NI Tech Talks

Google C++ Testing Framework

// Stefan Gränitz, Reaktor Dev Team, Berlin 2014 05 20

Quick Facts

- *Google C++ Testing Framework*, *Google Test* or *gtest*
- unit-testing of C/C++ code with minimal source modification
- development started in 2005, initial release in 2008
- open source under 3-clause BSD license
- xUnit architecture
- cross-platform
- **all we need for automated testing?!**

Overview

1. Terminology
2. Pros and Cons for Google Test
3. Basic Concepts
4. Test Fixtures
5. Advanced Concepts
6. Notes on Automated Testing
7. Automated vs. Interactive Testing
8. Conclusion

But first of all let's have a look at some code!

Basic Example

roduction code:

```
int Factorial(int n); // Returns the factorial of n
```

your test code:

```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput)
{
  EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput)
{
  EXPECT_EQ(1, Factorial(1));
  EXPECT_EQ(2, Factorial(2));
  EXPECT_EQ(6, Factorial(3));
  EXPECT_EQ(40320, Factorial(8));
}
```

Terminology

test case
group of related tests

unit test
set of independent test cases covering all aspects of a code unit

regression test
tests applied frequently to verify existing functionality

smoke test
a few unit tests for key parts of the system that execute automatically (ideally for every compilation)

flaky test
test that is non-deterministic, because of environmental-, code- or test-code-issues

white-box testing

knowledge of code under test is used when designing a test

Test-Driven Development

small development cycles of write test, write minimal implementation, refactor

Pros and Cons for Google Test

Pros

lightweight, widely used, debug support

under active development

https://code.google.com/p/googletest/source/list

documentation is quite good

https://code.google.com/p/googletest/wiki/Primer

Cons

extensive usage of preprocessor macros

not available for Objectiv-C and iOS

..anything else?

Alternatives

Boost Test

http://www.boost.org/doc/libs/1_55_0/libs/test/doc/html

CppUnit

http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html

a whole jungle of small developments and domain-specific libraries

http://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle

Basic Concepts

Your main.cpp:

```
#include "this/package/test/foo_TEST.h"
#include "that/package/test/bar_TEST.h"
#include "gtest/gtest.h"

int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

- put it to a separate test project
- include all relevant header files for test code
- link to gtest.lib + all relevant production code libs
- compile to console executable

gle test skeleton:

```
TEST(test_case_name, test_name)
{
 ... test body ...
}
```

- organize test code in test cases and single tests
- tests should be independent and repeatable
- automatic test discovery: no need to register tests manually

Non-Fatal Assertions

```
EXPECT_TRUE(condition);
EXPECT_FALSE(condition);
EXPECT_EQ(expected, actual);
EXPECT_NE(expected, actual);
EXPECT_LT(val1, val2);
EXPECT_LE(val1, val2);
EXPECT_GT(val1, val2);
EXPECT_GE(val1, val2);
```

Fatal Assertions

```
ASSERT_TRUE(condition);
ASSERT_FALSE(condition);
ASSERT_EQ(expected, actual);
ASSERT_NE(expected, actual);
ASSERT_LT(val1, val2);
ASSERT_LE(val1, val2);
ASSERT_GT(val1, val2);
ASSERT_GE(val1, val2);
```

- use *expect* by default: test execution continues after failture
- use *assert* for hard pre-conditions on remaining test body: test execution **does not** continue after failture — this may also affect clean-up code!
- both offer well-structured error descriptions in system console and IDE out of the box

Test Fixtures

```
class TestFixtureClass : public ::testing::Test
{
protected:
  void SetUp() override {
    // called before each test is run
  }

  void TearDown() override {
    // called after each test is run
  }

  // declare members your tests want to use

private:
  // declare internal members
};

TEST_F(TestFixtureClass, TestName) {
  // you can access protected members
  // from TestFixtureClass here
}
```

- each `TEST_F` gets a fresh instance of `TestFixtureClass`

- order of invokation:
  → `TestFixtureClass()`
  → `SetUp()` → test body → `TearDown()`
  → `~TestFixtureClass()`

- prefer `SetUp()` / `TearDown()` over constructor / destructor for anything but resource allocation / deallocation

- if possible use the C++11 `override` keyword to avoid common mistakes like misspelling `SetUp()` as `Setup()`

Advanced Concepts

Improve error messages:

```
ASSERT_EQ(x.size(), y.size()) << "Vectors x and y are of unequal length";

for (int i = 0; i < x.size(); i++)
{
  EXPECT_EQ(x[i], y[i]) << "Vectors x and y differ at index " << i;
}
```

- provide **additional** information using the stream-in operator
- works in the same way as for `std::cout`

The problem with subroutines..

tion under Test:

```
int fib(int n)
{
  return (n < 2) ? n : (fib(n - 1) + fib(n - 2));
}
```

Helper function:

```
bool isEven(int n)
{
  return (n % 2) == 0;
}
```

Test code:

```
TEST(Fibonacci, eachThirdIsEven)
{
  EXPECT_TRUE(isEven(fib(0)));
  EXPECT_TRUE(isEven(fib(3)));
  EXPECT_TRUE(isEven(fib(4))); //< this fails
  EXPECT_TRUE(isEven(fib(6)));
}
```

..solved using SCOPED_TRACE

Helper macro:

```
#define M_CALL_SCOPE_TRACED(statement) { \
  SCOPED_TRACE("in " #statement); statement; }
```

Helper function:

```
void isEven(int n) {
  EXPECT_EQ(0, n % 2);
}
```

Modified test code:

```
TEST(Fibonacci, eachThirdIsEven) {
  M_CALL_SCOPE_TRACED(isEven(fib(0)));
  M_CALL_SCOPE_TRACED(isEven(fib(3)));
  M_CALL_SCOPE_TRACED(isEven(fib(4)));
  M_CALL_SCOPE_TRACED(isEven(fib(6)));
}
```

- affects helper function and invoking test code
- requires another preprocessor macro

..solved using AssertionResult

Modified helper function:

```
::testing::AssertionResult isEven(int n)
{
  if ((n % 2) == 0)
    return ::testing::AssertionSuccess();
  else
    return ::testing::AssertionFailure() << n << " is odd";
}
```

- affects only helper function itself, invoking test code remains unchanged

Notes on Automated Testing

- avoid dependencies between single tests
- test code should be distinct: commit to a single issue per test
- work for full coverage of the issue and no coverage of anything else
- advoid execution of unrelated production code
- write maintainable tests and maintain them!

Automated vs. Interactive Testing

Or: Why would we need automated tests,
when we have a reliable QA that finds bugs?

Because it's not the same thing at all!

Automated tests..

- reflect the developer's perspective
- detect bugs in the internal logic of the code

QA..

- reflects the user's perspective
- reports bugs in design, UI, compatibility, interaction, etc.

Notes on Testing – revised

Production code must be written in a way that supports testing!

- think about testing opportunities when writing code, because getting rid of dependencies is hard
- testing needs real object-oriented design in the production code

But there's good news: **testable code is good code!**

Thanks for your attention!

Questions?

Literature & further reading

Google Test Advanced Guide

https://code.google.com/p/googletest/wiki/AdvancedGuide

Google Test Automation Conference

https://developers.google.com/google-test-automation-conference

Google Test Automation Conference 2013 Keynote

http://www.youtube.com/watch?v=nyOHJ4GR4iU

The Clean Code Talks - Unit Testsing
/* Thanks Timur! */

http://www.youtube.com/watch?v=wEhu57pih5w