

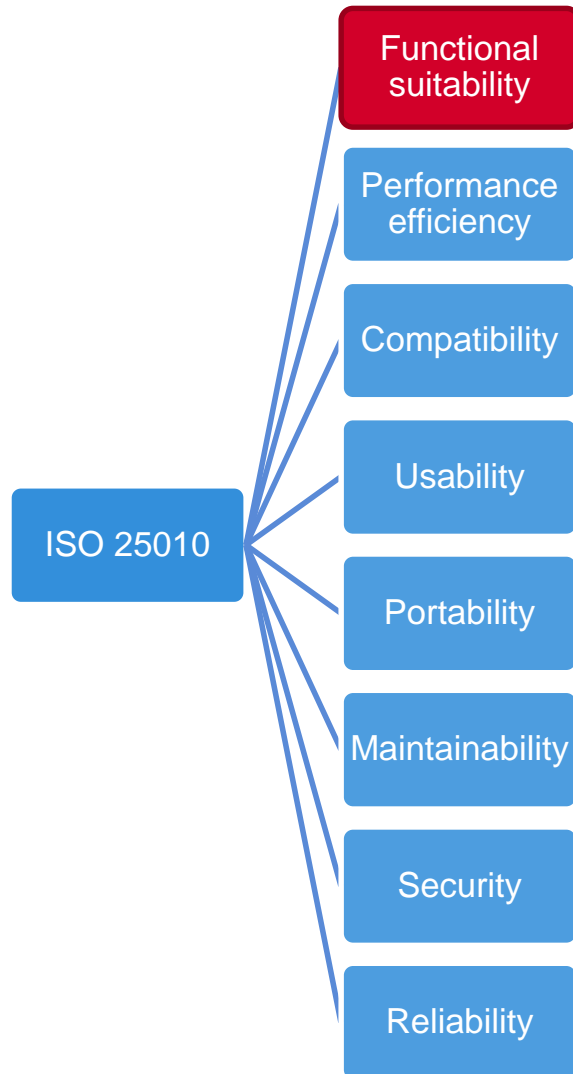
Clean Unit Tests

Johannes Hochrainer

Senior Consultant & Trainer

johannes.hochrainer@software-quality-lab.com

What does “clean” mean?

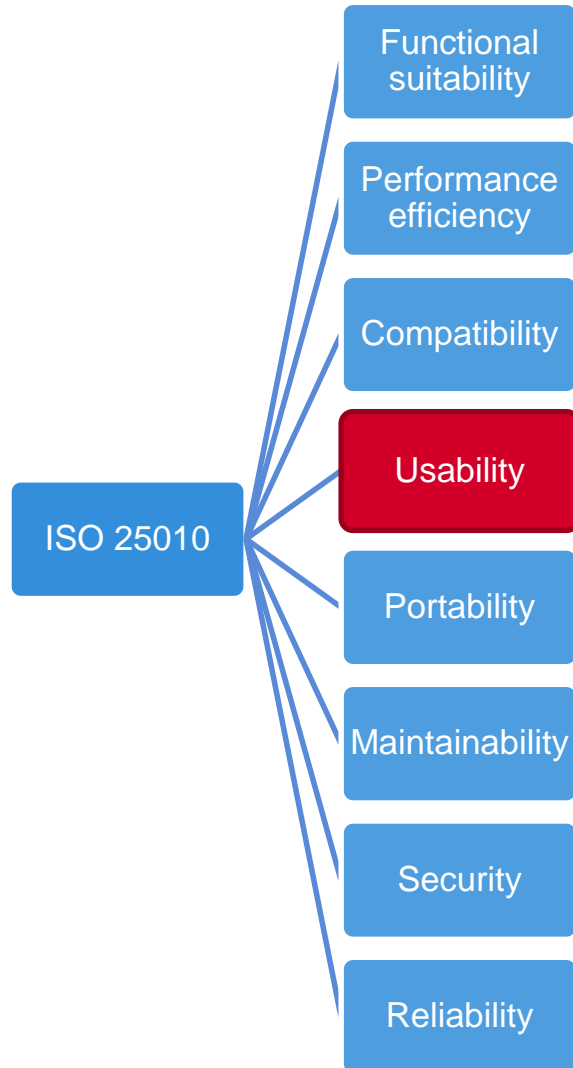


As a product owner I want that **all requirements are tested** so that the customer gets what he wants.

As a test manager I want that **unit tests are derived systematically** from requirements so that nobody doubts the quality of the tests.

As a developer I want to have a **trustworthy feedback** after a change so that I can **work faster**.



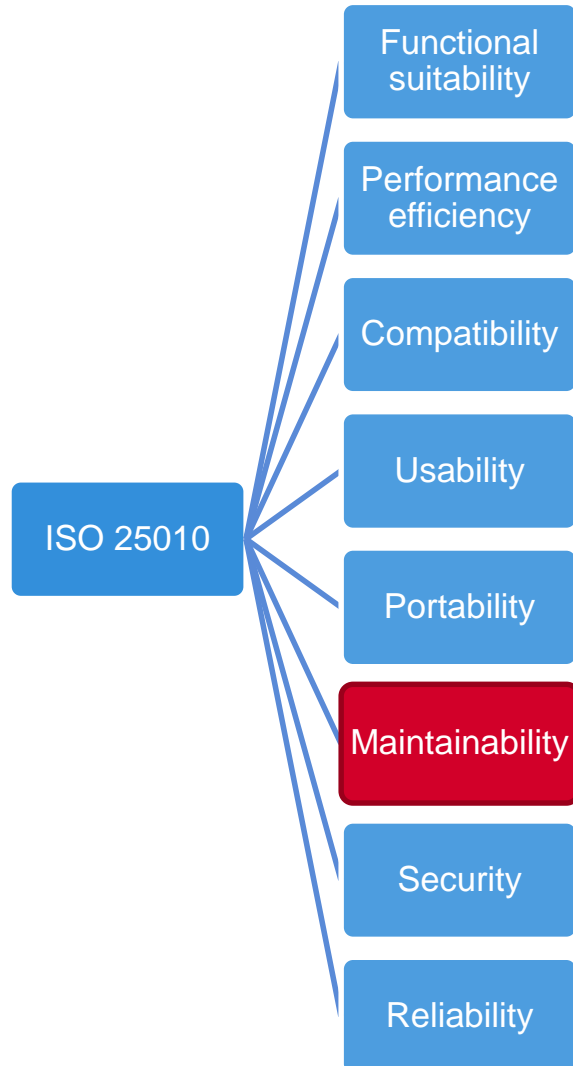


As a freshman I want to read unit tests in order to **learn the software**.

As a developer I want to understand the unit tests of my colleagues quickly so that I can **easily extend them**.

As a tester I want to **review the unit tests** in order to verify the coverage.





As a developer I want that a **code change affects only a few unit tests** so that I don't have to spend much time with updating unit tests.

As a developer I want to add a new unit tests without having to care about **side effects**.

As a developer I want to **separate unit tests form integration** tests so that I can execute them separately and have a quick feedback.



Functional Suitability

→ Trustworthiness

Clean Unit Tests

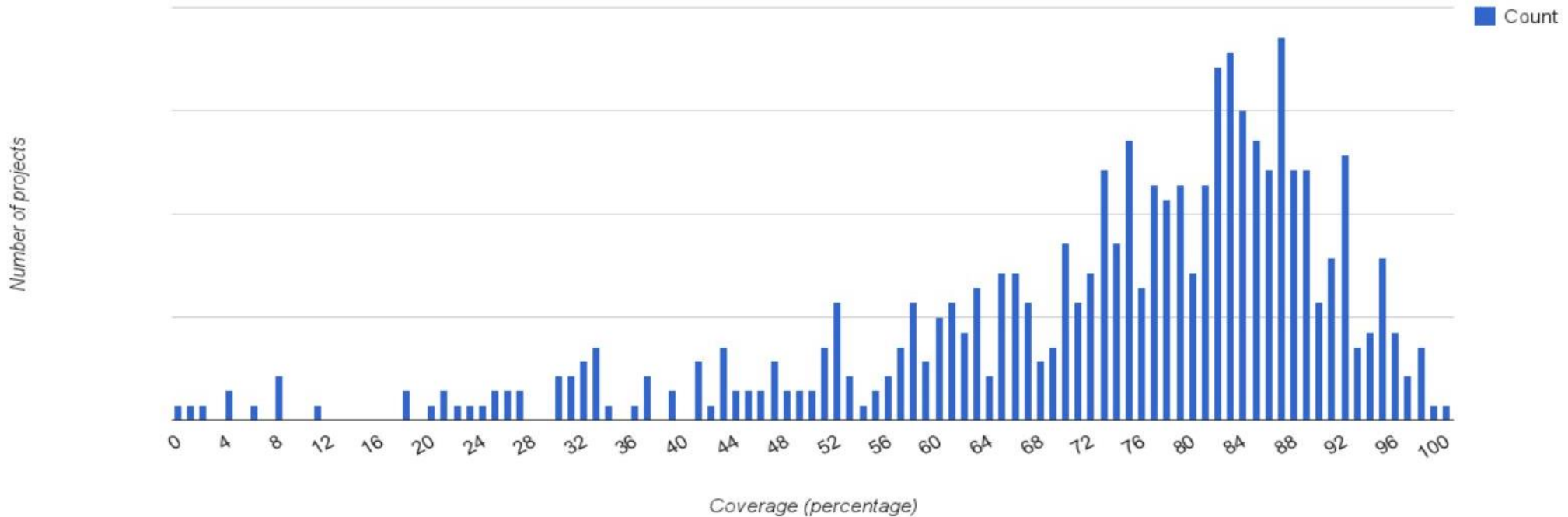
Be careful with coverage criteria

- Test smell
 - No reviews
 - Low code coverage
- Impact
 - A low code coverage ($< 50\%$) may miss a lot of bugs.
 - Developers may select path of least resistance.
- Solution
 - Go for high code coverage in components with much logic.
 - Monitor test coverage with tools.



Per-project coverage

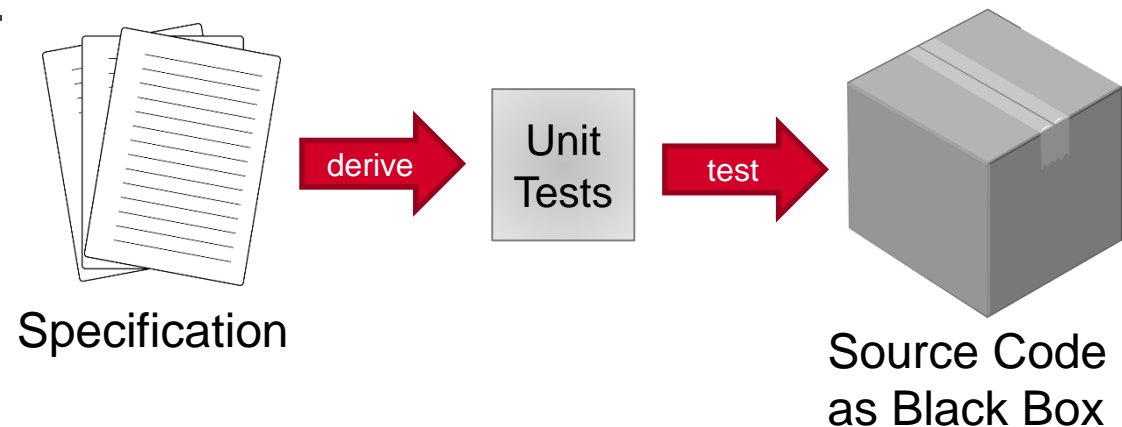
Histogram of average coverage over one month



The median is 78%, the 75th percentile 85% and 90th percentile 90%.

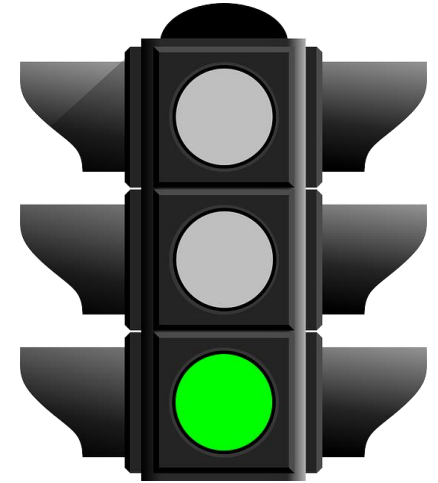
Start with black box test design techniques

- **Test smell**
 - Derive test cases from source code.
 - Focus only on coverage criteria.
- **Impact**
 - Developers focus on coverage criteria and not on quality of tests.
 - You find only a few bugs.
- **Solution**
 - Use black box test design techniques.
 - Verify test quality with reviews.



Fix failed tests immediately

- Test smell
 - Some tests are always red.
- Impact
 - Developer scrutinise test results.
 - Tests are unstable.
 - More and more tests will become red.
- Solution
 - Fix failed tests immediately.
 - Notify developers: “Do it more often if it hurts.”
 - Publish test results.



Usability

Clean Unit Tests

Find meaningful test class names

■ Test smell

- Generic test class name
- e.g. `BasicTest`, `MainTest`

■ Impact

- Intention of tests is not clearly visible.
- Not association with class under test.

■ Solution

- Pattern: *<ClassUnderTest>, “Test”, [“_”, Function \ Precondition]*
- Example: `Vector.cp`
 - `VectorTest.cpp` *//all tests in 1 test class*
 - `VectorTest_at.cpp` *//all tests for function at(..)*
 - `VectorTest_twoElements.cp` *//all tests with same precondition*

Find meaningful unit test names

■ Test smell

- Generic test method name
- e.g. test function `at(...)` → `atBasic`, `at_basic`, `atFirst`, ...

■ Impact

- Difficult to understand the test
- Difficult to verify requirements coverages

■ Solution



Given

When

Then

```
TEST (VectorTest, at_posSmallerSize_elemOfPos) { . . . }  
TEST (VectorTest, at_pos0_elemOfPos0) { . . . }  
TEST (VectorTest, at_posSizeMin1_elemOfPosSizeMin1) { . . . }  
TEST (VectorTest, at_posGreaterSize_throwsException) { . . . }
```

Test only one aspect

■ Test smell

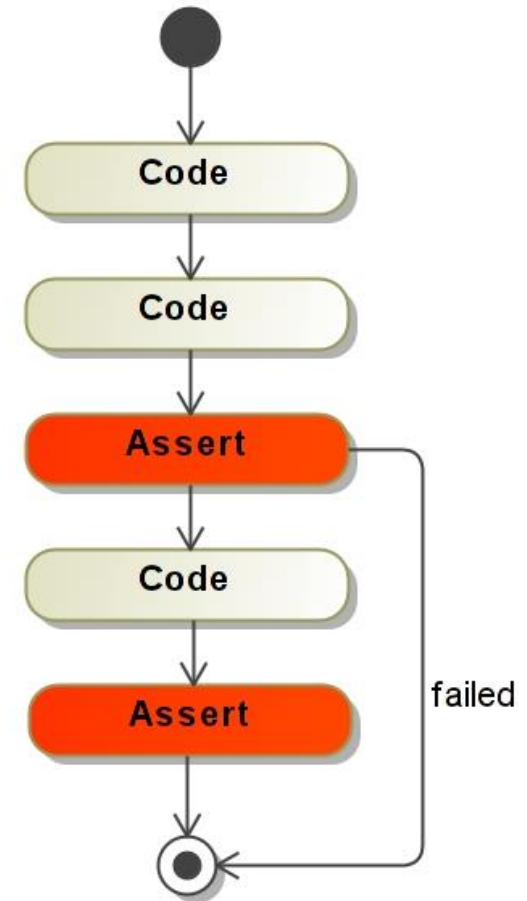
- Assertions on different positions
- More than one assert in test

■ Impact

- In the case of failure not all asserts are evaluated
 - Getting only part of the symptoms
 - Loss of information that could be helpful for debugging
- Difficult to name test meaningfully
- Test is difficult to understand

■ Solution

- Move each assert into a separate test
- Parameterize tests



Test only one aspect

```
TEST(VectorTest, multipleAsserts) {  
    vector<int> vector;  
    ASSERT_TRUE(vector.isEmpty());  
    vector.push_back(1234);  
    int size = vector.size();  
    ASSERT_EQ(1, size);  
}
```

Move each assert into a separate test



```
TEST(VectorTest, isEmpty_justInit_empty) {  
    vector<int> vector;  
    ASSERT_TRUE(vector.isEmpty());  
}
```

```
TEST(VectorTest, size_pushback_sizeIs1AndEmptyIsFalse) {  
    vector<int> vector;  
    vector.push_back(1234);  
    ASSERT_EQ(1, vector.size());  
    ASSERT_FALSE(vector.isEmpty());  
}
```

Test only one aspect

```
TEST(LocationTest, convert_coordinateWithFormatDegrees_stringWithFormatDegrees) {
    Location location;
    double input = 0.0;
    string result = "0.0000";
    ASSERT_EQ(result, location.convert(input, Location::FORMAT_DEGREES));
    input = 10.0;
    result = "10.0000";
    ASSERT_EQ(result, location.convert(input, Location::FORMAT_DEGREES));
    input = -10.0;
    result = "-10.0000";
    ASSERT_EQ(result, location.convert(input, Location::FORMAT_DEGREES));
    input = 180.0;
    result = "180.0000";
    ASSERT_EQ(result, location.convert(input, Location::FORMAT_DEGREES));
    input = -180.0;
    result = "-180.0000";
    ASSERT_EQ(result, location.convert(input, Location::FORMAT_DEGREES));
}
```

```
class LocationTest : public ::testing::TestWithParam<pair<double, const char*> > {
public:
    virtual void SetUp() {}
};

INSTANTIATE_TEST_CASE_P(PositiveValues, LocationTest, ::testing::Values(
    make_pair(0.0, "0.0000"),
    make_pair(10.0, "10.0000"),
    make_pair(180.0, "180.0000")
));

INSTANTIATE_TEST_CASE_P(NegativeValues, LocationTest, ::testing::Values(
    make_pair(-10.0, "-10.0000"),
    make_pair(-180.0, "-180.0000"),
    make_pair(-18.0, "-18.0000")
));

TEST_P(LocationTest, convert_coordinateWithFormatDegrees_stringWithFormatDegrees) {
    Location location;
    double input = GetParam().first;
    string expect = GetParam().second;
    string actual = location.convert(input, Location::FORMAT_DEGREES);
    ASSERT_EQ(expect, actual);
}
```


Avoid logic in tests

```
void LocationTest::getters_setters() {
    double testValues[3] = { 300.0, 34.12, 43.33 };
    Location location;

    location.setAltitude(300.0);
    if (location.hasAltitude()) {
        CPPUNIT_ASSERT_EQUAL(300.0, location.getAltitude());
        location.reset();
        for (int i = 0; i < 3; i++) {
            switch (i) {
                case 0:
                    location.setSpeed(testValues[i]);
                    if (location.hasSpeed())
                        CPPUNIT_ASSERT_EQUAL(testValues[i], location.getSpeed());
                    break;
                case 1:
                    location.setLatitude(testValues[i]);
                    if (location.hasLatitude())
                        CPPUNIT_ASSERT_EQUAL(testValues[i], location.getLatitude());
                    break;
                case 2:
                    location.setLongitude(testValues[i]);
                    if (location.hasLongitude())
                        CPPUNIT_ASSERT_EQUAL(testValues[i], location.getLongitude());
                    break;
                default:
                    CPPUNIT_FAIL("out of range");
            }
        }
    } else {
        CPPUNIT_FAIL("location has no altitude");
    }
}
```

Avoid logic in tests

■ Test smell

- if, while, try-catch, switch, ...

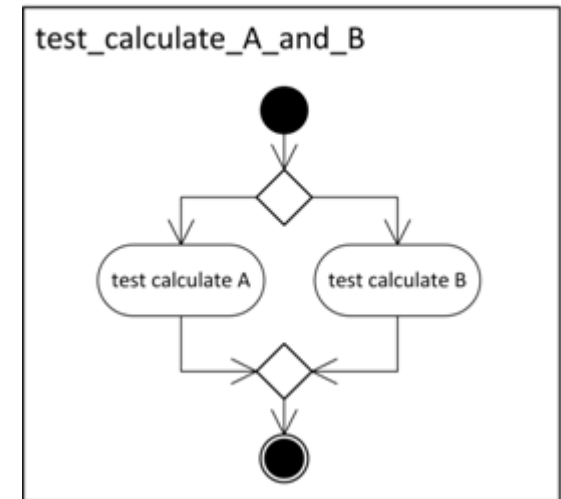
■ Impact

- Logic is error-prone
- Branches test more than one thing
- Test is harder to read and understand

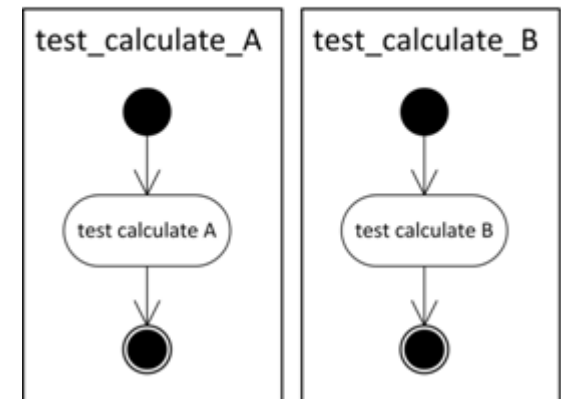
■ Solution

- Move each assertion into a single test without logic
- Test is sequence of 3 – 7 statements
- Bad design → complex test

1. Test with logic



2. Split test into two tests without logic



Don't move assertions to helper functions

```
TEST_F(IoTest, ZlibIo) {  
    const int kBufferSize = 2*1024;  
    uint8* buffer = new uint8[kBufferSize];  
    for (int i = 0; i < kBlockSizeCount; i++) {  
        for (int j = 0; j < kBlockSizeCount; j++) {  
            for (int z = 0; z < kBlockSizeCount; z++) {  
                int gzip_buffer_size = kBlockSizes[z];  
                int size;  
                {  
                    ArrayOutputStream output(buffer, kBufferSize, kBlockSizes[i]);  
                    GzipOutputStream::Options options;  
                    options.format = GzipOutputStream::ZLIB;  
                    if (gzip_buffer_size != -1) {  
                        options.buffer_size = gzip_buffer_size;  
                    }  
                    GzipOutputStream gzout(&output, options);  
                    WriteStuff(&gzout);  
                    gzout.Close();  
                    size = output.ByteCount();  
                }  
                {  
                    ArrayInputStream input(buffer, size, kBlockSizes[j]);  
                    GzipInputStream gzin(  
                        &input, GzipInputStream::ZLIB, gzip_buffer_size);  
                    ReadStuff(&gzin);  
                }  
            }  
        }  
    }  
    delete [] buffer;  
}
```

Maintainability

Clean Unit Tests

Enforce test isolation

■ Test smell

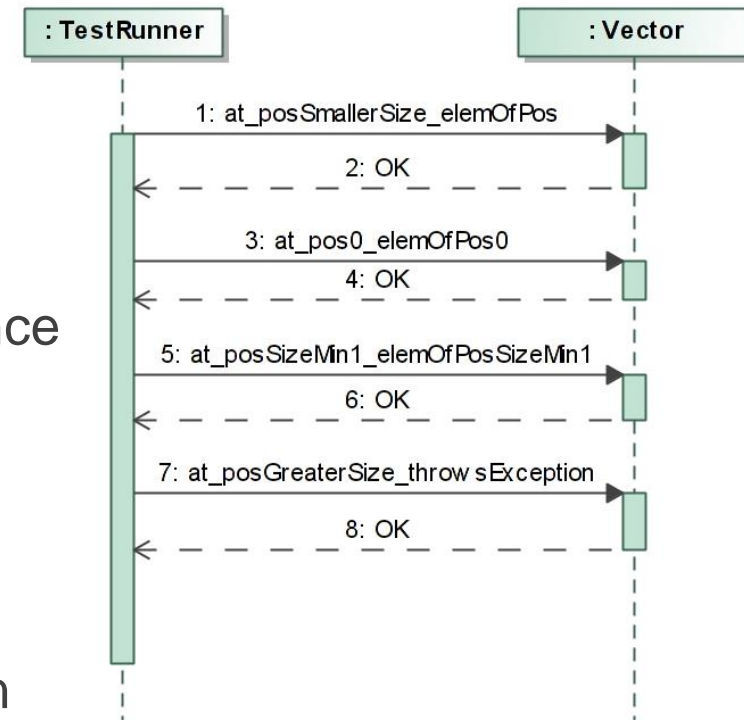
- Constrained test order
- Shared state

■ Impact

- Tests fail after introduction of new test
- Tests fail if they are executed in other sequence
- Resources in undefined state

■ Solution

- Use test fixtures (setup, teardown)
- All tests in a file must have same precondition
- Init global variables in Setup()



Avoid overspecification in tests

■ Test smell

- Too many assertions
 - preconditions
 - internal behavior
 - specific order
 - exact string match

FALSE
TRUE NO EQ THROW
ASSERT
STREQ

■ Impact

- Test is fragile and likely to fail
- Test is harder to read (too much information)
- Test checks too many aspects

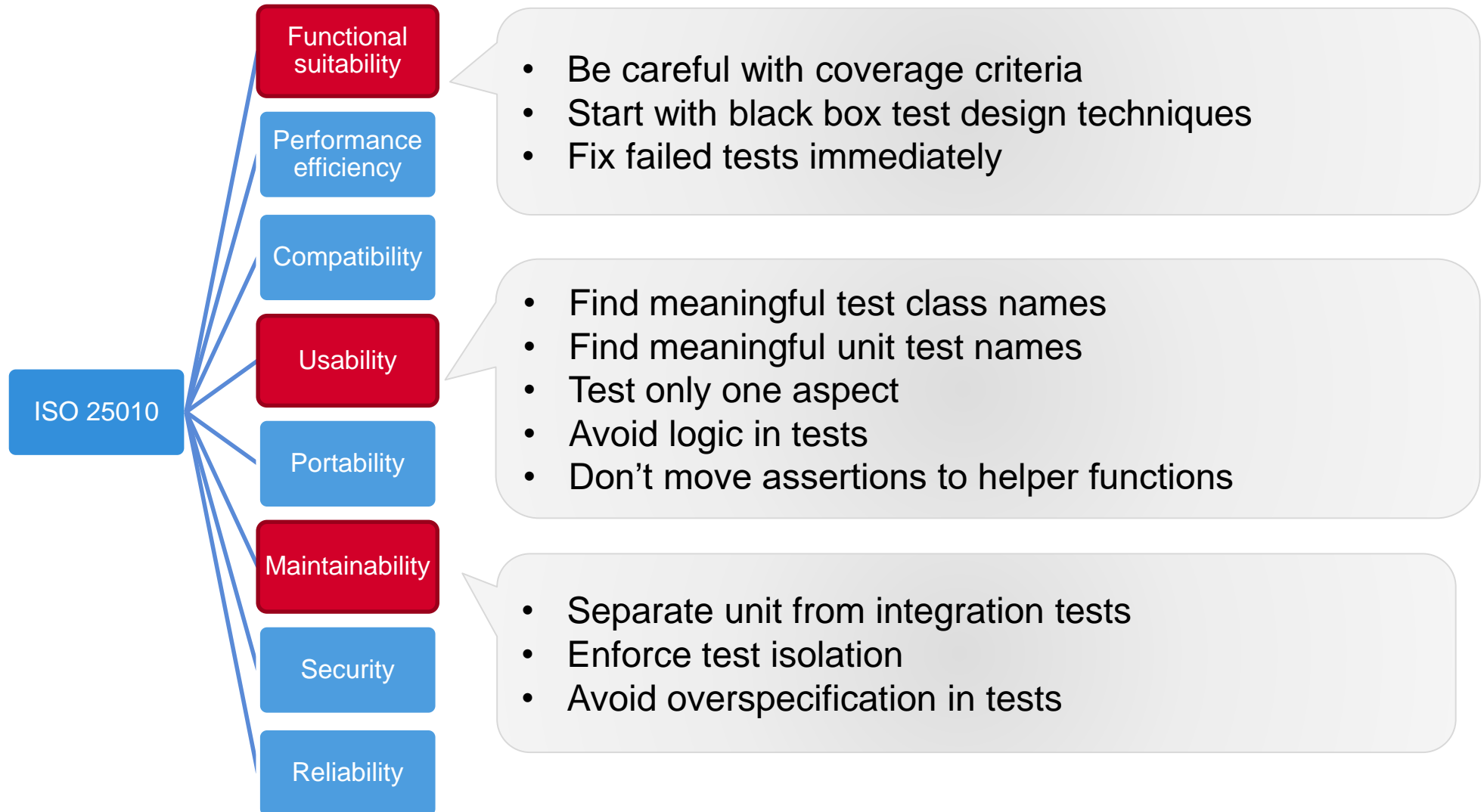
■ Solution

- Focus on the important aspects

Avoid overspecification in tests

```
TEST(LocationManagerTest, getProvider_notInitialized_throwsLogicErrorWithMessage) {
    LocationManager locationManager;
    try {
        locationManager.getProvider("gps");
        FAIL();
    } catch (logic_error& e) {
        ASSERT_STREQ("Providers not initialized", e.what());
    } catch (exception&) {
        FAIL();
    }
}
```

```
TEST(LocationManagerTest, getProvider_notInitialized_throwsLogicError) {
    LocationManager locationManager;
    ASSERT_THROW(locationManager.getProvider(LocationManager::GPS), logic_error);
}
```



Request more Clean Unit Test rules from
johannes.hochrainer@software-quality-lab.com

Software Quality Lab

INNOVATION MEETS QUALITY

Agnes-Pockels-Bogen 1
D-80992 München

[T] +49 89 4423066-0
[W] www.software-quality-lab.com