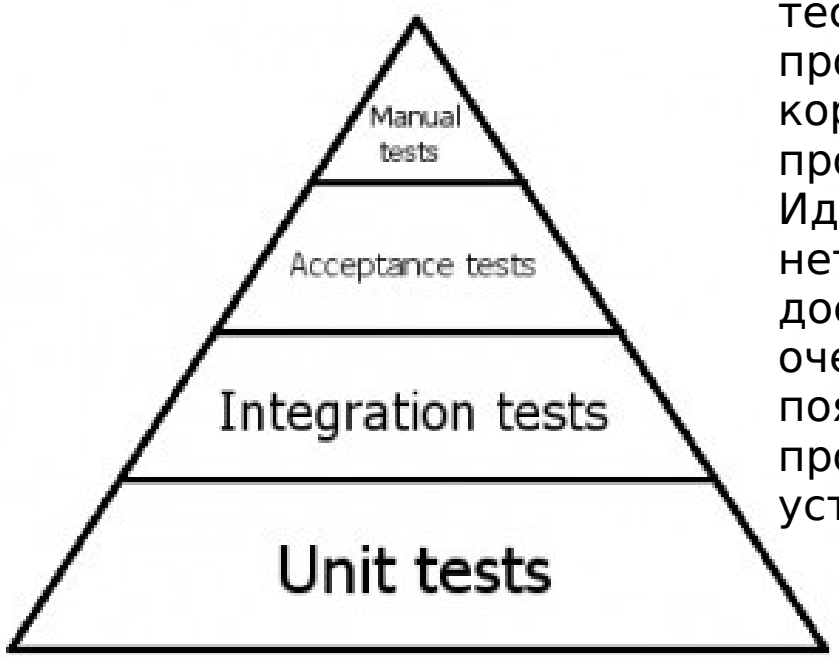
Юнит тестирование в языке Си на примере фреймворка Unity.

Introduction

Иерархии тестирования



Unit, Integration, System, Stress/Performance.

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 оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Классификация “по запущенности”

Без покрытия тестами. Обычно такие системы сопровождаются спагетти-кодом и уволившимися ведущими разработчиками. Никто в компании не знает, как именно все это работает. Да и что оно в конечном итоге должно делать, сотрудники представляют весьма отдаленно.

С тестами, которые никто не запускает и не поддерживает. Тесты в системе есть, но что они тестируют, и какой от них ожидается результат, неизвестно. Ситуация уже лучше. Присутствует какая-никакая архитектура, есть понимание, что такое слабая связанность. Можно отыскать некоторые документы. Скорее всего, в компании еще работает главный разработчик системы, который держит в голове особенности и хитросплетения кода.

С серьезным покрытием. Все тесты проходят. Если тесты в проекте действительно запускаются, то их много. Гораздо больше, чем в системах из предыдущей группы. И теперь каждый из них – атомарный: один тест проверяет только одну вещь. Тест является спецификацией метода класса, контрактом: какие входные параметры ожидает этот метод, и что остальные компоненты системы ждут от него на выходе. Таких систем гораздо меньше. В них присутствует актуальная спецификация. Текста немного: обычно пара страниц, с описанием основных фич, схем серверов и getting started guide’ом. В этом случае проект не зависит от людей. Разработчики могут приходить и уходить. Система надежно протестирована и сама рассказывает о себе путем тестов

Почему есть проекты второго типа?

Ваши тесты должны:

Быть достоверными

Не зависеть от окружения, на котором они выполняются

Легко поддерживаться

Легко читаться и быть простыми для понимания (даже новый разработчик должен понять что именно тестируется)

Соблюдать единую конвенцию именования

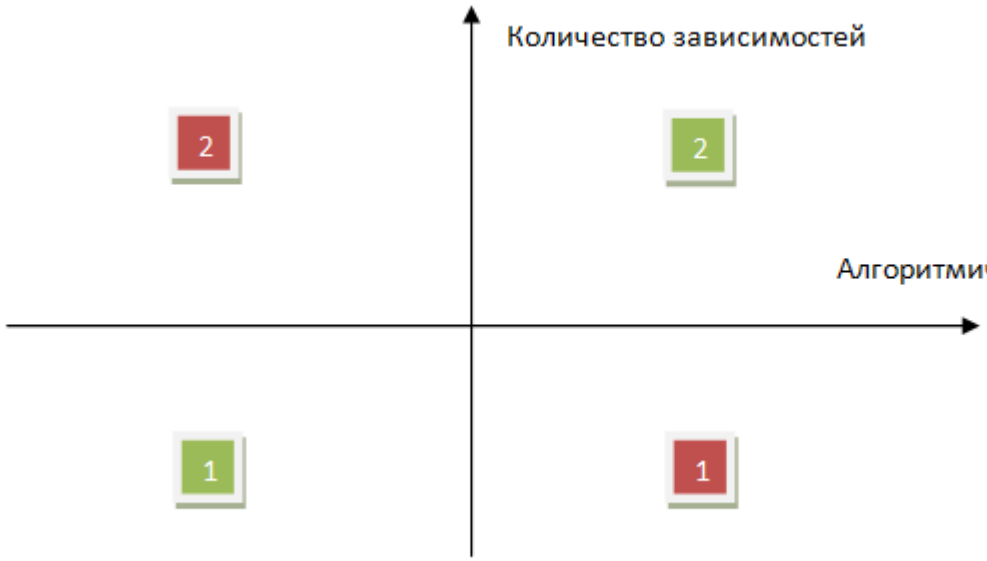
Запускаться регулярно в автоматическом режиме

Что тестировать, а что – нет?

Одни говорят о необходимости покрытия кода на 100%, другие считают это лишней тратой ресурсов.

Идеален будет такой подход: расчертите лист бумаги по оси X и Y, где X – алгоритмическая сложность, а Y – количество зависимостей. Ваш код можно разделить на 4 группы

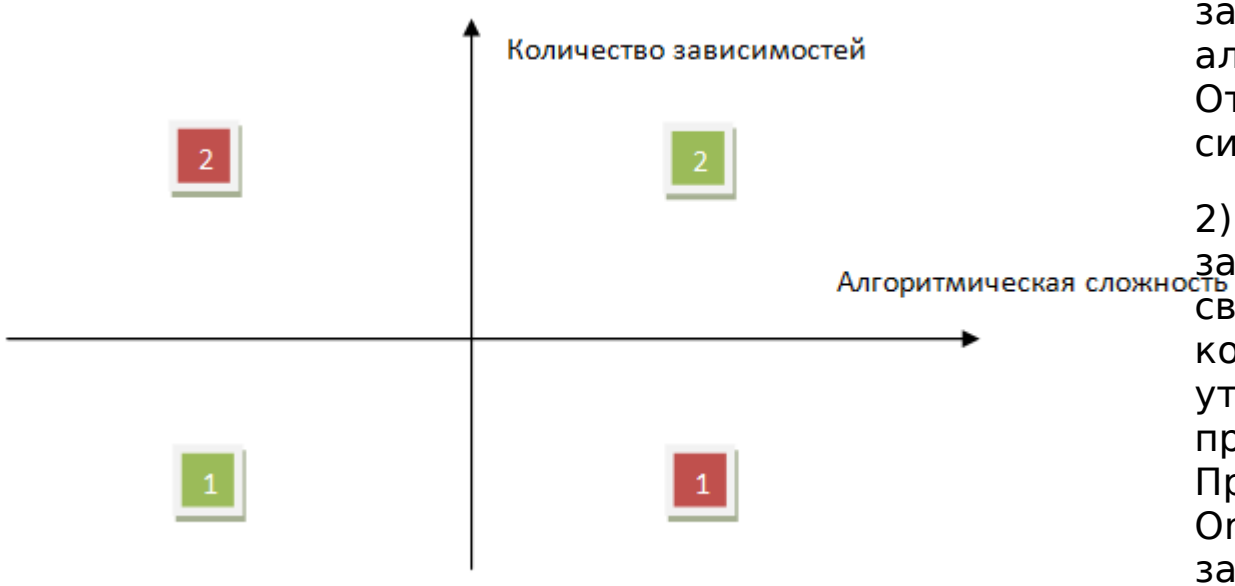
Зеленое



1)Простой код без зависимостей. Скорее всего здесь и так все ясно. Его можно не тестировать.

2)Сложный код с большим количеством зависимостей. Тут пахнет сильной связностью. Скорее всего, неплохо будет провести рефакторинг. Мы не станем покрывать этот код юнит-тестами, потому что перепишем его, а значит, у нас изменятся сигнатуры методов и появятся новые классы. Так зачем писать тесты, которые придется выбросить? Хочу оговориться, что для проведения такого рода рефакторинга нам все же нужно тестирование, но лучше воспользоваться более высокоуровневыми приемочными тестами. Мы рассмотрим этот случай отдельно.

Красное



Что у нас остается:

1)Сложный код без зависимостей. Это некие алгоритмы или бизнес-логика. Отлично, это важные части системы, тестируем их.

2)Не очень сложный код с зависимостями. Этот код связывает между собой разные компоненты. Тесты важны, чтобы уточнить, как именно должно происходить взаимодействие. Причина потери Mars Climate Orbiter 23 сентября 1999 года заключалась в программно-человеческой ошибке: одно подразделение проекта считало «в дюймах», а другое – «в метрах», и прояснили это уже после потери аппарата. Результат мог быть другим, если бы команды протестировали «швы» приложения.

Language Comparison

Мир языка C++ не такой дружественный к тестированию, как например, мир Java, C# или мир интерпретаторов. Главная причина — крайне слабый механизм интроспекции, то есть возможности исследования двоичного кода в плане получения информации о структуре исходных текстов. В Java, например, есть "The Reflection API", с помощью которого можно прямо на основе скомпилированных классов создать тестовую среду (понять иерархию классов, типа аргументов и т.д.). В C++ приходится многое закладывать в исходный текст на этапе его создания, чтобы облегчить будущее тестирование.

Language Comparison

А что же мы имеем в C? Тут, разрыв в удобстве тестирования по отношению к C++ в разы больше, чем между C++ и Java, например. Причин море: процедурная модель вместо объектно-ориентированной, отсутствие интроспекции вообще, крайне слабая защита при работе с памятью и т.д.

Language Comparison

Например, есть библиотека MinUnit, длиной в четыре строки. ROFL LMAO LMFAO :D :D :D
http://www.jera.com/techinfo/jtns/jtn002.html

```
/* file: minunit.h */

#define mu_assert(message, test) do { if (!(test)) return message; } while (0)

#define mu_run_test(test) do { char *message = test(); tests_run++; \
                               if (message) return message; } while (0)

extern int tests_run;
```

Why UT in C is not simple enough

```
#define  SUBSYSTEM Test
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>
#include "parse_my_url.h"

#ifndef UNIT_TEST
#include "system_text_logging.h"
#include "http_private_data.h"
#endif
```

Why UT in C is not simple enough

Как вы заметили, в коде есть специальный блок, ограниченный макросом UNIT_TESTING. В языке С приходится "готовить" код к потенциальному тестированию и вставлять фрагменты, позволяющие тестовой среде работать с этим кодом. Этот блок, если задан макрос UNIT_TESTING, отключается чтобы во внутренних файлах не переопределять структуры файловых дескрипторов например.

Why UT in C is not simple enough

Схема очень похожа на любое другое xUnit тестирование: каждый тест проверяет какой-то один функциональный элемент, тесты объединяются в группы и запускаются автоматически все вместе. Правда, из-за ограничений языка С каждый тест приходится вручную добавлять в список запуска, увы.

Unity Framework

```
int main(void)
{
    UNITY_BEGIN();
// now setUp
    RUN_TEST(first_function_here);
    RUN_TEST(another_function_a);
//now TearDown
    return UNITY_END();
}
```

#include <unity.h>

```
void setUp(char* k) { k=malloc(256)}
void tearDown(char *k) { free(k) }

    for (i = 0; i < sizeof(cases)/sizeof(cases[0]); i++) {

        bool ret = http_parse_url(cases[i].url, host, host_size,
                                  &port, filename, filename_size, &is_secure);

        TEST_ASSERT_TRUE(ret == cases[i].ret);

        if (cases[i].ret) {
            TEST_ASSERT_EQUAL_STRING(cases[i].host, host);
            TEST_ASSERT_EQUAL_UINT16(cases[i].port, port);
            TEST_ASSERT_EQUAL_STRING(cases[i].path, filename);
            TEST_ASSERT_TRUE(cases[i].is_secure == is_secure);
        }
    }
```

Samples

```
typedef struct {
    const char *url;     bool ret;     bool is_secure;     const char *host;     uint16_t port;     const char *path;
} testcase_t;

testcase_t cases[] = {
        {"http://onliner.by:1234/abcd", true, false, "onliner.by", 1234, "abcd"},
        {"ttp://tut.by:4455/path", false },
        {"http://tut.by:60100/path", true, false, "tut.by", 60100, "path"},
        {"https://[2001:db8:0:1]:80/qwe.txt", true, true, "2001:db8:0:1", 80, "qwe.txt" },
        {"http://tut.by:44ba/", false },
        {"http://tut.by:60100/path", true, false, "tut.by", 60100, "path"},
        {"http://12345678901234567890123456789:100/12345678901234567890123456789", true, false,
        "12345678901234567890123456789", 100, "12345678901234567890123456789"},
        {"http://12345678901234567890123456789:100/123456789012345678901234567890", false, false,
        "123456789012345678901234567890", 100, "123456789012345678901234567890"},
        {"http://12345678901234567890123456789:100/1234567890123456789012345678901", false, false,
        "123456789012345678901234567890", 100, "1234567890123456789012345678901"},
```

Unity workaround

throwtheswitch.org

CMock — инструмент позволяющий автоматически генерировать Си-код mock-ов для Ваших тестов. Написан на Ruby. Но использовать его автономно без следующего инструмента, на мой взгляд, не рационально.
Ceedling — это целая билд-система, как утверждают сами авторы. Но по сути — это все, что Вам нужно для работы. Данный пакет содержит в себе все необходимое: Unity («тест-раннеры» и «чекалки» значений), CMock (генератор моков) и поддержку командной строки через ruby make.
Other — под этим заголовком находится очень, полезный инструмент — CException. Маленькая библиотека для Си позволяющая получить некое подобие исключений.

ceedling new NewProject

В результате будет создана папка NewProject внутри которой будут сгенерированы следующие папки и файлы:

build — сюда будут помещаться все артефакты при сборке и прогоне тестов

src- это место для нашего «боевого» кода, который подлежит тестированию

test — будут лежать все наши тесты

vendor — собственно сам framework, с документацией и плагинами

project.yml — конфигурационный файл тестового проекта. Позволяет делать хороший тюнинг, но это с опытом

ls test/test_calc.c

```
#include "unity.h"
#include "calc.h"
void setUp(void) {}
void tearDown(void) {}
void test_add(void)
{
    int result = 0;
    result = calc_add(2,2);
    TEST_ASSERT_EQUAL_INT( 4, result );
}
```

RUN WITH

```
# ceedling test:test_calc.c
```

ls src/calc.c

«calc.h»

```
#ifndef CALC_H
#define CALC_H
int calc_add(int a, int b);
#endif
```

«calc.c»

```
#include "calc.h"
int calc_add(int a, int b)
{
    return a + b;
}
```

ceedling test:test_calc.c

```
Test 'test_calc.c'
------------------
Compiling test_calc_runner.c...
Compiling test_calc.c...
Compiling calc.c...
Compiling unity.c...
Compiling cmock.c...
Linking test_calc.out...
Running test_calc.out...

-----------------------
OVERALL UNIT TEST SUMMARY
-----------------------
TESTED:  1
PASSED:  1
FAILED:  0
IGNORED: 0
```

Test Driven Development for Embedded C (Pragmatic Programmers)

Спасибо за внимание :)