Effective C++ Testing Using Google Test

Zhanyong Wan
Google
September, 2009

Road map

- Why tests?
- Introduction to Google Test
- What to test for, and how
- How to make your code testable
- How to write good tests
- Summary

Why tests?

- Software evolves.
  - Requirements change.
  - Bug fixes
  - Optimizations
  - Restructuring for better design
- Can you confidently make changes?
  - Must not break existing behaviors.
  - Must not introduce bugs in new behaviors.
  - Unlikely to have complete knowledge on the code.
- Writing automated tests is the *only way* in which development can scale.

But I don’t have time for tests…

- Maybe you wouldn’t be so busy if you had written tests early on.
- You really cannot afford not writing tests:
  - “(Making large changes without tests) is like doing aerial gymnastics without a net.”
    - Michael Feathers, unit tester and author of *Working Effectively with Legacy Code*
  - Bugs detected later in the development process cost orders of magnitudes more to fix.
    - They reflect in your paycheck!

Without adequate tests…

- At one point, you'll no longer be able to confidently make changes to your project.
- It will seem that there is never hope to fix all the bugs.
  - One fix creates another bug or two.
  - Have to ship with "less critical" bugs.
  - Morale is low and people want to leave the team.
- Sounds familiar?

Writing tests = time for more feature work

- How could this be true?
  - Making code testable often leads to better designs:
    - It forces you to design from the client’s perspective.
    - Testable often means nimble and reusable, allowing you to achieve more in less time.
  - If done correctly, tests shouldn’t take long to write (and you’ll get better at it).
    - Test a module in isolation.
    - Small vs large tests

Small tests, not large tests

- *Large tests*
  - A.k.a *system / integration / regression / end-to-end tests*
  - Hard to localize error
  - Run slowly
  - How many of you run *all* your tests before making a check-in?
- *Small tests*
  - A.k.a *unit tests*
  - Test a module / class / function in isolation
  - Run quickly
  - Always run before check-in
- Large tests are valuable, but *most* tests should be small tests.

Google C++ Testing Framework (aka Google Test)

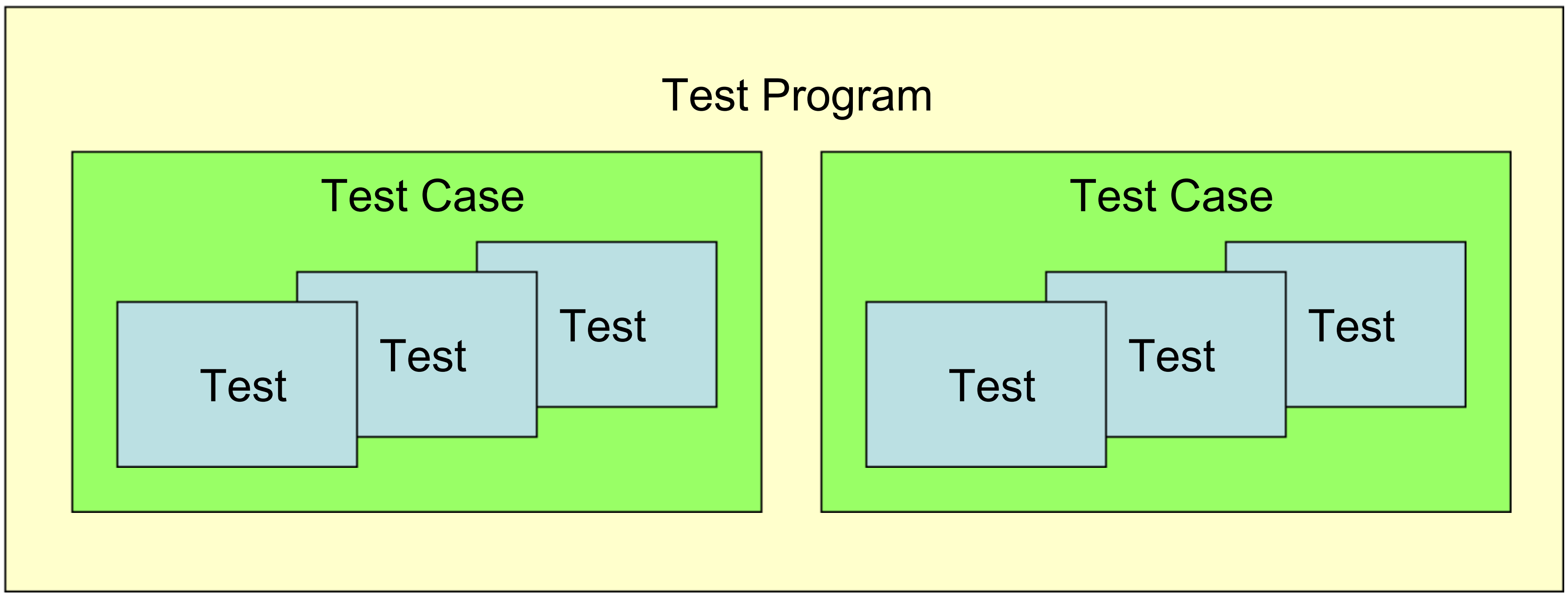
- What it is
  - A library for writing C++ tests
  - Open-source with new BSD license
  - Based on xUnit architecture
  - Supports Linux, Windows, Mac OS, and other OSes
  - Can generate JUnit-style XML, parsable by Hudson

Why Google Test

- Portable
- Easy to learn yet expressive
- Rich features
  - Add debug info to assertions using <<
  - Death tests
  - User-defined predicate assertions
  - Value/type-parameterized tests
  - Test event listener API (user-defined plug-ins)
  - Test filtering
  - Test shuffling
- Actively supported
  - Talk to googletestframework@googlegroups.com for questions or feature requests.

Google Test basics

- Basic concepts



- Each test is implemented as a function, using the TEST() or TEST_F() macro.

Simple tests

- Simple things are easy:

```
// TEST(TestCaseName, TestName)
TEST(NumberParserTest, CanParseBinaryNumber) {
  // read: a NumberParser can parse a binary number.

  NumberParser p(2); // radix = 2

  // Verifies the result of the function to be tested.
  EXPECT_EQ(0, p.Parse("0"));
  EXPECT_EQ(5, p.Parse("101"));
}
```

- TEST() remembers the tests defined, so you don't have to enumerate them later.
- A rich set of assertion macros

Reusing the same data configuration

- Define the set-up and tear-down logic in a test fixture class – you don’t need to repeat it in every test.

```
class FooTest : public ::testing::Test {
  protected:
   virtual void SetUp() { a = ...; b = ...; }
   virtual void TearDown() { ... }
   ...
};
TEST_F(FooTest, Bar) { EXPECT_TRUE(a.Contains(b)); }
TEST_F(FooTest, Baz) { EXPECT_EQ(a.Baz(), b.Baz()); }
```

- Google Test creates a fresh object for each test – tests won’t affect each other!

Google Test is organic

- Why?
  1. It’s *environment-friendly*, as it doesn’t force you to use costly C++ features (exceptions, RTTI).
  2. It prevents side effects in one test from *polluting* other tests.
  3. It helps to keep your tests *green*.
  4. The author ate exclusively *organic food* while developing it.
- Answer: A, B, and C.

What to test for: good and bad input

- Good input leads to expected output:
  - Ordinary cases
    - EXPECT_TRUE(IsSubStringOf("oo", "Google"))
  - Edge cases
    - EXPECT_TRUE(IsSubStringOf("", ""))
- Bad input leads to:
  - Expected error code – easy
  - Process crash
    - Yes, you should test this!
      - Continuing in erroneous state is *bad*.
    - But how?

Death Tests

```
TEST(FooDeathTest, SendMessageDiesOnInvalidPort) {
  Foo a;
  a.Init();
  EXPECT_DEATH(a.SendMessage(56, "test"),
               "Invalid port number");
}
```

- How it works
  - The statement runs in a *forked* sub-process.
  - Very fast on Linux
  - Caveat: side effects are in the sub-process too!

What not to test

- It’s easy to get over-zealous.
- Do not test:
  - A test itself
  - Things that cannot possibly break (or that you can do nothing about)
    - System calls
    - Hardware failures
  - Things your code depends on
    - Standard libraries, modules written by others, compilers
    - They should be tested, but not when testing your module – keep tests focused.
  - Exhaustively
    - Are we getting diminishing returns?
    - Tests should be fast to write and run, obviously correct, and easy to maintain.

How many tests are enough?

- Rule of thumb:
  - You should be able to let a new team member make a major change (new feature, bug fix, refactoring) to your code base, without fearing that existing behaviors get broken.

Make your code testable

- Often hard to test a component in isolation:
  - Components have dependencies.
  - Some components may be in another process or over the net.
  - Some components may require human interaction.
  - Slow
- Solution: break the dependencies.
  - Code to interfaces.
  - Specify the dependencies in the constructor.
  - Use *mocks* in tests.
-

```
class B : public BInterface { ... };
class A {
   public:
   A(BInterface* b) : b_(b) {}
  private:
   BInterface* b_;
};
```

Dependency injection

- Concerns:
  - A little overhead, but often acceptable – don’t optimize prematurely.
  - Inconvenient when constructing objects – the factory pattern to the rescue.
- Other ways to break dependencies:
  - Setters
    - void A::set_b(BInterface* b) { this->b_ = b; }
  - Template parameters
    - template<typename BType>
      class A { … BType b_; };
      A<B> obj;

What makes good tests?

- Good tests should:
  - Be independent
    - Don't need to read other tests to know what a test does.
    - When a test fails, you can quickly find out the cause.
    - Focus on different aspects: one bug • one failure.
  - Be repeatable
  - Run fast
    - Use mocks.
  - Localize bugs
    - Small tests
- Next, suggestions on writing better tests

Favor small test functions

- Don’t test too much in a single TEST.
  - Easy to localize failure
    - In a large TEST, you need to worry about parts affecting each other.
  - Focus on one small aspect
  - Obviously correct

Make the messages informative

- Ideally, the test log alone is enough to reveal the cause.
- Bad: “foo.OpenFile(path) failed.”
- Good: “Failed to open file /tmp/abc/xyz.txt.”
- Append more information to assertions using <<.
- *Predicate assertions* can help, too:
  - Instead of: EXPECT_TRUE(IsSubStringOf(needle, hay_stack))
  - Write: EXPECT_PRED2(IsSubStringOf, needle, hay_stack)

EXPECT vs ASSERT

- Two sets of assertions with same interface
  - EXPECT (continue-after-failure) vs ASSERT (fail-fast)
- Prefer EXPECT:
  - Reveals more failures.
  - Allows more to be fixed in a single edit-compile-run cycle.
- Use ASSERT when it doesn't make sense to continue (seg fault, trash results). Example:

```
TEST(DataFileTest, HasRightContent) {
  ASSERT_TRUE(fp = fopen(path, "r"))
     << "Failed to open the data file.";

  ASSERT_EQ(10, fread(buffer, 1, 10, fp))
     << "The data file is smaller than expected.";

  EXPECT_STREQ("123456789", buffer)
    << "The data file is corrupted.";
  ...
}
```

Getting back on track

- Your project suffers from the low-test-coverage syndrome. What should you do?
  - Every change must be accompanied with tests that verify the change.
    - Not just any tests – must cover the change
    - No test, no check-in.
    - Test only the delta.
    - Resist the temptation for exceptions.
  - Over time, bring more code under test.
    - When adding to module Foo, might as well add tests for other parts of Foo.
  - Refactor the code along the way.
- It will not happen over night, but you can do it.

Test-driven development (TDD)

- What it is:
  1. Before writing code, write tests.
  2. Write *just enough* code to make the tests pass.
  3. Refactor to get rid of duplicated code.
  4. Repeat.
- Pros:
  - Think as a client • better design
  - Clear metric for progress
  - No more untested code
- Cons:
  - May interfere with train of thought
- Should you do it?
  - I don’t care about which comes first, as long as the code is properly tested.

Resources

- Learn Google Test:
  - Homepage: http://code.google.com/p/googletest/
  - Primer: http://code.google.com/p/googletest/wiki/GoogleTestPrimer
  - Questions: googletestframework@googlegroups.com
- Dependency injection, mocks
  - Google C++ Mocking Framework (aka Google Mock): http://code.google.com/p/googlemock/

Summary

- Key points to take home:
  1. Keep tests small and obvious.
  2. Test a module in isolation.
  3. Break dependencies in production code.
  4. Test everything that can possibly break, but no more.
  5. No test, no check-in.