experience needed to know how to change the process



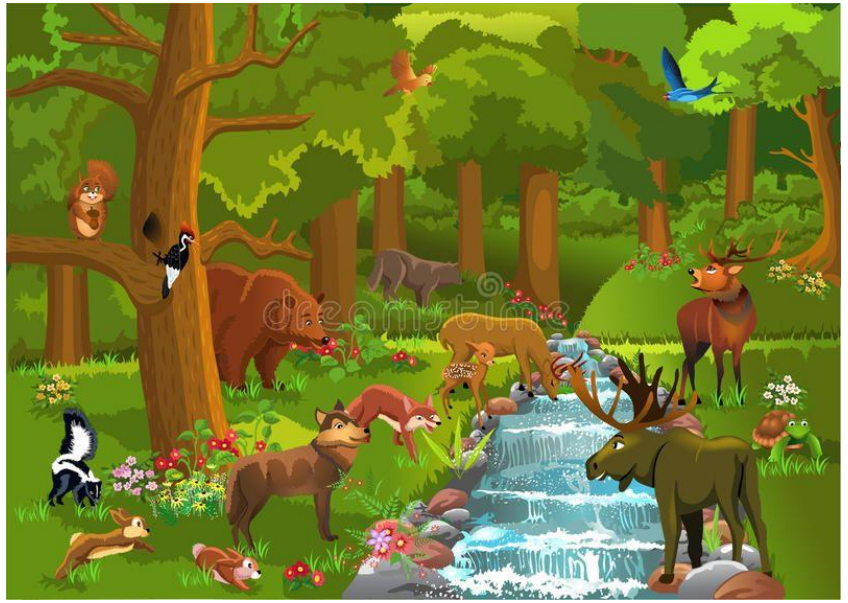
# Problem – Industrial Application

Application of mutation testing requires mature tooling for the project context and facts to enact a policy change.

- Tooling: need to be easy to use, good enough performance for the project size, easy to digest the result.
- Process: how to use mutation testing. Diff based via PR? Blocking activity? Legacy code? What can be ignored? What is important? Integration tests vs unit tests. What operators to use?
- Business value: what is quality? Why is test suite quality important? Coverage vs mutation testing?

# Grassroots vs Top Down

- We have chosen to use a grassroots approach
- Help teams get started with mutation testing.
- Be open that we do not know the best way to apply mutation testing. We need the teams creative help in figuring it out
- Engineers are curious by nature. If the information and tool is *a click away* and easy to understand they will sooner or later look at it
- Continuous dialog with the teams
- Collect experience of how it is used, share between teams. Write down **our** best practice.



# Industrial Value of Mutation Testing

**Statement**: Each line of code cost money to develop and maintain over time.

**Statement**: A small (few lines of code) and effective (good at finding faults) test suite is preferred. It is cheaper to maintain, understand and change.

**Assumption**: Use of mutation testing to improve a test suite will reduce the development and maintenance cost.

- Each test case is effective with reduced overlap
- Only relevant parts are unit tested

**Assumption**: In a process with requirements it **may** lead to an improvement of the requirements (reduction, addition, tweaking)

**Assumption**: When the test cases are manually inspected less flaws are found

# Experience - Focus

**Experience**: The focus should be on the individual test cases.

User feedback: 

- “I want this technique to be able to tell me what is unique with each test procedures using objective facts”
  - Report the mutants that are only killed by the test procedure in question
- "I have these test procedures that I wrote during one phase of the development. I haven't really kept them up to date. Since then I have written many more test procedures. I want to know if these old test cases are worth keeping or if they can be thrown away.“
  - Report the similarity between the test cases based on the mutants that are killed
- “I want to know if all test cases actually verify anything of the implementation”
  - Report test procedures that kill zero mutants

Related to inspection of test cases

# Pitfall - Interpretation

**Pitfall**: Mutation testing is based on the source code, not the requirements. The mutation testing result reflect the implementation.

- Each mutant is a *data point*
- What *data points* are gathered is based on the used mutation operators
- Only what is implemented is mutated
  - Missing requirements are NOT found by mutation testing
- Consequences:
  - Too few *unit of observation*
  - Wrong mutation operators
  - Bug in tool implementation of the mutation operator

Affects the interpretation of the result and its correlation to test suite quality

# Pitfall – Silver Bullet?

**Pitfall**: Too much focus on killing all mutants which miss the point of the quality of the test suite.

- All mutants are not equally important
  - Engineering judgement
- A project have a limited budget for test activities
  - Use the time wisely

# User Experience

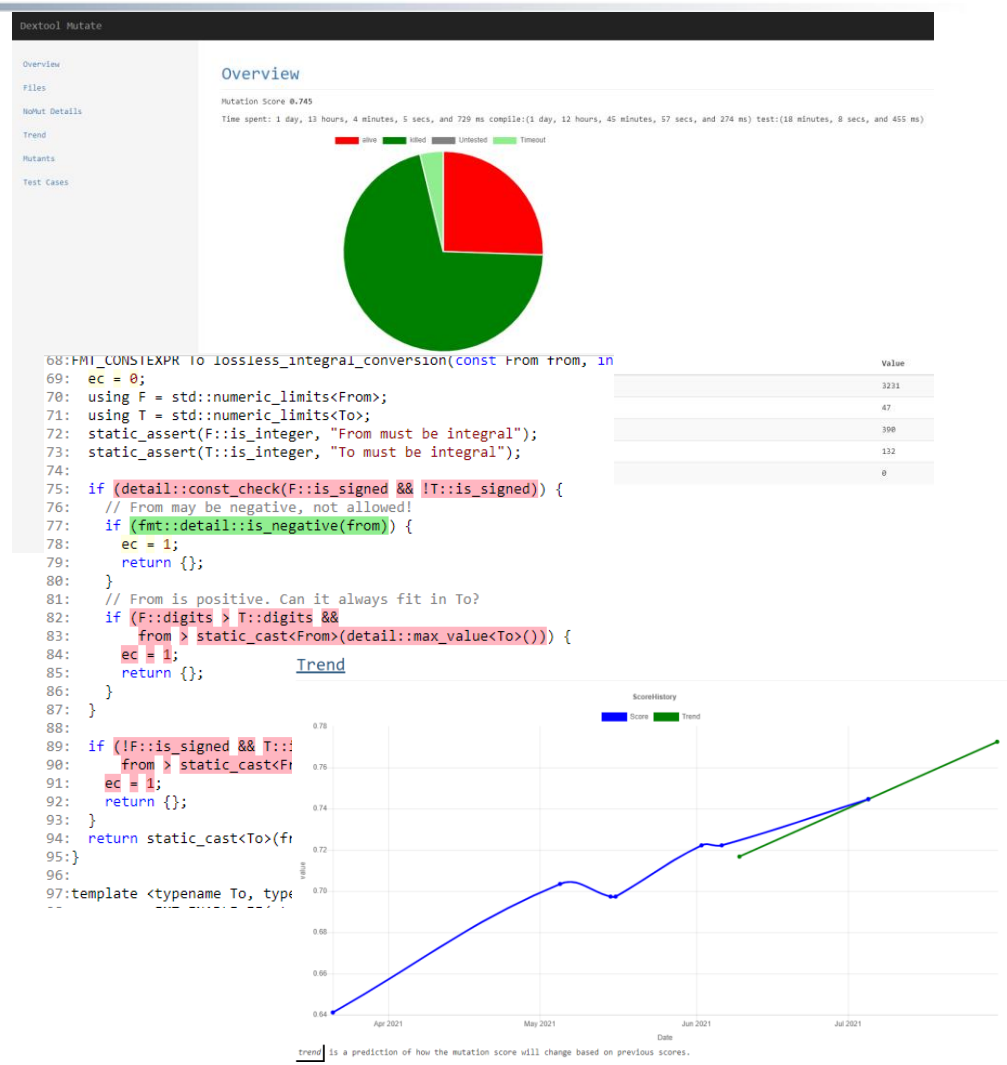
**Assumption**: the user is time constrained. Every minute must be *valuable*. Value is determined by subjective, soft facts and hard business value.

Example:

- Negative soft fact: the user **feel** that it is hard to understand the mutation testing report.
- Positive hard fact: improvements to the test suite based on mutation testing result in fewer bugs found in later verification activities. Effort in **other** verification activities are reduced.

# Report

- Actionable Information
  https://arxiv.org/abs/2103.07189
  Does mutation testing improve testing practices?
- Visualization and navigation is key factors for user acceptance
- Summarize information to be acted upon to reduce information overload
  - Detailed information is desired to better understand when the summarized information is is *interesting*.
- Trend make improving the test suite **fun**.
  The work is clearly visible to everyone.
  *We are best this month!*
  *See how much we have improved since we started.*



# Report

- Summary is a powerful tool for guiding developers without being explicit in a *process*
- Prioritize test case improvement
  - Fix broken test cases
  - Remove redundant test cases
  - Reduce overlap between test cases
  - Good test cases have at least one unique aspect
- Prioritize killing *significant* mutants
  - Large source code change

## Test Cases

| Unique 2 | Redundant 105 | Buggy 52 | Normal 263 |
|---|---|---|---|

kills mutants that no other test case do.

| Name | Killed |
|---|---|
| printf_test.printf_custom | 37 |
| ostream_test.print | 66 |

## High Interest Mutants

This list the 5 mutants that affect the most source code and has survived.

| Link | Tested | Priority |
|---|---|---|
| include/fmt/core.h:2767 | 2021-Jul-03 18:32:24.874Z | 1720 |
| include/fmt/format.h:522 | 2021-Jul-03 17:29:34.705Z | 1038 |
| include/fmt/format.h:522 | 2021-Jul-03 17:29:34.704Z | 1038 |
| include/fmt/format.h:522 | 2021-Jul-03 17:29:34.705Z | 1038 |
| include/fmt/format.h:522 | 2021-Jul-03 17:29:34.704Z | 1038 |

equivalent 132 skipped 390

| Tested |
|---|
| 2021-07-05 |
| include/fmt/args.h:136 16 0 2021-07-05 |
| include/fmt/args.h:201 40 0 2021-07-05 |
| include/fmt/args.h:212 0 0 2021-06-07 |
| include/fmt/args.h:222 0 0 2021-06-07 |
| include/fmt/args.h:225 0 0 2021-06-07 |
| include/fmt/args.h:226 0 0 2021-06-07 |
| include/fmt/chrono.h:47 22 0 2021-07-04 |
| include/fmt/chrono.h:51 38 0 2021-07-04 |

# Report Quality

**Problem**: Each mutation operator cost time and money because an operator add mutants that need to be tested (slower) and maybe assessed (manually inspected).

The choice of mutation operators is crucial.

Academic research recommendation:

- *An Experimental Determination of Sufficient Mutant Operators*
  - A. Jeffersson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untach, Christian Zapf
  - *The 5 sufficient operators are … ROR, AOR, LCR, UOI and ABS*
- Our experience from C/C++ and the tool implementation of the operators: LCR, DCR, SDL, AOR, ROR

# Mutation Operators

**Experience**: Not all mutation operators are equal

- Few unproductive mutants (LCR)
  - No protest from an user when a LCR survived
- Testing a new aspect compared to code coverage
  - Mathematical (AOR)
  - Data flow (SDL)
  - Bit operations (LCRb)
- Naive implementation of mutation operators
  - UOI, SDL. Too many undesired mutants
  - UOI, ABS, ROR. Too many equivalent mutants

**Productive Mutant**
A mutant is productive if 1) the mutant is killable and elicits an effective test, or 2) the mutant is equivalent but its analysis advances knowledge and code quality
Googles Paper 2018

**NOTE**: undefined behavior is a **higly** costly mutant that is unproductive. They are worse than equivalent mutants.

# Mutation Operators

**Experience**:

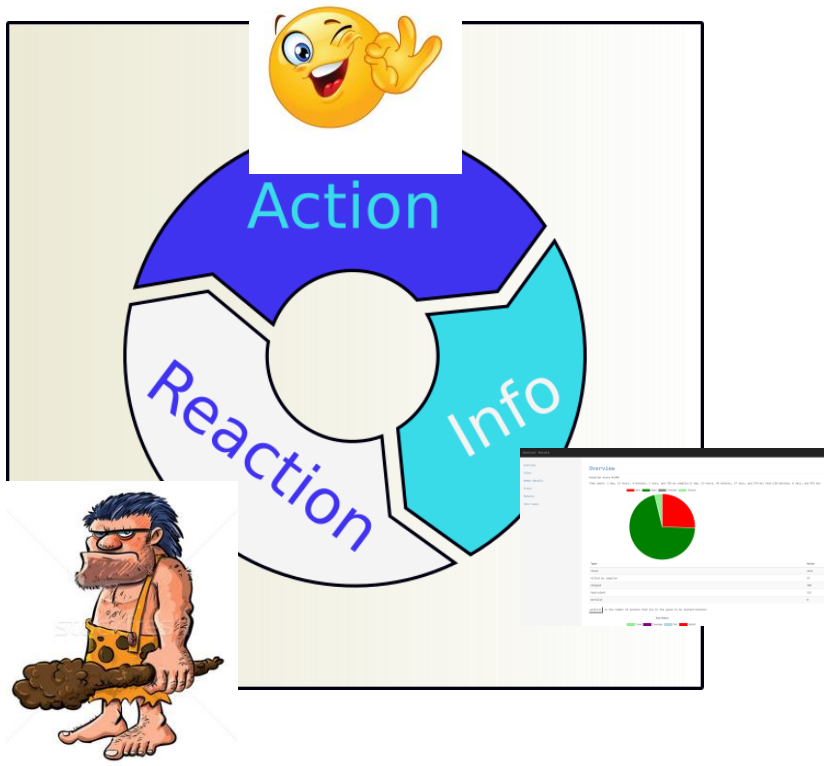
More research needed here!

- Priority: LCR, DCR, SDL, AOR, RORp
- Naive to sane implementation. Improvements:
  - ROR. RORG (see paper) and type information.
    50% reduction in equivalent mutants.
- Adjust existing operators
  - Focus SDL on data flow
    See Offuts paper Designing Deletion Mutation Operators
  - Add DCR and
  - Bitwise LCR
  - Limit UOI to removing negation
- The problem of equivalent mutants is highly affected by the mutation operators used
  - Use mutation operators with few equivalent mutants
  - Choose those most needed for the type of software Saab develope

**Productive Mutant**
A mutant is productive if 1) the mutant is killable and elicits an effective test, or 2) the mutant is equivalent but its analysis advances knowledge and code quality
Googles Paper 2018

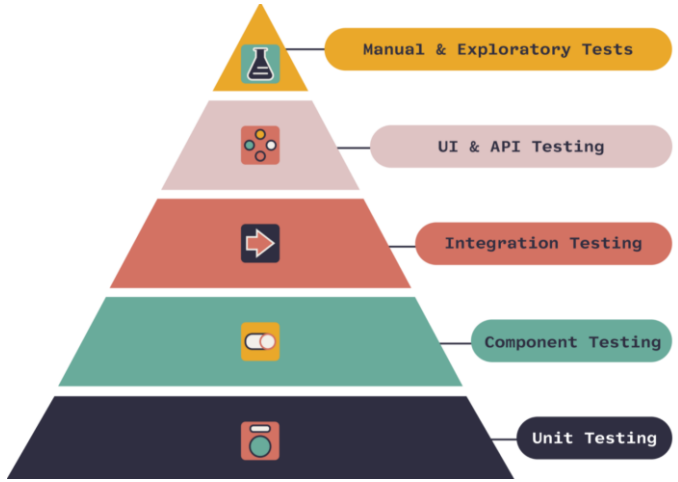
# Feedback Speed

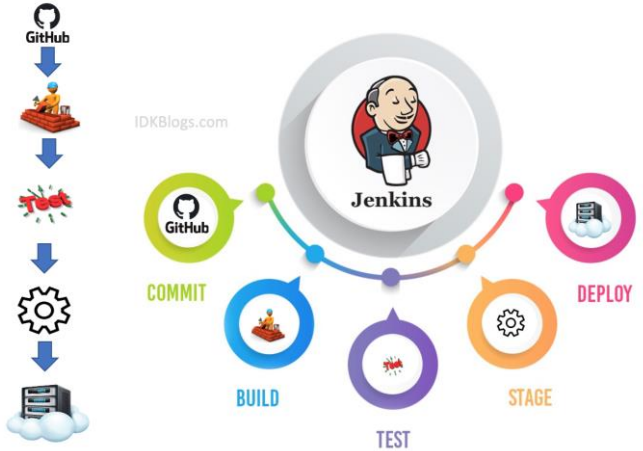
- Google use diff based with smart selection
  - https://arxiv.org/abs/2102.11378
    Practical Mutation Testing at Scale
- Saab use incremental, continues, full testing
- Which one is better?
  It depends on the context
- What is important is that the time between change to feedback is acceptable for the user and how it is used
  - Pull request, minutes?
  - Overview, hours?



# Saab Context

- Applications are 10-50k lines
- Test strategy per application to determine what is tested where in the test pyramid
  **Heavy** focus on component testing for verification. Integration for validation.
- CI automatically run mutation testing on merge to main
  - Only changes are tested because previous results are re-used
  - User have a fresh report within minutes to hours
- Users check the report in the afternoon
- Teams check the report each sprint for focused test suite improvements.
  - Monitors that the quality improve
  - Where are the ”holes”



# Saab Context - Summary

- Tooling is mature enough for daily use
- The size of the application mean that the speed is acceptable together with the tool improvements
- Business value is still an assumption
  - Challenge being worked on
- Teams and users like the methodology
  - Easy to understand
  - Very actionable information
  - Fun to write test cases
- Equivalent mutants is not a practical problem
- Unproductive mutants are manageable
- Process under development

# Research Papers – Recommended Reading

- https://arxiv.org/abs/2103.07189
  Does mutation testing improve testing practices?
- https://arxiv.org/abs/2102.11378
  Practical Mutation Testing at Scale
- https://arxiv.org/abs/2010.13464
  What It Would Take to Use Mutation Testing in Industry--A Study at Facebook
- https://research.google/pubs/pub46584
  https://testing.googleblog.com/2021/04/mutation-testing.html
  State of Mutation Testing at Google
- https://cs.gmu.edu/~offutt/rsrch/mut.html
  An Experimental Determination of Sufficient Mutant Operators, 1996
  Designing Deletion Mutation Operators, 2014

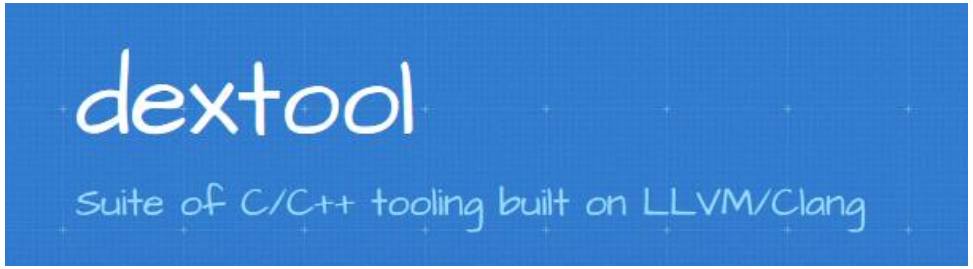
# Short tool introduction

**Niklas Pettersson** niklas.pettersson2@saabgroup.com
**Joakim Brännström** joakim.k.brannstrom@saabgroup.com

# Dextool Mutate

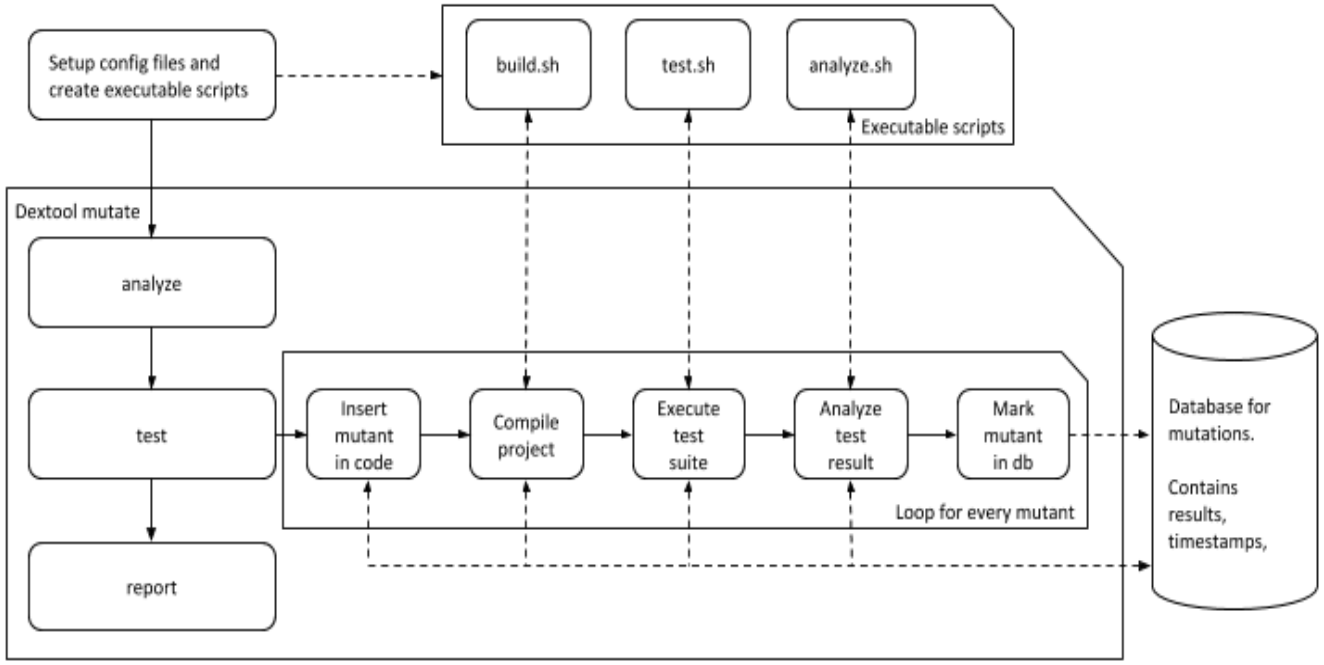
Dextool's plugin for mutation testing of C/C++ projects.

- Can operate on big projects and applications
- Help you design new tests
- Evaluate the quality of existing tests
- Open source



# Dextool Mutate

- Divided into 3 main parts
  - *Analyze*
  - *Test*
  - *Report*
- Utilizes user-provided scripts for
  - Compilation
  - Execution of tests
  - Analyzation of test result
- Works by traversing the AST
  - Utilizes type information for “smarter” mutations
- Developed from user-feedback
- Is used at SAAB



# Dextool Mutate

Demo is available here:

https://www.youtube.com/watch?v=RUJvqiyUdSY