Тема 10. Unit тестирование. Библиотека Google Test

ФРАГМЕНТЫ МАТЕРИАЛОВ ЛЕКЦИИ

1

Тестирование кода

Тест — это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода.

Когда программист проверяет работоспособность разработанного им кода, он обычно выполняет тестирование вручную.

Тест содержит проверки условий, которые могут либо выполняться, либо нет. Когда они выполняются, говорят, что тест пройден.

! Прохождение теста подтверждает поведение, предполагаемое программистом.

2

TDD — Test Driven Development

TDD — это методология разработки ПО, которая основывается на повторении коротких циклов разработки:

- изначально пишутся тесты, покрывающие желаемое поведение,
- затем пишется программный код, который реализует желаемое поведение системы и позволит пройти написанные тесты один за другим,
- затем проводится рефакторинг написанного кода с постоянной проверкой прохождения тестов.

В 1999 году при своём появлении разработка через тестирование была тесно связана с концепцией «сначала тест» (англ. test-first), применяемой в экстремальном программировании (XP-методология), однако позже выделилась как независимая методология.

Разработка через тестирование требует от разработчика создания автоматизированных модульных тестов, определяющих требования к коду непосредственно перед написанием самого кода.

3

Библиотеки для unit-тестирования

Разработчики часто пользуются библиотеками для тестирования (англ. testing frameworks) для создания и автоматизации запуска наборов тестов.

На практике **модульные тесты покрывают критические и нетривиальные участки кода**. Это может быть код, который подвержен частым изменениям, код, от работы которого зависит работоспособность большого количества другого кода, или код с большим количеством зависимостей.

TDD не только предполагает проверку корректности, но и влияет на дизайн программы. Опираясь на тесты, разработчики могут быстрее представить, какая функциональность необходима пользователю.

4

Библиотеки для unit-тестирования

См. https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C++

1. **Boost::Test** - Boost — одна из самых известных (и сам больших) библиотек для C++, а Boost::Test – это Фреймворк для тестирования, входящий в неё и построенный на макросах
2. **Google Test** - Google C++ Testing Framework— библиотека для модульного тестирования (англ. unit testing) на языке C++. Исходные тексты открыты с середины 2008 года под лицензией BSD. Документация частично переведена на русский язык.
3. … много других фреймворков

Google Test построена на **методологии тестирования xUnit**, то есть **когда отдельные части программы (классы, функции, модули) проверяются отдельно друг от друга, в изоляции.** Библиотека сама по себе разработана с активным применением тестирования, когда при добавлении каких-либо частей в официальную версию, кроме кода самих изменений необходимо написать набор тестов, подтверждающих их корректность.

5

Выбор фреймворка

- Прост и понятен в использовании
- Хорошо организован (тесты удобно группировать логически, чтобы эта структура и иерархия отображала рабочий код)
- Понятные результаты (результата прохождения тестов были понятны и информативны с 1го взгляда)
- Переносим между платформами
- Быстр и производителен

6

Возможности Google Test

1. Широкий выбор утверждений (assert).
2. Различная параметризация типами и значениями.
3. Объединение тестов в группы (наборы – test case). Полное имя теста формируется из имени группы и собственного имени теста.
4. Тесты могут использовать тестовые классы (англ. test fixture), что позволяет создавать и повторно использовать одну и ту же конфигурацию объектов для нескольких различных тестов.
5. Удобная конфигурация запусков.
6. Безопасен для многопоточного использования, но для использования утверждений в разных потоках одновременно необходимо самостоятельно разработать примитивы синхронизации.
7. Поддержка разных платформ (Linux, Windows и Mac, AIX, HP-UX, Solaris, Tru64, zSeries и др.) Для официально неподдерживаемых платформ разработчик должен самостоятельно скомпилировать Google Test).
8. XML отчеты.
9. В состав библиотеки входит специальный скрипт, который упаковывает её исходные тексты всего в два файла: gtest-all.cc и gtest.h. Эти файлы могут быть включены в состав проекта без каких-либо дополнительных усилий по предварительной сборке библиотеки.

7

Ключевые понятия

Минимальной единицей тестирования является одиночный **тест**. Тесты не требуется отдельно регистрировать для запуска. Каждый объявленный в программе тест автоматически будет запущен.

Ключевым понятием в Google test framework является понятие утверждения (**assert**). **Утверждение** представляет собой **выражение**, результатом выполнения которого может быть **успех (success)**, **некритический отказ (nonfatal failure) и критический отказ (fatal failure)**.

Критический отказ вызывает завершение выполнения теста, в остальных случаях тест продолжается.

В фреймворке ***mecm*** представляет собой **набор утверждений.**

Тесты могут быть сгруппированы в **наборы (test case).**

Если сложно настраиваемая группа объектов должна быть использована в различных тестах, можно использовать **фиксации (тестовые классы) (fixture).**

Объединенные наборы тестов являются **тестовой программой** (test program).

8

Установка

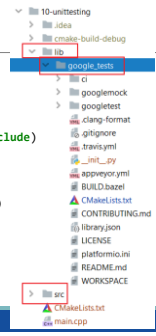
Фреймворк – это набор заготовочных файлов и библиотек, он open sourse, документация и исходный код на GitHub

Чтобы начать работать с библиотекой надо

1. Скачать исходники https://github.com/google/googletest
2. Собрать (build) библиотеку
3. Подключить (link) к своей программе

9

Подключение к проекту



Подключение фреймворка:

```
add_subdirectory(lib/google_tests)
include_directories(lib/google_tests/googletest/include)
```

Слинковать нашу программу с библиотекой gtest:

```
target_link_libraries(DemoGoogleTest gtest)
```

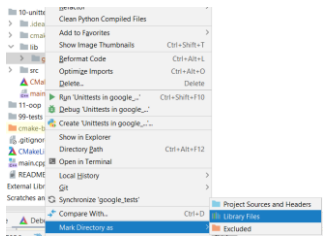
Далее запускаем генерацию Cmake (reload)

10

Пометить папку библиотеки

При индексировании кода, при рефакторинге, IDE не будет рассматривать этот каталог, как каталог вашего исходного кода.

Так мы явно говорим, что ничего не хотим менять.



11

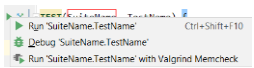
Создание теста

Сгенерировать Test

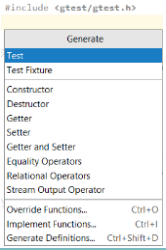
```
TEST(SuiteName, TestName) {
}
```

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

TEST(SuiteName, TestName) {
}
```




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

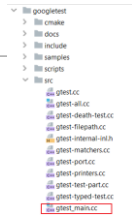


12

Если нет main(), то используем main из фреймворка:

```
target_link_libraries(DemoGoogleTest gtest gtest_main)
```

```
GTEST_API_ int main(int argc, char **argv) {
  printf("Running main() from %s\n", __FILE__);
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```



Макрос RUN_ALL_TESTS() сам соберет все тесты нашей программы.

13

Утверждения (assertion)

Это макросы, которые проверяют выполнение условий, делятся на 2 категории:

1. ASSERT_ – для проверки критичных участков кода. Эти утверждения порождают критические отказы.
2. EXPECT_ - для проверки не критичных участков кода. Эти утверждения порождают не критические отказы.

Следует иметь ввиду, что в случае **критического отказа** выполняется немедленный возврат из функции, в которой встретилось вызвавшее отказ утверждение. Если за этим утверждением идет какой-то очищающий память код или какие-то другие завершающие процедуры, можете получить утечку памяти.

14

Простейшие логические

_TRUE(condition);

_FALSE(condition);

T.e.

ASSERT_TRUE(condition);

ASSERT_FALSE(condition);

EXPECT_TRUE(condition);

EXPECT_FALSE(condition);

15

Сравнение

_EQ(expected, actual); = =

_NE(val1, val2); !=

_LT(val1, val2); <

_LE(val1, val2); <=

_GT(val1, val2); >

_GE(val1, val2); >=

16

Сравнение строк

_STREQ(expected_str, actual_str); // равно

_STRNE(str1, str2); // не равно

_STRCASEEQ(expected_str, actual_str); // равно - регистронезависимо

_STRCASENE(str1, str2); // не равно регистронезависимо

17

Проверка на исключения

_THROW(statement, exception_type); // выбрасывается ли исключение определенного типа

_ANY_THROW(statement);

_NO_THROW(statement);

Death - условие прохождения тестов - программа завершается с каком-то кодом ошибки или «убивается» по сигналу и проверяется там именно ожидаемый код возврата, и в поток вывода ошибок мы пишем сообщение о том, что что-то произошло.

_DEATH

_DEATH_IF_SUPPORTED

_EXIT

18

Проверка предикатов

ЕСЛИ предыдущих проверок не достаточно, то можно использовать свои собственные функции-предикаты (возвращают bool) в качестве проверки условий прохождения тестового сценария.

_PREDN(pred, val1, val2, ..., valN); // N <= 5 – количество принимаемых предикатом параметров

_PRED_FORMATN(pred_format, val1, val2, ..., valN); — работает аналогично предыдущей, но позволяет контролировать вывод (вы переопределяете вывод об ошибках сообщений в тесте)

19

Итоги

1. TDD — Test Driven Development
2. Библиотеки для unit-тестирования
3. Выбор фреймворка
4. Возможности Google Test
5. Подключение к проекту
6. Создание теста
7. Утверждения: логические, сравнения, сравнения строк, проверка на исключения, проверка предикатов

20