C++ Software Engineering

for engineers of other disciplines

Module 7

"Software Quality Assurance"

2nd Lecture: **`gtest`**

ALTEN

Spring 2022

Gothenburg, Sweden

GoogleTest

ALTEN

- A *unit testing* library for C++.
- Available across many platforms.
- Industry's *de facto standard* unit testing framework for C++.
- Tests are executable of their own and have their own **`main`** functions.
- CMake can generate **`main`**.

googletest Term	ISTQB Term
TEST()	Test Case

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

- There are extensions available for VSCode.

C++ TestMate matepek.vscode-catch2-test-adapter

Mate Pek | 33,569 | ★★★★★ | Repository | License | v3.6.4

Run GoogleTest, Catch2 and DOCtest tests from VSCode and VSCodium

Install

Features

- An xUnit test framework.
- Test discovery.
- A rich set of assertions.
- User-defined assertions.
- Death tests.
- Fatal and non-fatal failures.
- Value-parameterized tests.
- Type-parameterized tests.
- Various options for running the tests.
- XML test report generation.

https://github.com/google/googletest

- To install GoogleTest, clone the repo, and build the project using Cmake. Then make and sudo make install to install the framework on your system – the repo is located at: https://github.com/google/googletest

```
g++ test.cpp -lgtest -lgtest_main -lpthread
```

Basics

- GoogleTest is an *advanced* **`assert`**!
- It provides *MACRO*s to test through making assertions -- *failed* assertions are reported.
- Industry's *standard defacto* unit testing frame work for C++.
- There are extensions available for VSCode.

googletest Term	ISTQB Term
TEST()	Test Case

```cpp
#include <cassert>
#include <iostream>
int main() {
    int *a = nullptr;
    assert(a!=nullptr);
    *a = 12;
    return 0;
}
```

Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_TRUE(condition);`	`EXPECT_TRUE(condition);`	`condition` is true
`ASSERT_FALSE(condition);`	`EXPECT_FALSE(condition);`	`condition` is false

https://github.com/google/googletest/blob/master/googletest/docs/primer.md

- *GoogleTest's assertions* print more elaborate reports.
- *Fatal assertions* would abort execution of their relative test case i.e. the rest of the body of **`TEST()`** would not be invoked.

```
a.out: art.cpp:5: int main(): Assertion `a!=nullptr' failed.
```

Basics


```
TEST(TestSuiteName, TestName) {
  ... test body ...
}
```

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

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

actual expected

Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_EQ(val1, val2);`	`EXPECT_EQ(val1, val2);`	`val1 == val2`
`ASSERT_NE(val1, val2);`	`EXPECT_NE(val1, val2);`	`val1 != val2`
`ASSERT_LT(val1, val2);`	`EXPECT_LT(val1, val2);`	`val1 < val2`
`ASSERT_LE(val1, val2);`	`EXPECT_LE(val1, val2);`	`val1 <= val2`
`ASSERT_GT(val1, val2);`	`EXPECT_GT(val1, val2);`	`val1 > val2`
`ASSERT_GE(val1, val2);`	`EXPECT_GE(val1, val2);`	`val1 >= val2`

https://github.com/google/googletest/blob/master/googletest/docs/primer.md

- Using **`ASSERT_EQ(val1, val2)`** is preferred to **`ASSERT_TRUE(val1 == val2)`** since it prints both values in the report upon failure.

Basics

ALTEN

Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_STREQ(str1,str2);`	`EXPECT_STREQ(str1,str2);`	the two C strings have the same content
`ASSERT_STRNE(str1,str2);`	`EXPECT_STRNE(str1,str2);`	the two C strings have different contents
`ASSERT_STRCASEEQ(str1,str2);`	`EXPECT_STRCASEEQ(str1,str2);`	the two C strings have the same content, ignoring case
`ASSERT_STRCASENE(str1,str2);`	`EXPECT_STRCASENE(str1,str2);`	the two C strings have different contents, ignoring case

https://github.com/google/googletest/blob/master/googletest/docs/primer.md

- Using **`ASSERT_EQ(char*, nullptr)`** checks if the pointers are equal as the MACROs provide pointer comparison that is comparing the value of the pointers i.e. memory cells they are pointing to.

Test Fixtures

- Setting up the same *data/objects* for multiple tests.

```
TEST_F(QueueTest, IsEmptyInitially) {
  EXPECT_EQ(q0_.size(), 0);
}

TEST_F(QueueTest, DequeueWorks) {
  int* n = q0_.Dequeue();
  EXPECT_EQ(n, nullptr);

  n = q1_.Dequeue();
  ASSERT_NE(n, nullptr);
  EXPECT_EQ(*n, 1);
  EXPECT_EQ(q1_.size(), 0);
  delete n;

  n = q2_.Dequeue();
  ASSERT_NE(n, nullptr);
  EXPECT_EQ(*n, 2);
  EXPECT_EQ(q2_.size(), 1);
  delete n;
}
```

```
template <typename E>  // E is the element type.
class Queue {
 public:
  Queue();
  void Enqueue(const E& element);
  E* Dequeue();  // Returns NULL if the queue is empty.
  size_t size() const;
  ...
};
```

- For each test *fixture* object is created, its constructor and **SetUp** function are invoked, then the assertions are evaluated. At the end of the test, the **TearDown** function and *fixture*'s objects' destructor are called. The same procedure occurs for the next test.
- Using **override** keyword on **TearDown** and **SetUp** methos makes sure you are overloading the correct functions specially in c++11.

```
class QueueTest : public ::testing::Test {
 protected:
  void SetUp() override {
     q1_.Enqueue(1);
     q2_.Enqueue(2);
     q2_.Enqueue(3);
  }

  // void TearDown() override {}

  Queue<int> q0_;
  Queue<int> q1_;
  Queue<int> q2_;
};
```

https://github.com/google/googletest/blob/master/googletest/docs/primer.md

Predicate & Assertion Results

ALTEN

- Predicates provide *detailed* reports.

Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_PRED1(pred1, val1)`	`EXPECT_PRED1(pred1, val1)`	`pred1(val1)` is true
`ASSERT_PRED2(pred2, val1, val2)`	`EXPECT_PRED2(pred2, val1, val2)`	`pred2(val1, val2)` is true
`...`	`...`	`...`

```
Value of: IsEven(Fib(4))
  Actual: false (3 is odd)
Expected: true
```

```
EXPECT_TRUE(IsEven(Fib(4)))
```

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

https://github.com/google/googletest/blob/master/googletest/docs/advanced.md

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

```
Value of: IsEven(Fib(4))
  Actual: false
Expected: true
```

```
bool testStupid(int n) {
  return 0;
}
TEST(ExpectTest, HandlesZeroInput) {
  EXPECT_EQ(testStupid(0), 1);
}
TEST(PredicateTest, HandlesZeroInput) {
  ASSERT_PRED1(testStupid,0);
}
```

```
[ RUN      ] ExpectTest.HandlesZeroInput
test.cpp:53: Failure
Expected equality of these values:
  testStupid(0)
    Which is: false
  1
[  FAILED  ] ExpectTest.HandlesZeroInput (0 ms)
[----------] 1 test from ExpectTest (0 ms total)

[----------] 1 test from PredicateTest
[ RUN      ] PredicateTest.HandlesZeroInput
test.cpp:56: Failure
testStupid(0) evaluates to false, where
0 evaluates to 0
[  FAILED  ] PredicateTest.HandlesZeroInput (1 ms)
[----------] 1 test from PredicateTest (1 ms total)
```

More MACROs

Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_FLOAT_EQ(val1, val2);`	`EXPECT_FLOAT_EQ(val1, val2);`	the two `float` values are almost equal
`ASSERT_DOUBLE_EQ(val1, val2);`	`EXPECT_DOUBLE_EQ(val1, val2);`	the two `double` values are almost equal

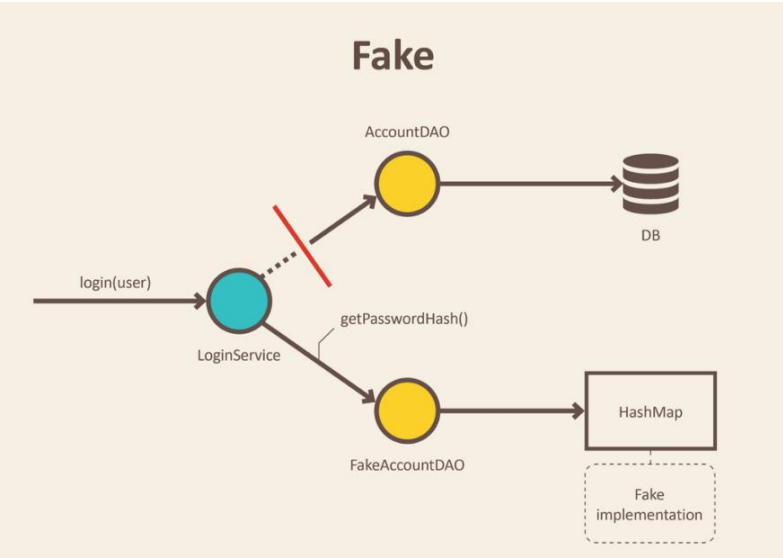
Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_NEAR(val1, val2, abs_error);`	`EXPECT_NEAR(val1, val2, abs_error);`	the difference between `val1` and `val2` doesn't exceed the given absolute error

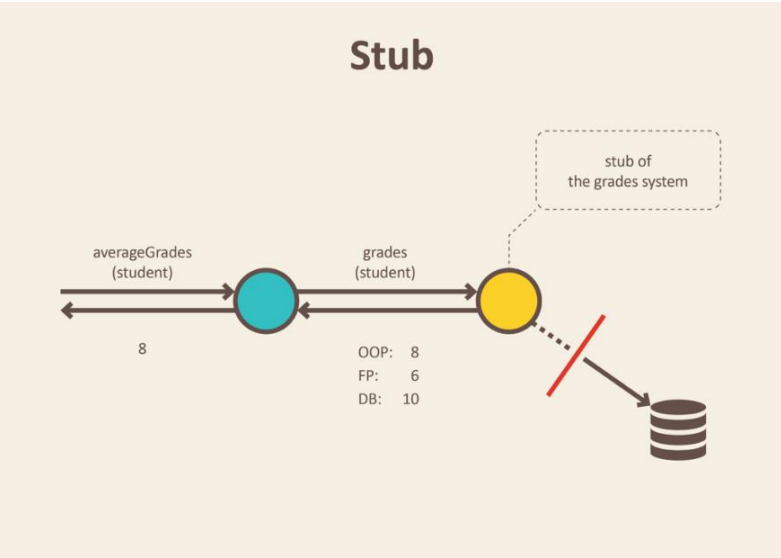
Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_PRED_FORMAT1(pred_format1, val1);`	`EXPECT_PRED_FORMAT1(pred_format1, val1);`	`pred_format1(val1)` is successful
`ASSERT_PRED_FORMAT2(pred_format2, val1, val2);`	`EXPECT_PRED_FORMAT2(pred_format2, val1, val2);`	`pred_format2(val1, val2)` is successful
`...`	`...`	...

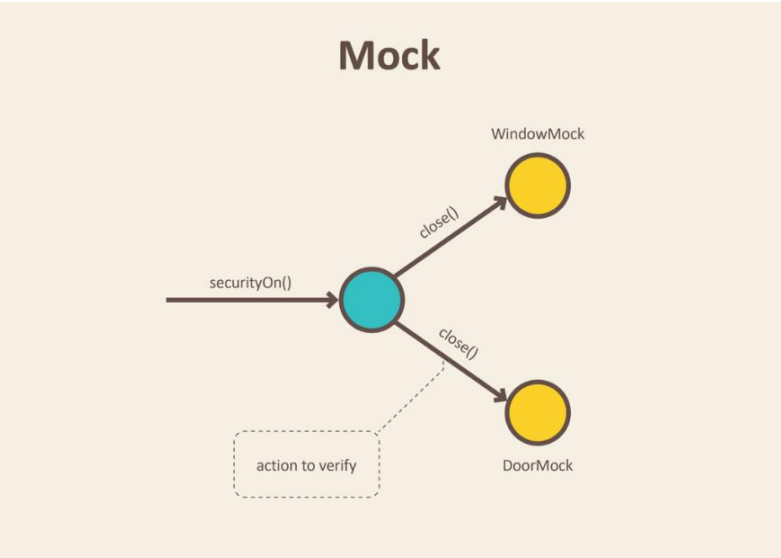
```cpp
int main(int argc, char** argv) {
  testing::InitGoogleTest(&argc, argv);
  testing::FLAGS_gtest_death_test_style = "fast";
  return RUN_ALL_TESTS();
}

TEST(MyDeathTest, TestOne) {
  testing::FLAGS_gtest_death_test_style = "threadsafe";
  // This test is run in the "threadsafe" style:
  ASSERT_DEATH(ThisShouldDie(), "");
}

TEST(MyDeathTest, TestTwo) {
  // This test is run in the "fast" style:
  ASSERT_DEATH(ThisShouldDie(), "");
}
```

Test Stunts

- Test Double is a generic term used when a production object is replaced.
- It is very common in TDD and other incremental/iterative development approaches – specially in the early stages.

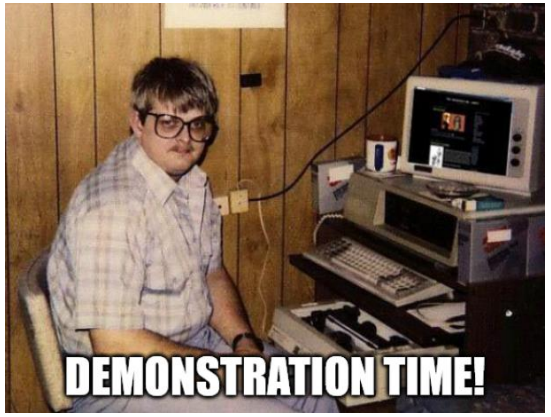






https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

DEMO!



Test Stunts

- Dummies only constructed but not used.
- Stubs could verify state by statically predefined answers.
- Spies are stubs that *log statistics* on a certain behavior – usually they are checked at some other place.
- Mock is a *spy* which provides *informal representation of* an object behavior (i.e. which function is called in what order) rather than the actual values – more interested in input than output.
- Fake tries to simulate the *business*, in a very simplified fashion.

```
class Authorizer {
public:
    virtual bool authorize(std::string username,
                           std::string password) = 0;
};
```

```
class fake_Athorizer : public Authorizer {
public:
    std::string     userid = "Davood",
                    passwd = "safest";
    bool authorize(std::string username,
                   std::string password) {
                       return   username == userid &&
                                password == passwd ;
                   }
};
```

```
class dummy_Athorizer : public Authorizer {
public:
    bool authorize(std::string username,
                   std::string password) {
                       return "Something";
                   }
};
```

```
class stub_Athorizer : public Authorizer {
public:
    bool authorize(std::string username,
                   std::string password) {
                       return true;
                   }
};
```

```
class spy_Athorizer : public Authorizer {
public:
    bool visited = false;
    bool authorize(std::string username,
                   std::string password) {
                       visited = true;
                       return true;
                   }
};
```

```
class mock_Athorizer : public Authorizer {
public:
    bool visited = false;
    bool authorize(std::string username,
                   std::string password) {
                       visited = true;
                       return true;
                   }
    bool loggedIn() {return visited;}
};
```

gMock

- Dummies, Stubs, Spies, and Mocks could be auto generated.

- Fake tries to simulate the *business*, in a very simplified fashion.

```
class MyMock {
 public:
  MOCK_METHOD(ReturnType, MethodName, (Args...));
  MOCK_METHOD(ReturnType, MethodName, (Args...), (Specs...));
};
```

```
class Turtle {
  ...
  virtual ~Turtle() {}
  virtual void PenUp() = 0;
  virtual void PenDown() = 0;
  virtual void Forward(int distance) = 0;
  virtual void Turn(int degrees) = 0;
  virtual void GoTo(int x, int y) = 0;
  virtual int GetX() const = 0;
  virtual int GetY() const = 0;
};
```

```
class MockTurtle : public Turtle {
 public:
  ...
  MOCK_METHOD(void, PenUp, (), (override));
  MOCK_METHOD(void, PenDown, (), (override));
  MOCK_METHOD(void, Forward, (int distance), (override));
  MOCK_METHOD(void, Turn, (int degrees), (override));
  MOCK_METHOD(void, GoTo, (int x, int y), (override));
  MOCK_METHOD(int, GetX, (), (const, override));
  MOCK_METHOD(int, GetY, (), (const, override));
};
```

gMock

> *"If a mock method has no EXPECT_CALL spec but is called, we say that it's an "uninteresting call", and the default action (which can be specified using ON_CALL()) of the method will be taken."*
>
> https://github.com/google/googletest/tree/main/googlemock

```
MockFoo mock_foo;
```

```
NiceMock<MockFoo> mock_foo;
```

```
StrictMock<MockFoo> mock_foo;
```

```
ON_CALL(mock_object, method(matchers))
    .With(multi_argument_matcher)  ?
    .WillByDefault(action);
```

- Return of a value
- Side effect
- Using a Function or a Functor as an Action
- Default Action
- Composite Action
- Defining Action

```
EXPECT_CALL(mock_object, method(matchers))
    .With(multi_argument_matcher)  // ?
    .Times(cardinality)            // ?
    .InSequence(sequences)         // *
    .After(expectations)           // *
    .WillOnce(action)              // *
    .WillRepeatedly(action)        // ?
    .RetiresOnSaturation();        // ?
```

```
using ::testing::Return;
...
EXPECT_CALL(turtle, GetX())
    .Times(5)                      // called 5 times
    .WillOnce(Return(100))         // return 100 first time,
    .WillOnce(Return(150))         // return 150 second time
    .WillRepeatedly(Return(200)); // return 200 the remaining times
```