Автоматическое тестирование

Кирилл Корняков (Itseez, ННГУ)
Сентябрь 2015

Содержание

Вся правда о ручном тестировании

- Ключевые термины
  - ***Тест*** *— проверка, осуществляемая "руками"*
  - ***Тест-план*** *— документ со списком проверок*
  - *Отдел тестирования*
- **Профессия ручного тестировщика умирает!**
  - *Программисты несут ответственность за качество (пишут тесты!)*
  - *Google: Software Engineer in Test*
- Ручное тестирование все еще используется для:
  - *Тестирования GUI и UX (удобства использования)*
  - *Бета-тестирование с реальными пользователями*

Автоматические тесты

- **Тест** — это "обычная" функция, реализующая некоторый сценарий использования программных сущностей.

```
#include <gtest/gtest.h>

TEST(TBitField, can_set_bit)
{
    TBitField bf(10);
    EXPECT_EQ(0, bf.GetBit(3));

    bf.SetBit(3);
    EXPECT_NE(0, bf.GetBit(3));
}
```

- **Тестовая сборка (test suite)** — приложение с тестами (обычно консольное).

Тест пишется один раз, а запускается десятки тысяч раз!

Расстановка приоритетов

1. API (h-файлы)
    - *Единственное, что по-настоящему интересует ваших пользователей*
    - *GoF*: ***Program to an interface, not an implementation***.
2. Unit-tests (cpp-файлы с тестами)
    - *Автоматический контроль корректности, на века!*
    - ***Лучшая документация!***
3. Implementation (cpp-файлы)
    - *По большому счету это ваше личное дело, как выглядит реализация при условии что интерфейс простой и понятный, а тесты полны*
    - *Главные достоинства реализации: простота и корректность*

Пример тестов на Java с использованием JUnit

```java
@Test
public void canAddNumbers()
{
    // Arrange
    ComplexNumber z1 = new ComplexNumber(1, 2);
    ComplexNumber z2 = new ComplexNumber(3, 4);

    // Act
    ComplexNumber sum = z1.add(z2);

    // Assert
    assertEquals(new ComplexNumber(4, 6), sum);
}

@Test
public void canMultiplyNumbers()
{
    // Arrange
    ComplexNumber z1 = new ComplexNumber(1, 2);
    ComplexNumber z2 = new ComplexNumber(3, 4);

    // Act
    ComplexNumber mult = z1.multiply(z2);

    // Assert
    assertEquals(new ComplexNumber(-5, 10), mult);
}
```

Фреймворки для Unit-тестирования

Значительно упрощают создание и запуск unit-тестов,
позволяют придерживаться единого стиля.

1. xUnit — общее обозначение для подобных фреймворков.
2. Бесплатно доступны для большинства языков:
    - *C/C++: CUnit, CPPUnit, GoogleTest*
    - *Java: JUnit*
    - *.NET: NUnit*
3. **Встроены в современные языки:**
    - *D, Python, Go*

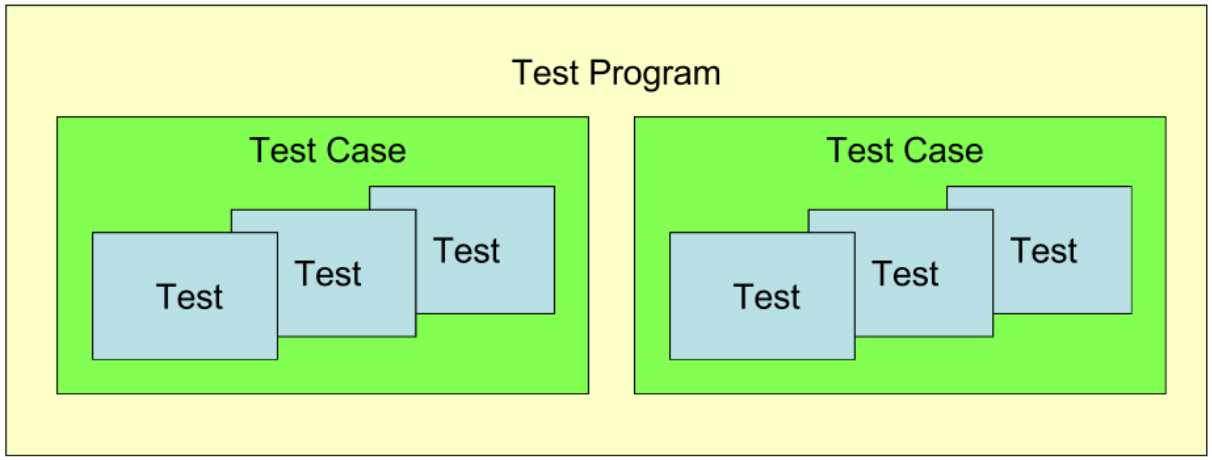
Типичные возможности

1. Удобное добавление тестов
    - *Простая регистрация новых тестов*
    - *Набор функций-проверок (`assert`)*
    - *Общие инициализации и деинициализации*
2. Удобный запуск тестов
    - *Пакетный режим*
    - *Возможность фильтрации тестов по именам*
3. Часто допускают интеграцию с IDE
4. Генерация отчета в стандартном XML-формате
    - *Возможность последующего автоматического анализа*
    - *Публикация на web-страницах проекта*

Google Test

1. Популярный фреймворк для написания модульных тестов на C++, разработанный Google.
2. Open-source проект с лицензией BSD (допускает использование в закрытых коммерческих проектах).
3. Используется в целом ряде крупных проектов
    - *Chromium, LLVM компилятор, OpenCV*
4. Написан на C++, строится при помощи CMake
    - *Поддерживает: Linux, Mac OS X, Windows, Cygwin, Windows CE, и Symbian*
5. Как правило используется в консольном режиме, но существует вспомогательное GUI приложение.

Базовые концепции



- Каждый тест реализован как функция, с использованием макроса `TEST()` или `TEST_F()`.
- `TEST()` не только определяет, но и "регистрирует" тест.

Демонстрация

Пример 1

```
#include <gtest/gtest.h>

TEST(MathTest, TwoPlusTwoEqualsFour) {
  EXPECT_EQ(2 + 2, 4);
}
```

Пример 2

Функция

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

Тесты

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

Пример 3

```
#include <gtest/gtest.h>
#include <vector>

using namespace std;

// A new one of these is created for each test
class VectorTest : public testing::Test {
public:
  vector<int> m_vector;

  virtual void SetUp() {
    m_vector.push_back(1);
    m_vector.push_back(2);
  }

  virtual void TearDown() {}
};

TEST_F(VectorTest, testElementZeroIsOne) {
  EXPECT_EQ(m_vector[0], 1);
}

TEST_F(VectorTest, testElementOneIsTwo) {
  EXPECT_EQ(m_vector[1], 2);
}

TEST_F(VectorTest, testSizeIsTwo) {
  EXPECT_EQ(m_vector.size(), (unsigned int)2);
}
```

Консольный лог Google Test

```
[mlong@n6-ws2 x86]$ bin/hellotest
Running main() from gtest_main.cc
[==========] Running 4 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 3 tests from VectorTest
[ RUN      ] VectorTest.testElementZeroIsOne
[       OK ] VectorTest.testElementZeroIsOne (0 ms)
[ RUN      ] VectorTest.testElementOneIsTwo
[       OK ] VectorTest.testElementOneIsTwo (0 ms)
[ RUN      ] VectorTest.testSizeIsTwo
[       OK ] VectorTest.testSizeIsTwo (0 ms)
[----------] 3 tests from VectorTest (0 ms total)

[----------] 1 test from MathTest
[ RUN      ] MathTest.Zero
[       OK ] MathTest.Zero (0 ms)
[----------] 1 test from MathTest (0 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 2 test cases ran. (0 ms total)
[  PASSED  ] 4 tests.
```

Полезные советы

Тесты можно временно выключать

```
TEST(MathTest, DISABLED_two_plus_two_equals_four)
{
  int x = 2 + 2;

  EXPECT_EQ(4, x);
}
```

Тесты можно фильтровать по имени при запуске

```
$ ./bin/hellotest --gtest_filter=*Vector*
```

У Google Test есть ряд других полезных опций

```
$ ./bin/hellotest --help
```

Возможности Google Test

- Automatic test discovery
- Rich set of assertions, user-defined assertions
- Death tests
- Fatal and non-fatal failures
- Value- and type-parameterized tests
- Various options for running the tests
- XML test report generation

Порядок использования Google Test

Начальная стадия

1. Скомпилировать Google Test в библиотеку.
2. Создать новое консольное приложение (test suite)
   и прилинковать к нему библиотеку Google Test.
3. Добавить тесты.
4. Скомпилировать приложение с тестами и запустить его.

Основная стадия

- Новые тесты добавляются в тот же test suite, их могут быть тысячи.
- При необходимости test suite разбивается на несколько
  - *Корректность и производительность*
  - *Быстрый (pre-commit) и полный (ночной)*

Юнит-тест курильщика

```
[Test]
public void TestMethod1()
{
   var calc = new Calculator();
   calc.ValidOperation = Calculator.Operation.Multiply;
   calc.ValidType = typeof (int);
   var result = calc.Multiply(-1, 3);
   Assert.AreEqual(result, -3);
   calc.ValidOperation = Calculator.Operation.Multiply;
   calc.ValidType = typeof (int);
   result = calc.Multiply(1, 3);
   Assert.IsTrue(result == 3);
   if (calc.ValidOperation == Calculator.Operation.Invalid)
   {
      throw new Exception("Operation should be valid");
   }
   calc.ValidOperation = Calculator.Operation.Multiply;
   calc.ValidType = typeof (int);
   result = calc.Multiply(10, 3);
   Assert.AreEqual(result, 30);
}
```

Юнит-тест здорового человека

```
[TestMethod]
public void CanAuthenticateUser() {
    var page = new TestableLoginPage();

    page.AuthenticateUser("user", "user");

    Assert.AreEqual("user", page.AuthenticatedUser);
}
```

Авторство: Антон Бевзюк, SmartStepGroup.

Критерии хорошего теста

1. Короткий (имеет чистый код)
2. Сфокусированный (только один assert)
3. Быстрый
4. Автоматический
5. Независим от порядка исполнения и окружения

Паттерн AAA: Arrange, Act, Assert

Необходимость автоматических тестов

- Развитие системы, доработка уже отлаженных компонент
- Коллективное владение
- Работа с унаследованым и сторонним кодом
- Портирование ПО на новые платформы
- Тестирование новых платформ

Testing Levels @ Google

Summary of Test Certified Levels

Level 1

Set up test coverage bundles.

Set up a continuous build.

Classify your tests as Small, Medium, and Large.

Identify nondeterministic tests.

Create a smoke test suite.

Level 2

No releases with red tests.

Require a smoke test suite to pass before a submit.

Incremental coverage by all tests >= 50%.

Incremental coverage by small tests >= 10%.

At least one feature tested by an integration test.

Level 3

Require tests for all nontrivial changes.

Incremental coverage by small tests >= 50%.

New significant features are tested by integration tests.

Level 4

Automate running of smoke tests before submitting new code.

Современная стратегия тестирования

- Без "зеленых" тестов нет уверенности в работоспособности кода
- Фокус на максимальную автоматизацию
  - *Полное тестирование требуется несколько раз в день, каждому члену команды*
- **Тесты пишутся самими разработчиками, одновременно с реализацией**
  - *Тесты это лучшая документация, которая всегда актуальна (компилятор!)*
  - *Тесты это первые сэмплы, показывающие простые примеры использования*
  - *Test-Driven Development*
- Код тестируется **непрерывно**
  - *Это делается локально на машине разработчика*
  - *Это делается на сервере до того, как добавить его в репозиторий*

Современная стратегия тестирования (2)

- Автоматические тесты замещают отладку
  - *Предсказуемость времени разработки*
  - ***Пойманный баг документируется в виде теста***
- Тесты — это "first-class citizens"
  - *Стоит отдавать код вместе с тестами*
  - *Нужно заботиться о качестве кода тестов*
  - *Метафора тестов: скелет, позволяющий организму двигаться*

Ключевые моменты

1. Пишите "жесткие" тесты, старайтесь сломать свой код.
2. Создание тестов — это составляющая самого процесса программирования.
    - *Почитайте про Test-Driven Development*
3. Без тестов нет уверенности в работоспособности кода.
4. Весь продуктовый код должен быть покрыт автоматическими тестами.
5. Автоматические тесты должны прогоняться при каждом изменении кода.
6. Ежедневно должно проводиться полное тестирование проекта,
   желательно с публикацией тестовых дистрибутивов (nightly builds).

Контрольные вопросы

1. Классификация тестов по назначению.
2. Современная стратегия тестирования (основные 5 утверждений).
3. Основные возможности фреймворков модульного тестирования.
4. Критерии хорошего теста.
5. Возможности Google Test.
6. Порядок использования Google Test.

Ссылки

1. GTest
2. Google Test Talk

План демонстрации

- Клонирование репозитория с имплементацией
- Построение и запуск сэмпла
- Запуск полного набора тестов
- Чтение тестов
- Изучение опций Google Test
  - *`--help`*
  - *`--gtest_list_tests`*
  - *`--gtest_filter_tests`*
  - *`--gtest_break_on_failure`*
  - *`--gtest_output=xml:`*
- Написание собственных тестов
  - *Создание нового файла, добавление соответствующих заголовков*
  - *Реализация простеших тестов на `TBitField`*
  - *Создание битового поля (длина 10, -10, 100000000)*
  - *Создание TestFixture*
  - *Создание параметризованного теста*

Спасибо!

Вопросы?

Классификация тестов: по масштабу

- Модульные (Unit)
- Интеграционные
  - *Инфраструктурные*
- Системные
  - *Приемочные (acceptance), функциональные*

Классификация тестов: по назначению

- Функциональные требования
  - *На задымление (smoke)*
  - *Регрессионные (regression)*
  - *На точность (accuracy)*
  - *Соответствие/совместимость (conformance/compliance)*
  - *Приемочные (acceptance)*
  - *Функциональные (functional)*
- Нефункциональные требования
  - *На производительность (performance)*
  - *Стресс (stress)*
  - *Нагрузочные (load)*
  - *Качество кода (code quality)*