

ООП
Семестр 2
Тестирование
Черновик

Кафедра ИВТ и ПМ
ЗабГУ

2018

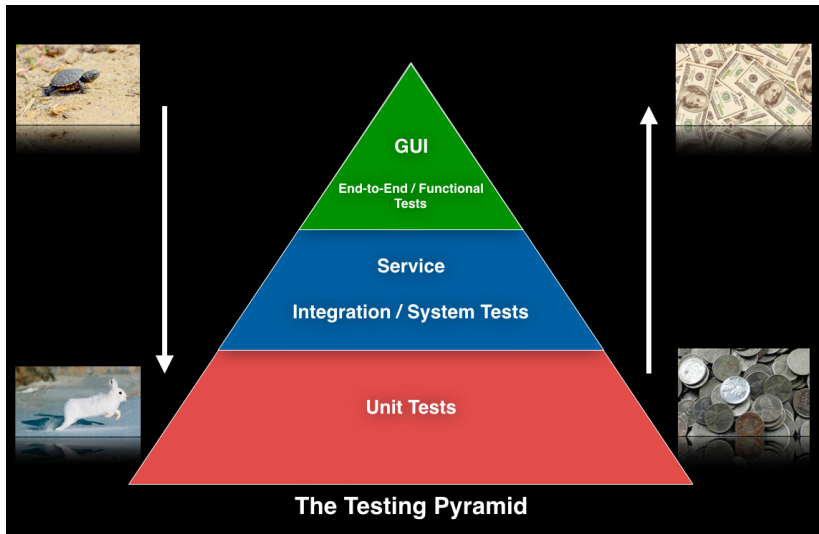
План

Модульное тестирование

Пример

Дополнительно

Ссылки и литература



Проблема

- ▶ Тестирование отдельных частей кода проще тестирования всего приложений и упрощает обнаружение ошибок
- ▶ Может требоваться проверка с большим числом вариантов исходных данных
- ▶ Как проверить корректность работы функции или метода?
- ▶ Когда нужно проверять корректность?

Проблема

- ▶ Тестирование отдельных частей кода проще тестирования всего приложений и упрощает обнаружение ошибок
- ▶ Может требоваться проверка с большим числом вариантов исходных данных
- ▶ Как проверить корректность работы функции или метода?
- ▶ Когда нужно проверять корректность?
После каждого изменения вносимого в проверяемый код.
- ▶ Изменения в функции и методы выполняющие сложную работу могут вноситься часто.
- ▶ Чтобы гарантировать корректность работы кода придётся много времени потратить на тестирование

Проблема

- ▶ Сигнатуры функций и методов изменяется гораздо реже чем их тела
- ▶ Проверка корректности как правило значительно проще чем проверяемый алгоритм
- ▶ Значим можно автоматизировать процесс тестирования написав код, который будет тестировать другой код
- ▶ Например тестируемая функция будет вызываться с заранее определёнными входными данными, а затём результат её работы будет сравниваться с эталонным

Outline

Модульное тестирование

Пример

Дополнительно

Ссылки и литература

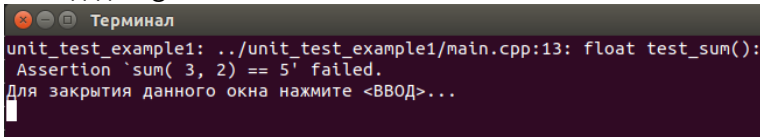
Модульное тестирование

Модульное тестирование, или юнит-тестирование (unit testing) – процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.

Разработка через тестирование (test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

Assert

- ▶ характер диагностической информации зависит от компилятора
- ▶ Вывод для gcc 7.3



The screenshot shows a terminal window titled "Терминал" (Terminal). The output text is as follows:

```
unit_test_example1: ../unit_test_example1/main.cpp:13: float test_sum():  
Assertion `sum( 3, 2) == 5' failed.  
Для закрытия данного окна нажмите <ВВОД>...
```

- ▶ имя файла и строка в который сработал assert
- ▶ имя функции в который сработал assert
- ▶ выражение внутри assert

Assert

Так как `assert` не должен присутствовать в релизной версии программы, его нужно отключать. Для этого в `cassert` используется следующий макрос

```
#ifdef NDEBUG  
#define assert(condition) ((void)0)  
#else  
#define assert(condition) /*implementation defined*/  
#endif
```

После объявления `NDEBUG` все `assert` будут отключены. Это избавляет от необходимости просматривать код и удалять `assert` вручную.

Assert

Пример

```
#include <iostream>
// uncomment to disable assert()
// #define NDEBUG
#include <cassert>

int main()
{
    assert(2+2==4);
    std::cout << "Выполнение программы продолжится\n";
    assert(2+2==5);
    std::cout << "А вот это сообщение никто не увидит\n";
}
```

Assert

Static Assertion

https://en.cppreference.com/w/cpp/language/static_assert

Простой пример

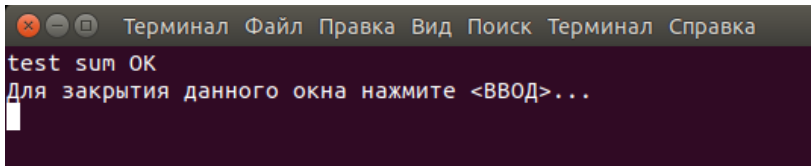
Код

```
// тестируемая функция
int sum(int x, int y){
    return x + y; }

// тестирующая функция
float test_sum(){
    // проверка работы функции на разных входных данных
    assert( sum( 3,  2) ==  5 );
    assert( sum( 2,  3) ==  5 );
    assert( sum(-3,  2) == -1 );
    assert( sum(-3, -2) == -5 );
    assert( sum( 0,  2) ==  2 );
    assert( sum( 0, -2) == -2 );
    assert( sum( 0,  0) ==  0 );
    cout << "test sum OK" << endl;
    // программа завершится с ошибкой
    // если хотя-бы одно из условий в assert будет ложным
}
```

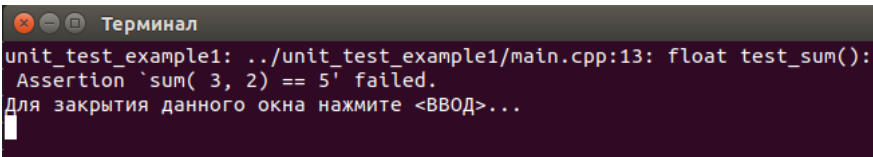
Простой пример

Результат работы программы



```
Терминал  Файл  Правка  Вид  Поиск  Терминал  Справка
test sum OK
Для закрытия данного окна нажмите <ВВОД>...
```

тест прошёл



```
Терминал
unit_test_example1: ../unit_test_example1/main.cpp:13: float test_sum():
Assertion `sum( 3, 2) == 5' failed.
Для закрытия данного окна нажмите <ВВОД>...
```

тест упал

Outline

Модульное тестирование

Пример

Дополнительно

Ссылки и литература

Пример с классом

github.com/VetrovSV/OOP/tree/master/simple_class

Преимущества юнит тестов

- ▶ Время на написание теста как правило меньше времени поиска без модульного теста
- ▶ Вероятность обнаружения ошибки после запуска теста выше чем после просмотра исходного кода
- ▶ При написании модульного теста обнаруживаются недостатки в API
- ▶ Обнаружение ошибок в контролируемом эксперименте
- ▶ Как итог, повышение надёжности программы

Outline

Модульное тестирование

Пример

Дополнительно

Ссылки и литература

Дополнительно

- ▶ Существуют отдельные фреймворки, переназначенные для создания юнит-тестов
googletest, QtTest, CppTest
- ▶ Кроме того, некоторые среды разработки (например Visual Studio) имеют отдельные шаблоны проектов для юнит-тестов и интегрировать их в существующий проект

Дополнительно

- ▶ Существуют отдельные фреймворки, переназначенные для создания юнит-тестов
googletest, QtTest, CppTest
- ▶ Кроме того, некоторые среды разработки (например Visual Studio) имеют отдельные шаблоны проектов для юнит-тестов и интегрировать их в существующий проект
- ▶ Для проверки качества кода, в дополнение к юнит-тестам используется **статические анализаторы кода**
Например cppcheck и PVS-studio (C++, C, C, Java)

Ссылки

- ▶ <http://rsdn.org/article/testing/UnitTesting.xml> Модульное тестирование: $2+2 = 4$?
- ▶ Создание юнит-теста в Visual Studio [youtube.com/watch?v=p3EUhUjv2LM](https://www.youtube.com/watch?v=p3EUhUjv2LM)

Outline

Модульное тестирование

Пример

Дополнительно

Ссылки и литература

Материалы курса

Слайды, вопросы к экзамену, задания, примеры

github.com/VetrovSV/OOP