Тестирование программного обеспечения. Библиотеки для тестирования C/C++ программ

Данилов Борис Радиславович

Осень, 2017

Тестирование программного обеспечения. Определение

Тестирование (Т.) программного обеспечения (ПО) — это процесс исследования, испытания программного продукта, имеющий две цели:

- продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям;
- выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации.

Тестирование ПО. Немного истории...

- Т. первых систем (научные исследования, программы для нужд министерств обороны) проводится строго формализованно с записью всех тестовых процедур, тестовых данных, полученных результатов. Т. выделяется в отдельный процесс, начинающийся после завершения кодирования.
- 1960-е гг. Уделяется много внимания «исчерпывающему» Т., которое должно проводиться с использованием всех путей в коде или всех возможных входных данных. Отмечается, что полное Т. ПО невозможно, потому что 1) количество возможных входных данных очень велико, 2) существует множество путей выполнения программы, 3) сложно найти проблемы в архитектуре и спецификациях. «Исчерпывающее» Т. отклонено и признано теоретически невозможным.

Тестирование ПО. Немного истории...

- Начало 1970-х гг. Т. ПО — «процесс, направленный на демонстрацию корректности продукта» или как «деятельность по подтверждению правильности работы программного обеспечения». Верификация ПО — «доказательство правильности». Перспективная концепция, но на практике требует много времени и недостаточно всеобъемлющая. Решено, что доказательство правильности — также неэффективный метод Т. ПО.

- Вторая половина 1970-х гг. Т. — выполнение программы с намерением найти ошибки, а не доказать, что она работает. Успешный тест — это тест, который обнаруживает ранее неизвестные проблемы. Данный подход прямо противоположен предыдущему.

Тестирование ПО. Немного истории...

- 1980-е гг. Т. расширяется понятием предупреждение дефектов. Высказываются мысли, что необходима методология Т., в частности, что Т. должно включать проверки на всем протяжении цикла разработки, и это должен быть управляемый процесс. В ходе Т. надо проверить не только собранную программу, но и требования, код, архитектуру, сами тесты. «Традиционное» Т., существовавшее до начала 1980-х, относилось только к скомпилированной, готовой системе (сейчас это — системное тестирование), но в дальнейшем тестировщики вовлекаются во все аспекты жизненного цикла разработки.

- Середина 1980-х гг. Появляются первые крайне простые инструменты для автоматизированного Т.

Тестирование ПО. Немного истории...

- Начало 1990-х гг. В понятие «Т.» включают планирование, проектирование, создание, поддержку и выполнение тестов и тестовых окружений. Переход от тестирования к обеспечению качества, охватывающего весь цикл разработки ПО. Появление различных инструментов для поддержки процесса Т.: более продвинутые среды для автоматизации с возможностью создания скриптов и генерации отчетов, системы управления тестами, ПО для проведения нагрузочного Т.

- Середина 1990-х гг. С развитием Интернета и разработкой большого количества веб-приложений особую популярность получает «гибкое Т.» (по аналогии с гибкими методологиями программирования).

Классификация видов тестирования ПО

По объекту тестирования

- Функциональное (Functional) — проверка заранее указанного поведения на основе спецификации функциональности компонента или всей системы (*что именно* делает ПО).
- Производительности (Performance) — определение, как быстро работает система или её часть под определённой нагрузкой.
  - Нагрузочное (Load) — оценка поведения приложения под заданной ожидаемой нагрузкой.
  - Стресс (Stress) — во время экстремальных нагрузок.
  - Стабильности и надёжности (Stability/Reliability) — ожидаемая нагрузка в течение длительного времени.
  - Конфигурационное (Configuration) — эффект влияния на производительность изменений в конфигурации; может совмещаться с предыдущими тремя.
  - Объёмное (Volume) — оценка производительности при увеличении объёмов данных в базе данных приложения.

Классификация видов тестирования ПО

По объекту тестирования

- Безопасности (Security and Access Control) — оценка уязвимости ПО к различным атакам; анализ рисков, связанных с обеспечением целостного подхода к защите приложения.
- Удобства использования (Usability) — проверка эргономичности объекта или системы.
  - Пользовательского интерфейса (User Interface) — удобен ли объект для предполагаемого применения.
  - Удобства использования (User eXperience) — измерение ощущений, испытываемых пользователем во время использования цифрового продукта.
- На отказ и восстановление (Failover and Recovery) — проверка продукта противостоять сбоям и успешно восстанавливаться.
- Установки (Installation) — проверка инсталляции, настройки, обновления и удаления ПО.

Классификация видов тестирования ПО

По степени изолированности

- Компонентов (Unit) — проверка минимально возможных для Т. компонент: функций, классов, методов и т.п.
- Интеграционное (Integration) — Т. более крупных компонент системы; выполняется через их интерфейс в виде Т. «чёрного ящика».
- Системное (System) — тестируется интегрированная система на её соответствие требованиям.
  - Альфа (Alpha) — имитация реальной работы на ранней стадии разработки.
  - Бета (Beta) — интенсивное использование почти готовой версии продукта с целью выявления максимального числа ошибок в его работе.
- Операционное (Release) — убедиться в том, что ПО удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации.

Классификация видов тестирования ПО

По связи с изменениями в ПО в процессе разработки

- Дымовое (Smoke, Sanity) — подтверждения того, что после сборки кода приложение стартует и выполняет основные функции.
- Повторное (Re-testing) — исполнение тестовых сценариев, выявивших ошибки во время последнего запуска, для подтверждения успешности исправления этих ошибок.
- Регрессионное (Regression) — подтверждение того факта, что существующая ранее функциональность работает как и прежде.

Классификация видов тестирования ПО

А также по...

...знанию системы:

- Чёрного/Белого ящика (Black/White-box)

...степени автоматизации:

- Ручное/Автоматизированное (Manual/Automated)

...исполнению программы:

- Статическое/Динамическое (Static/Dynamic) — к статическому также относят анализ требований, спецификаций, документации.

...другим критериям.

Библиотеки для тестирования C/C++ программ

- Catch
  https://github.com/philsquared/Catch
- Boost.Test
  http://www.boost.org/doc/libs/1_49_0/libs/test/doc/html/index.html
- Google Test
  https://github.com/google/googletest
- CppUnit
  https://sourceforge.net/projects/cppunit
- И другие...
  https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C.2B.2B

Catch. Плюсы и минусы

- Только заголовочные файлы.
- Поддерживает написание юнит-тестов по технике разработки через поведение (BDD, Behaviour Driven Development).
- Легка в освоении.
- Не поддерживает Mock-объектов (дословно «объект-пародия», «объект-имитация»).
- Один макрос REQUIRE для почти любого сравнения. Легка в использовании.

Boost.Test. Плюсы и минусы

- Поддерживает управляемые данными тесты (DDT, Data Driven Tests). Полезно для случайной генерации тестовых данных.
- Начиная с версии 1.60, поддерживает подобно Catch один макрос BOOST_CHECK.
- Нет поддержки Mock-объектов, но можно использовть boost.turtle.
- Есть какая-то поддержка «заголовочных» вариантов библиотеки, но лучше использовать «бинарную» версию.

Google Test. Плюсы и минусы

- Поддержка тестов на отказ (Death Tests), намеренно приводящих к критическим сбоям таким как ошибки сегментации и др.
- Поставляется вместе с Google Mock.
- Не поддерживается C++11 move-семантика. Есть обходные пути, но может оказаться неудобным.
- Необходимо встраивание в проект в виде компилируемого вместе с ним исходного кода.
- Множество различных макросов, необходимо помнить немного больше чем с Boost.Test и Catch.

Catch

Туториал: https://github.com/philsquared/Catch/blob/master/docs/tutorial.md

```
// Tells Catch to provide a main() - only do this in one cpp file
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

unsigned int Factorial(unsigned int number) {
        return number <= 1 ? number : Factorial(number - 1)*number;
}

TEST_CASE("Factorials are computed", "[factorial]") {
        REQUIRE(Factorial(1) == 1);
        REQUIRE(Factorial(2) == 2);
        REQUIRE(Factorial(3) == 6);
        REQUIRE(Factorial(10) == 3628800);
}
```

Catch

```
TEST_CASE("Factorials are computed", "[factorial]") {
        REQUIRE(Factorial(0) == 1);
        REQUIRE(Factorial(1) == 1);
        REQUIRE(Factorial(2) == 2);
        REQUIRE(Factorial(3) == 6);
        REQUIRE(Factorial(10) == 3628800);
}
```

```
Example.cpp:9: FAILED:
  REQUIRE( Factorial(0) == 1 )
with expansion:
  0 == 1
```

```
unsigned int Factorial( unsigned int number ) {
  return number > 1 ? Factorial(number-1)*number : 1;
}
```

Catch

```
TEST_CASE( "vectors can be sized and resized", "[vector]" ) {

    std::vector<int> v( 5 );

    REQUIRE( v.size() == 5 );
    REQUIRE( v.capacity() >= 5 );

    SECTION( "resizing bigger changes size and capacity" ) {
        v.resize( 10 );

        REQUIRE( v.size() == 10 );
        REQUIRE( v.capacity() >= 10 );
    }
    SECTION( "resizing smaller changes size but not capacity" ) {
        v.resize( 0 );

        REQUIRE( v.size() == 0 );
        REQUIRE( v.capacity() >= 5 );
    }
...
```

Catch

```
...
    SECTION( "reserving bigger changes capacity but not size" ) {
        v.reserve( 10 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 10 );
    }
    SECTION( "reserving smaller does not change size or capacity" ) {
        v.reserve( 0 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );
    }
}
```

Catch

```
    SECTION( "reserving bigger changes capacity but not size" ) {
        v.reserve( 10 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 10 );

        SECTION( "reserving smaller again does not change capacity" ) {
            v.reserve( 7 );

            REQUIRE( v.capacity() >= 10 );
        }
    }
```

Catch. Test and Behaviour Driven Development

Разработка через тестирование (TDD, Test Driven Development) — техника разработки ПО, 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.

Разработки по поведению (BDD, Behaviour Driven Development) — одна из разновидностей разработки через тестирование, комбинирующая эти техники с принципами проблемно- и объектно-оринентированного проектирования, поддерживающая более тесное взаимодействие между разработчиками и менеджерами проекта.

Catch. Behaviour Driven Development

```
Story: Returns go to stock

As a store owner
In order to keep track of stock
I want to add items back to stock when they're returned.

Scenario 1: Refunded items should be returned to stock
Given that a customer previously bought a black sweater from me
And I have three black sweaters in stock.
When he returns the black sweater for a refund
Then I should have four black sweaters in stock.

Scenario 2: Replaced items should be returned to stock
Given that a customer previously bought a blue garment from me
And I have two blue garments in stock
And three black garments in stock.
When he returns the blue garment for a replacement in black
Then I should have three blue garments in stock
And two black garments in stock.
```

Catch. Behaviour Driven Development

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {

    GIVEN( "A vector with some items" ) {
        std::vector<int> v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( "the size is increased" ) {
            v.resize( 10 );

            THEN( "the size and capacity change" ) {
                REQUIRE( v.size() == 10 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "the size is reduced" ) {
            v.resize( 0 );

            THEN( "the size changes but not capacity" ) {
                REQUIRE( v.size() == 0 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
...
```

Catch. Behaviour Driven Development

```
...
        WHEN( "more capacity is reserved" ) {
            v.reserve( 10 );

            THEN( "the capacity changes but not the size" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "less capacity is reserved" ) {
            v.reserve( 0 );

            THEN( "neither size nor capacity are changed" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
    }
}
```

```
Scenario: vectors can be sized and resized
     Given: A vector with some items
      When: more capacity is reserved
      Then: the capacity changes but not the size
```

Catch. Как выглядит в более менее реальном коде

```
// tests-main.cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

```
// tests-factorial.cpp
#include "catch.hpp"

#include "factorial.h"

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

Спасибо за внимание!