

ТЕСТИРОВАНИЕ КОДА C++

1.1 Понятие тестирования кода. Дефекты

Тестирование программного кода — процесс выполнения программного кода, направленный на выявление существующих в нем дефектов. Под **дефектом** здесь и далее будем подразумевать участок программного кода, выполнение которого при определенных условиях приводит к неожиданному поведению системы (т.е. поведению, не соответствующему требованиям). Неожиданное поведение системы может приводить к сбоям в ее работе и отказам, в этом случае говорят о существенных дефектах программного кода. Некоторые дефекты вызывают незначительные проблемы, не нарушающие процесс функционирования системы, но несколько затрудняющие работу с ней. В этом случае говорят о средних или малозначительных дефектах.

Задача **тестирования** при таком подходе — определение условий, при которых проявляются дефекты системы, и протоколирование этих условий. В задачи тестирования обычно не входит выявление конкретных дефектных участков программного кода и никогда не входит исправление дефектов - это задача отладки, которая выполняется по результатам тестирования системы.

1.2 Методы тестирования

Прежде чем переходить к тестированию на практике, нужно определить, какого вида тестирование мы будем использовать. Существует несколько методов тестирования программного кода:

1. метод «чёрного ящика»;
2. метод «белого(стеклянного) ящика»;
3. тестирование моделей;
4. анализ программного кода(инспекция).

1.2.1 Метод «чёрного ящика»

Основная идея в тестировании системы как черного ящика состоит в том, что все материалы, которые доступны, — **требования на систему**, описывающие ее поведение, и **сама система**, работать с которой он может, только **подавая на ее входы некоторые внешние воздействия и наблюдая на выходах некоторый результат**. Все внутренние особенности реализации системы скрыты — таким образом, система представляет собой «черный ящик», правильность поведения которого по отношению к требованиям и предстоит проверить.

С точки зрения программного кода черный ящик может представлять собой набор классов (или модулей) с известными внешними интерфейсами, но недоступными исходными текстами.

Основная задача для данного метода тестирования состоит в последовательной проверке соответствия поведения системы требованиям. Кроме того, нужно проверить работу системы в критических ситуациях — что происходит в случае подачи неверных входных значений. В идеальной ситуации все варианты критических ситуаций должны быть описаны в требованиях на систему, и остается только придумывать конкретные проверки этих требований. Однако в реальности в результате тестирования обычно выявляется два типа проблем системы:

1. Несоответствие поведения системы требованиям.
2. Неадекватное поведение системы в ситуациях, не предусмотренных требованиями.

Отчеты об обоих типах проблем документируются и передаются разработчикам. При этом проблемы первого типа обычно вызывают изменение программного кода, гораздо реже - изменение требований. Изменение требований в данном случае может потребоваться из-за их противоречивости (несколько разных требований описывают разные модели поведения системы в одной и той же самой ситуации) или некорректности (требования не соответствуют действительности).

Проблемы второго типа однозначно требуют изменения требований ввиду их неполноты - в требованиях явно пропущена ситуация, приводящая к неадекватному поведению системы. При этом под неадекватным поведением может пониматься как полный крах системы, так и вообще любое поведение, не описанное в требованиях.

Тестирование чёрного ящика называют также тестированием по требованиям, т.к. это единственный источник информации для построения тест-плана.

1.2.2 Метод «белого(стеклянного) ящика»

При тестировании системы как стеклянного ящика имеется доступ не только к требованиям к системе, ее входам и выходам, но и к ее внутренней структуре — **видно её программный код**.

Доступность программного кода расширяет возможности тем, что появляется возможность увидеть соответствие требований участкам программного кода и определять тем самым, на весь ли программный код существуют требования. Программный код, для которого отсутствуют требования, называют кодом, не покрытым требованиями. Такой код является потенциальным

источником неадекватного поведения системы. Кроме того, прозрачность системы позволяет углубить анализ ее участков, вызывающих проблемы - часто одна проблема нейтрализует другую, и они никогда не возникают одновременно.

1.2.3 Тестирование моделей

Тестирование моделей находится несколько в стороне от классических методов верификации программного обеспечения. Причина прежде всего в том, что **объект тестирования** не сама система, а ее **модель, спроектированная формальными средствами**. Если оставить в стороне вопросы проверки корректности и применимости самой модели (считается, что ее корректность и соответствие исходной системе могут быть доказаны формальными средствами), то в нашем распоряжении есть достаточно мощный инструмент анализа общей целостности системы. На модели можно создать такие ситуации, которые невозможно создать в тестовой лаборатории для реальной системы. Работая с моделью программного кода системы, можно анализировать его свойства и такие параметры системы, как оптимальность алгоритмов или ее устойчивость.

Однако тестирование моделей не получило широкого распространения именно из-за трудностей, возникающих при разработке формального описания поведения системы. Одно из немногих исключений — системы связи, алгоритмический и математический аппарат которых достаточно хорошо проработан.

1.2.4 Анализ программного кода (инспекция)

Во многих ситуациях тестирование поведения системы в целом невозможно — отдельные участки программного кода могут никогда не выполняться, при этом они будут покрыты требованиями. Примером таких участков кода могут служить обработчики исключительных ситуаций. Если, например, два модуля передают друг другу числовые значения и функции проверки корректности значений работают в обоих модулях, то функция проверки модуля-приемника никогда не будет активизирована, т.к. все ошибочные значения будут отсечены еще в передатчике.

В этом случае выполняется ручной анализ программного кода на корректность, называемый также просмотрами или инспекциями кода. Если в результате инспекции выявляются проблемные участки, то информация об этом передается разработчикам для исправления наравне с результатами обычных тестов.

СОЗДАНИЕ ЮНИТ-ТЕСТОВ НА ПРИМЕРЕ ФРЕЙМВОРКА GTEST

В данной главе мы познакомимся библиотекой *GoogleTest* и научимся писать базовые модульные тесты на примере макросов из этого фреймворка. Про другие возможности тестирования можно почитать [здесь](#).

1.3 Понятие юнит-тестов

Модульное тестирование, или **юнит-тестирование** (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже от-тестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

1.4 Аналог программы «Hello, world!» для GTest

Как и во всех ЯП вначале рассмотрим так называемый «Hello world». Для Gtest он будет выглядеть следующим образом:

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

TEST(TestGroupName, Subtest_1) {
    ASSERT_TRUE(1 == 1);
}

TEST(TestGroupName, Subtest_2) {
    ASSERT_FALSE('b' == 'b');
    std::cout << "continue test after failure" << std::endl;
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}
```

Листинг 1.1: «Hello world» для GTest

Как видим, фреймворк активно использует макросы. В макросе TEST первый аргумент в скобках означает название группы тестов, объединенных общей логикой. Второй аргумент — название конкретного теста в подгруппе.

ASSERT_TRUE и ASSERT_FALSE — тоже макросы, реализующие т.н. «утверждения», которые будет проверять фреймворк.

Утверждения бывают:

1. успешные (success);
2. неудачные, но нефатальные (non-fatal failure);
3. неудачные, фатальные (fatal failure).

Отличия второго от третьего варианта можно понять, взглянув на код теста выше. Макросы ASSERT_FALSE и ASSERT_TRUE прерывают выполнение теста (fatal failure) и идущая следом команда уже не будет вызвана.

Такое же поведение можно наблюдать у макроса ASSERT_EQ(param1, param2) сравнивающего два своих аргумента на равенство:

```
TEST(TestGroupName, Subtest_1) {
    ASSERT_EQ(1, 2);
    std::cout << "continue test\n"; // не будет выведено на экран
}
```

Листинг 1.2: Макрос ASSERT_EQ

По-другому работает макрос EXPECT_EQ — в случае неудачи выполнение кода после него продолжится:

```
TEST(TestGroupName, Subtest_1) {
    EXPECT_EQ(1, 2); // логи покажут тут ошибку
    cout << "continue test\n"; // при этом будет выведено на экран
                                //данное сообщение
}
```

Листинг 1.3: Макрос EXPECT_EQ

Для ASSERT_ и EXPECT_ можно использовать следующие окончания:

- EQ — Equal
- NE — Not Equal

- LT — Less Than
- LE — Less than or Equal to
- GT — Greater Than
- GE — Greater than or Equal to

Окончаний на самом деле больше, т.к. при тестировании сравнивают не только целые числа. Для вещественных чисел, строк, предикатов примеры окончаний можно подглядеть [здесь](#).

1.5 Написание юнит-тестов для C/C++ в Visual Studio

Перед написанием тестов нужно настроить необходимые проекты и библиотеки в самой Visual Studio. [Тут](#) можно почитать как это сделать.

Теперь после знакомства с некоторыми из макросов и настройки проекта мы можем перейти к написанию тестов.

1.5.1 Функции *WrongMultiply* и *CorrectMultiply*

Рассмотрим две функции, которые, на первый взгляд, делают одно и то же: выполняют умножение двух чисел типа *int32_t*.

1. Функция *WrongMultiply*:

```
int32_t WrongMultiply(int32_t a, int32_t b) {
    return a * b;
}
```

Листинг 1.4: Функция WrongMultiply

2. Функция *CorrectMultiply*:

```
int64_t CorrectMultiply(int32_t a, int32_t b) {
    return static_cast<int64_t>(a) * b;
}
```

Листинг 1.5: Функция CorrectMultiply

Попробуем написать простые тесты к этим двум функциям с помощью макроса EXPECT_EQ:

```
namespace SimpleMultiplyTests {

TEST(TestWrongMultiply, SimpleTests) {
    EXPECT_EQ(WrongMultiply(0, 1), 0);
    EXPECT_EQ(WrongMultiply(1, 1), 1);
    EXPECT_EQ(WrongMultiply(-1, 1), -1);

    EXPECT_EQ(WrongMultiply(-1, -1), 1);
    EXPECT_EQ(WrongMultiply(6, 7), 42);
    EXPECT_EQ(WrongMultiply(15, 4), 60);
}

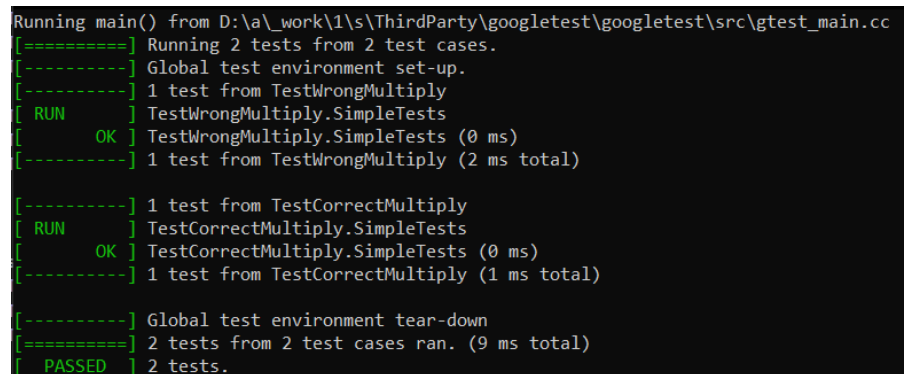
TEST(TestCorrectMultiply, SimpleTests) {
    EXPECT_EQ(CorrectMultiply(0, 1), 0);
    EXPECT_EQ(CorrectMultiply(1, 1), 1);
    EXPECT_EQ(CorrectMultiply(-1, 1), -1);

    EXPECT_EQ(CorrectMultiply(-1, -1), 1);
    EXPECT_EQ(CorrectMultiply(6, 7), 42);
    EXPECT_EQ(CorrectMultiply(15, 4), 60);
}

} // namespace SimpleMultiplyTests
```

Листинг 1.6: Базовые тесты

При запуске этих тестов видим, что обе функции их проходят:



```
Running main() from D:\a\_work\1\s\ThirdParty\googletest\googletest\src\gtest_main.cc
[=====] Running 2 tests from 2 test cases.
[-----] Global test environment set-up.
[-----] 1 test from TestWrongMultiply
[ RUN      ] TestWrongMultiply.SimpleTests
[ OK       ] TestWrongMultiply.SimpleTests (0 ms)
[-----] 1 test from TestWrongMultiply (2 ms total)

[-----] 1 test from TestCorrectMultiply
[ RUN      ] TestCorrectMultiply.SimpleTests
[ OK       ] TestCorrectMultiply.SimpleTests (0 ms)
[-----] 1 test from TestCorrectMultiply (1 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 2 test cases ran. (9 ms total)
[ PASSED  ] 2 tests.
```

Рисунок 1.1: Результаты прохождения простых тестов

Теперь рассмотрим краевые случаи. Так как мы работаем с типом *int32_t*, то может возникнуть такой случай, что при умножении двух чисел этого типа мы получим переполнение (например, можем перемножить числа INT32_MAX и INT32_MAX):

```
namespace OverflowMultiplyTests {

TEST(TestWrongMultiply, OverflowTests) {
    EXPECT_EQ(WrongMultiply(INT32_MAX, INT32_MAX),
              4611686014132420609LL);
    EXPECT_EQ(WrongMultiply(INT32_MIN, INT32_MIN),
              4611686018427387904LL);
    EXPECT_EQ(WrongMultiply(999999993, -100000019),
              -100000018299999867LL);
}

TEST(TestCorrectMultiply, OverflowTests) {
    EXPECT_EQ(CorrectMultiply(INT32_MAX, INT32_MAX),
              4611686014132420609LL);
    EXPECT_EQ(CorrectMultiply(INT32_MIN, INT32_MIN),
              4611686018427387904LL);
    EXPECT_EQ(CorrectMultiply(999999993, -100000019),
              -100000018299999867LL);
}

} // namespace OverflowMultiplyTests
```

Листинг 1.7: Краевые тесты

Результат работы программы вы можете видеть на рисунке 1.2 на следующей странице. Как видно, функция ***WrongMultiply*** не прошла краевые тесты. Из этого следует, что при реализации этой функции мы не учли переполнение (см. листинг 1.4). Об этом даже подсказывают предупреждения при сборке этого проекта.

Обратите внимание на то, как решена проблема с переполнением в функции ***CorrectMultiply*** (см. листинг 1.5). Во-первых, возвращаемое значение у функции *int64_t* вместо *int_32t*. Во вторых, мы здесь используем оператор приведения типа ***static_cast*** для приведения к типу *int_64t* только лишь **одного множителя**, а не всего выражения, т.к. в этом случае у нас бы сначала выполнялась операция умножения с переполнением, а только потом результат был приведён к *int_64t*.


```

=====] Running 4 tests from 2 test cases.
-----] Global test environment set-up.
-----] 2 tests from TestWrongMultiply
RUN      ] TestWrongMultiply.SimpleTests
OK       ] TestWrongMultiply.SimpleTests (0 ms)
RUN      ] TestWrongMultiply.OwerflowTests
C:\Users\po4ta\source\repos\multiplication\TestMultiply\test.cpp(18): error: Expected equality of these values:
WrongMultiply(2147483647i32, 2147483647i32)
  Which is: 1
4611686014132420609LL
  Which is: 4611686014132420609
C:\Users\po4ta\source\repos\multiplication\TestMultiply\test.cpp(19): error: Expected equality of these values:
WrongMultiply((-2147483647i32 - 1), (-2147483647i32 - 1))
  Which is: 0
4611686018427387904LL
  Which is: 4611686018427387904
C:\Users\po4ta\source\repos\multiplication\TestMultiply\test.cpp(20): error: Expected equality of these values:
WrongMultiply(999999993, -100000019)
  Which is: 1605511557
-100000018299999867LL
  Which is: -100000018299999867
[ FAILED ] TestWrongMultiply.OwerflowTests (2 ms)
-----] 2 tests from TestWrongMultiply (4 ms total)

-----] 2 tests from TestCorrectMultiply
RUN      ] TestCorrectMultiply.SimpleTests
OK       ] TestCorrectMultiply.SimpleTests (1 ms)
RUN      ] TestCorrectMultiply.OwerflowTests
OK       ] TestCorrectMultiply.OwerflowTests (0 ms)
-----] 2 tests from TestCorrectMultiply (1 ms total)

-----] Global test environment tear-down
=====] 4 tests from 2 test cases ran. (9 ms total)
PASSED  ] 3 tests.
[ FAILED ] 1 test, listed below:
[ FAILED ] TestWrongMultiply.OwerflowTests

1 FAILED TEST

```

Рисунок 1.2: Результаты прохождения краевых тестов

Таким образом, мы познакомились с основами юнит-тестирования.