Introduction to Google Software Testing

Roles

- SWE (software engineer)
  - traditional developer role
  - write functional code
  - write test code (TDD, unit tests, small/medium/large tests)
  - 100 percent of their time writing code
- SET (software engineer in test)
- TE (test engineer)

Roles

- SWE (software engineer)
- SET (software engineer in test)
  - focus on testability and general test infrastructure
  - reivew designs, code quality and risk
  - refactor code to be more testable
  - write unit testing frameworks and automation
  - increasing quality and test coverage (not adding new features, or improve performance)
  - 100 percent of their time witing code (in service of quality instead of coding features for customer)
- TE (test engineer)

Roles

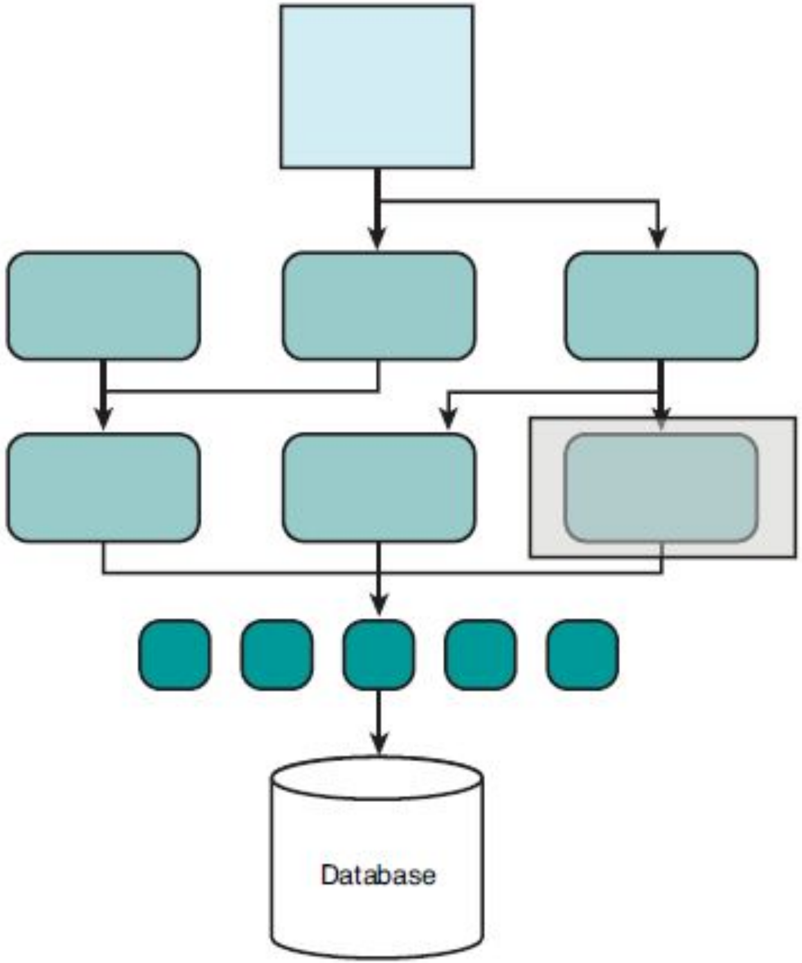
- SWE (software engineer)
- SET (software engineer in test)
- TE (test engineer)
  - like SET, with different focus
  - testing on behalf of the user first and developers second
  - writing code in the form of
    - automation scripts
    - code driving usage scenarios and mimics the user
  - late stage of the project, toward release
  - product experts, quality advisers, and analyzers of risk

Types of Tests

- Instead of code, integration, and system testing
- Google uses “small, medium, and large tests”

Small Tests

- Cover a single unit of code in a completely faked environment
- typical functional issues, data corruption, error conditions, and off-by-one mistakes
- Mostly written by a SWE, less often by an SET, and hardly by TEs
- To answer: Does this code do what it is supposed to do ? Verification



- only a single function is involved
- a single class, a small group of related functions
    - no external dependencies
    - unit tests
- focus on function operating in isolation
- provide comprehensive coverage of low-level code (large tests cannot)
- external services (file systems, networks, and database) must be mocked or faked
    - reduce external dependencies and isolated scope
    - run faster
    - by SWE

Benefits and Weekness of Small Tests

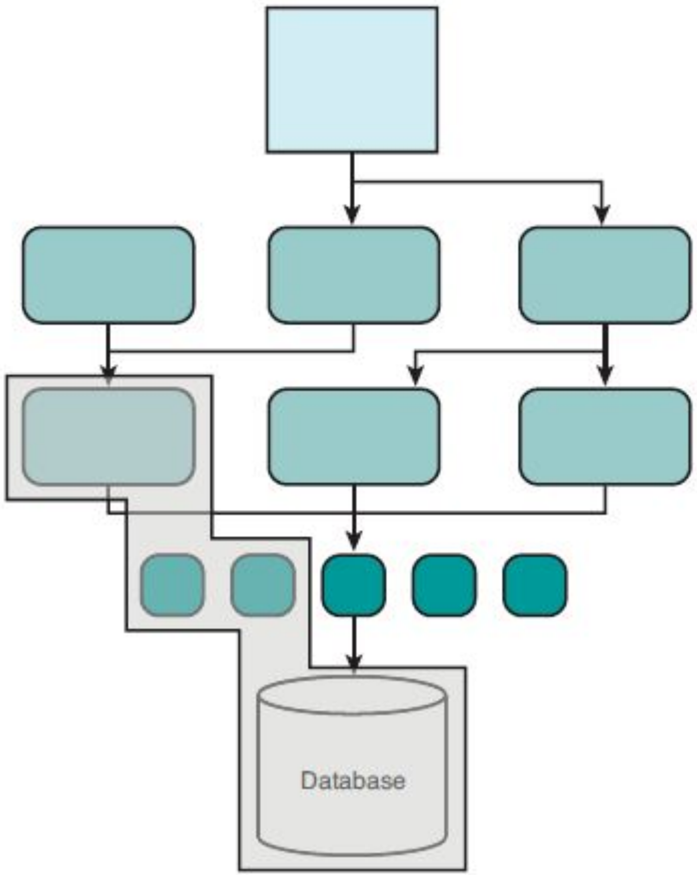
- Cleaner code and mocking requirements lead to well-defined interfaces between subsystems
- run quickly and catch bugs early
  - with immediate feedback when code changes
- run reliably in all environments
- easier testing of edge cases and error conditions, such as null pointers
- with focused scope, and isolation of errors is easy

Benefits and Weakness of Small Tests

- Don’t exercise integration between modules (and need medium tests)
- Mocking subsystems can sometimes be challenging
- Mock or fake environments can get out of sync with reality

Medium Tests

- Cover multiple and interacting units of code in a faked or real environment
- SETs develop these tests early
- To answer: Does a set of near neighbor functions interoperate with each other the way they are supposed to ?



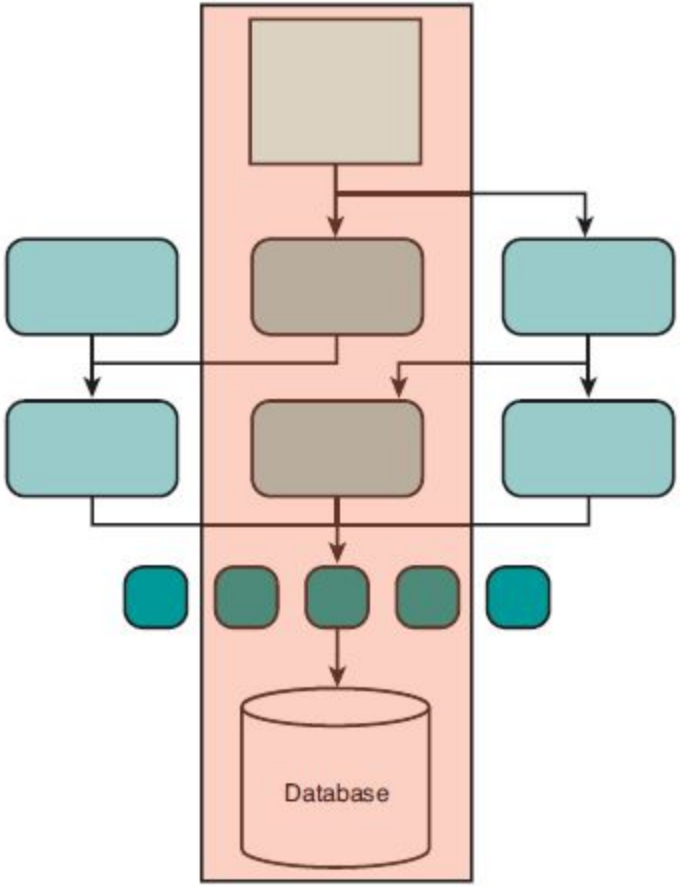
- also called integration test
- include multiple modules and can include external data sources
- testing interaction across a limited subset of modules
- Executed by SETs
- Mocking is encouraged, but not required
- Lightweight fakes such as in-memory databases

Benefits and Weakness of Medium Tests

- With looser mocking requirements and runtime limitations, provide development a stepping stone to move from large tests toward small tests
- run relatively fast, so run frequently
- run in a standard developer environment, so run easily
- can be nondeterministic because of dependencies on external systems
- slower than small tests

Large Tests

- Cover any number of units of code in the actual production environment with real and not faked resources
- Answer the question: Does the product operate the way a user would expect and produce the desired results ?
  - Validation



- also called system tests or end-to-end tests
- run any or all application subsystems from UI to backend data storage
- might make use of exernal resources: databases, file systems, and network services

Benefits and Weekness of Large Tests

- Test how the applications works and account for the behavior of external sybsystems
- can be nondeterministic because of dependencies on external systems
- broad scope and when tests fail, the cause is difficult to find
- data setup for testing scenarios is time-consuming
- impractical to exercise specific corner cases (and need small tests)

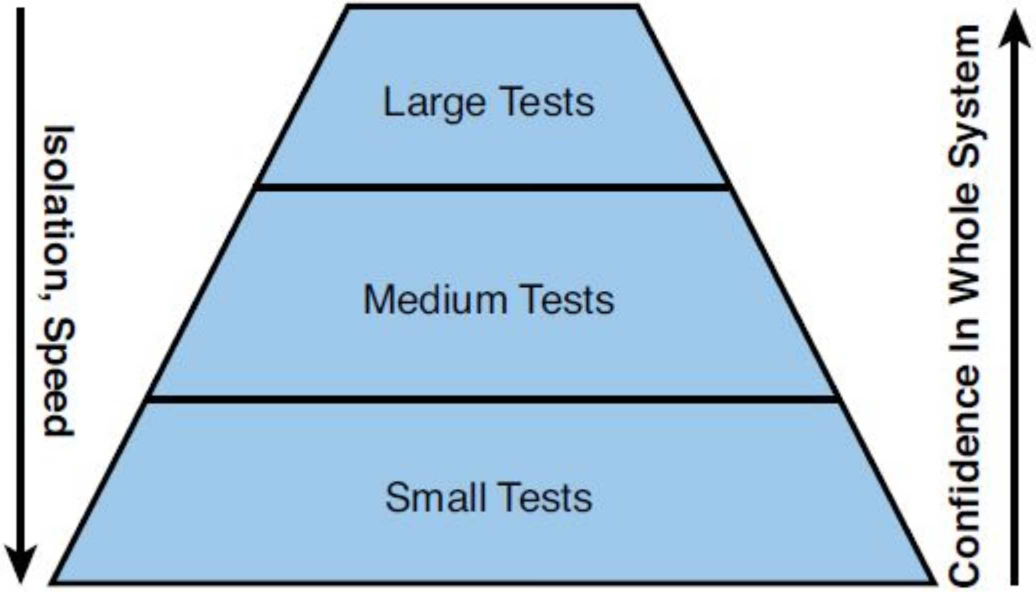
Goals and Limits of Test Execution Time by Test Size

	Small Tests	Medium Tests	Large Tests	Enormous Tests
Time Goals (per method)	Execute in less than 100 ms	Execute in less than 1 sec	Execute as quickly as possible	Execute as quickly as possible
Time Limits Enforced	Kill small test targets after 1 minute	Kill medium test targets after 5 minutes	Kill large test targets after 15 minutes	Kill enormous test targets after 1 hour

Resource Usage by Test Size

Resource	Large	Medium	Small
Network Services (Opens a Socket)	Yes	localhost only	Mocked
Database	Yes	Yes	Mocked
File System Access	Yes	Yes	Mocked
Access to User-Facing Systems	Yes	Discouraged	Mocked
Invoke Syscalls	Yes	Discouraged	No
Multiple Threads	Yes	Yes	Discouraged
Sleep Statements	Yes	Yes	No
System Properties	Yes	Yes	No

Benefits of Test Sizes



Summary of Test Sizes

- Small tests lead to code quality, good exception handling, and good error reporting (verification)
- Larger tests lead to overall product quality and data validation
- No single test size in isolation can solve all of a project’s testing needs
- It is wrong to perform only end-to-end testing framework as to provide only small unit tests for a project

Code Coverage Evaluation of mixture of size of tests

- Each project with acceptable amount of coverage in isolation
- If medium and large tests produce only 20% code coverage in isolation, while small tests provide nearly 100% coverage, the project is likely lacking in evidence that the system works end-to-end

General rules of thumb

- 70/20/10: 70% of tests are small, 20% medium and 10% large
- If projects are user-facing, high interaction, and complex user interfaces, should be with more medium and large tests
- Infrastructure or data-focused projects (such as indexing or crawling) should be with large number of small tests and fewer medium or large tests

Introduction to Google Testing Framework

- Tests are indepenent and repeatable
- Tests are well organized and reflect the structure of the test code
- Tests are portable and reusable
- When tests fail, the information about the problem is kept
- Keep track of all tests defined
- Tests are fast

Google Test

- 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.

Basic Concepts

- Writing assertions
  - check if a condition is true
  - success, nonfatal failure or fatal failure
  - if fatal failure occurs, abort the current function; otherwise continue normally
  - if a test crash or with a failed assertion, then fails ; otherwise succeeds.
- A test case contains one or many tests
- A test program contains multiple test cases

Meaning	Google Test Term	ISTQB Term
Exercise a particular program path with specific input values and verify the results	TEST()	Test Case
A set of several tests related to one component	Test Case	Test Suite

Assertions

- Assertions are macros like function calls
- test a class or function by making assertions about its behavior
- when assertion fails, print the assertion’s source file, line numbe location, and failure message. (and with user custome failure message to be appended)

Two types

- ASSERT_* version: generate fatal failures when fail and abort the current execution
  - use this version when doesn’t make sence to continue execution when the assertion fails
- EXPECT_* version: general nonfatal failures and don’t abort the current function
  - allow more than one failures to be reported in a test

Custom failure message

```
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;
}
```

Basic Assertions

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

Binary Comparison

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_EQ(*expected*, *actual*);	**EXPECT_EQ(*expected*, *actual*);**	***expected* == *actual***
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***

String Comparison

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

Simple Tests

To create a test

1. Use the TEST() macro and name a test function
2. include and use Test Assertions to check values
3. test result determined by the assertions

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

A simple example

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

A test case for this function might look like:

```
// 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));
}
```

Test Fixtures

1. Derive a class from ::testing::Test
   a. Start its body with protected: or public: for fixture members to be access from subclasses
2. Inside the class, declare any objects to be used
3. Optionally, write a default constructor or SetUp() function to prepare the objects for each test
4. Optionally, wrie a destructor or TearDown() to release resouces in SetUp()
5. Optionally, define subroutines for tests to share

- two or more tests on similar data
- reuse same configuration of objects for different tests

use TEST_F() to use a fixture

```
TEST_F(test_case_name, test_name) {
  … test body …
}
```

the test_case_name must be the name of test fixture class

TEST_F()

1. Create a fresh test fixture at runtime
2. Initialize it via SetUp()
3. run the test
4. Clean up by TearDown()
5. Delete the test fixture

tests for FIFO queue class name Queue

```
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;
  ...
};
```

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

  // virtual void TearDown() {}

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

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

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

  n = q1_.Dequeue();
  ASSERT_TRUE(n != NULL);
  EXPECT_EQ(1, *n);
  EXPECT_EQ(0, q1_.size());
  delete n;

  n = q2_.Dequeue();
  ASSERT_TRUE(n != NULL);
  EXPECT_EQ(2, *n);
  EXPECT_EQ(1, q2_.size());
  delete n;
}
```

1. Google Test constructs a QueueTest object (let's call it t1).
2. t1.SetUp() initializes t1 .
3. The first test (IsEmptyInitially) runs on t1 .
4. t1.TearDown() cleans up after the test finishes.
5. t1 is destructed.
6. The above steps are repeated on another QueueTest object, this time running the DequeueWorks test.

About RUN_ALL_TESTS()

1. Saves the state of all Google Test flags.
2. Creates a test fixture object for the first test.
3. Initializes it via `SetUp()`.
4. Runs the test on the fixture object.
5. Cleans up the fixture via `TearDown()`.
6. Deletes the fixture.
7. Restores the state of all Google Test flags.
8. Repeats the above steps for the next test, until all tests have run.

writing main()

```
#include "this/package/foo.h"
#include "gtest/gtest.h"

namespace {

// The fixture for testing class Foo.
class FooTest : public ::testing::Test {
 protected:
  // You can remove any or all of the following functions if its body
  // is empty.

  FooTest() {
    // You can do set-up work for each test here.
  }

  virtual ~FooTest() {
    // You can do clean-up work that doesn't throw exceptions here.
  }

  // If the constructor and destructor are not enough for setting up
  // and cleaning up each test, you can define the following methods:

  virtual void SetUp() {
    // Code here will be called immediately after the constructor (right
    // before each test).
  }

  virtual void TearDown() {
    // Code here will be called immediately after each test (right
    // before the destructor).
  }

  // Objects declared here can be used by all tests in the test case for Foo.
};

// Tests that the Foo::Bar() method does Abc.
TEST_F(FooTest, MethodBarDoesAbc) {
  const string input_filepath = "this/package/testdata/myinputfile.dat";
  const string output_filepath = "this/package/testdata/myoutputfile.dat";
  Foo f;
  EXPECT_EQ(0, f.Bar(input_filepath, output_filepath));
}

// Tests that Foo does Xyz.
TEST_F(FooTest, DoesXyz) {
  // Exercises the Xyz feature of Foo.
}

}  // namespace

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