Технологии разработки программного обеспечения

Старший преподаватель Кафедры вычислительных систем
Елизавета Ивановна Токмàшева

email: eliz_tokmasheva@sibguti.ru

2022, 1 курс, 2 семестр

Основы тестирования

- Тестирование приложений
  - Основы
  - Уровни зрелости
  - Типы тестирования
  - Уровни тестирования
  - Невозможно тестировать всё
- Тестирование классов эквивалентности
- Тестирование граничных значений
- Ctest
- Code coverage

Основы тестирования

Тестирование программного обеспечения (Software Testing) - проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом.

В более широком смысле, тестирование - это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis)

Lee Copeland, «A Practitioner's Guide to Software Test Design»
IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004

Основные термины

1. **Требования** - совокупность утверждений, относительно атрибутов, свойств или качеств программной системы, подлежащей разработке.

2. **Спецификация** - законченное описание поведения программы, которую требуется разработать

Включает в себя :

3. **Функциональные требования** - требуемые характеристики системы (функциональность)

4. **Не функциональные требования** - требования, которые не влияют на основную функциональность системы

Основные термины

Тестовый случай (test case) - набор условий, при которых тестировщик будет определять удовлетворяется ли заранее определенное требование

Ошибка (bug)- отклонение фактического результата от ожидаемого

Отчет об ошибке (bug report) - документ, описывающий ситуацию, которая привела к обнаружению ошибки, фактический и ожидаемый результат

Модель зрелости тестирования ПО

Модель зрелости тестирования программного обеспечения — это систематизированный подход к развитию процесса тестирования, который предлагает систему элементов эффективных процессов и пути достижения конкретных процессных целей.

Опираясь на модель зрелости, можно решить две основные задачи процессного развития: понять и зафиксировать текущий процесс тестирования и определить направления, требующие улучшения.

Практика показывает, что процессные изменения возможны только на основании четкого понимания руководством необходимости внесения таких изменений — любые структурные и процедурные изменения невозможны без политической воли руководства. Помимо получения поддержки руководства и необходимых ресурсов, внесение изменений в процесс работ по тестированию требует четкого планирования, как и любая другая проектная деятельность.

Уровни зрелости

Пять уровней зрелости (Software Testing Maturity Model):

1. Хаотический

Процесс тестирования программного обеспечения имеет хаотический характер, что отличает большинство начинающих компаний. Процесс тестирования не определен как выделенная активность и не отделен от процесса отладки кода.

Тестирование выполняется по факту создания кода и построения или сборки системы. Цель тестирования — показать, что приложение работает.

2. Фаза разработки

Тестирование программного обеспечения отделено от кодирования и выделяется как следующая фаза. Главная цель тестирования — показать, что приложение соответствует требованиям. Имеются базовые подходы и практики тестирования.

Уровни зрелости

3. Интегрированный

Процесс тестирования интегрирован в жизненный цикл разработки программного обеспечения. Цели тестирования базируются на требованиях. Имеется организация тестирования, а само тестирование выделено в профессиональную деятельность

4. Управление и измерение

Тестирование является измеряемым и контролируемым процессом. Процессы критических осмотров (review) проектных артефактов (тестовые планы и сценарии, сообщения об ошибках, итоговые отчеты о состоянии версии и т.д.) относятся к тестовым активностям. Продукт тестируется на соответствие таким качественным метрикам, как надежность, удобство, сопровождаемость

Уровни зрелости

5. Оптимизация процесса, предотвращение ошибок и контроль качества

Тестирование является определенным и управляемым процессом. Стоимость тестирования наравне с показателями эффективности может быть определена. Тестирование как процесс поддается изменениям, которые однозначно положительно на него влияют. Внедрены и используются практики предотвращения ошибок и контроля качества. Автоматизированное тестирование применяется как основной подход в тестировании. Проектирование тестов, анализ полученных результатов, обработка описаний ошибок, а также метрик, связанных с тестированием, осуществляется при помощи соответствующих инструментальных средств

Жизненный цикл ПО

Жизненный цикл разработки ПО (SDLC – Software development lifecycle) – это серия из шести фаз, через которые проходит любая программная система.

1. Сбор и анализ требований
2. Документирование требований
3. Дизайн
4. Разработка ПО
5. Тестирование
6. Внедрение и поддержка продукта

Жизненный цикл ПО

1. Сбор и анализ требований

Определение объема работ, согласование четкого, краткого документа с требованиями, создание прототипов (макетов) для подтверждения и уточнения окончательных требований
Цель: понимание и анализ требований

2. Документирование требований

Четкое определение и документирование требований к продукту, утверждение со стороны клиента.
Создание документа SRS (Software Requirement Specification), содержащего все требования к продукту, которые должны быть спроектированы и разработаны в течение жизненного цикла проекта.

Жизненный цикл ПО

3. Дизайн

Все предложенные подходы к проектированию архитектуры продукта документируются в спецификации DDS (Design Document Specification) и выбирается наилучший подход к проектированию.

Данный подход очень четко определяет все архитектурные модули продукта, а также его связь с внешними и сторонними модулями.

Жизненный цикл ПО

4. Разработка ПО

Разработка и сборка продукта. Программный код разрабатывается на основе DDS. Написанный код покрывается Unit-тестами, взаимодействие новых функциональностей с другими модулями тестируется с помощью интеграционных тестов. Данный этап полностью выполняется разработчиками.

Жизненный цикл ПО

5. Тестирование

Затрагивает все этапы жизненного цикла.

Дефекты продукта регистрируются, отслеживаются, исправляются и повторно тестируются. Это происходит до тех пор, пока продукт не достигнет стандартов качества, которые прописаны в SRS.

6. Внедрение и поддержка продукта

Релиз продукта. Иногда внедрение происходит поэтапно, в соответствии с бизнес-стратегией.

Типы тестирования

Black box

Стратегия, основанная на требованиях и спецификациях. Не требует знания реализации.

White box

Стратегия, основанная на знании реализации тестируемого продукта. Требует навыков программирования.

Gray box

Стратегия, основанная на знании структуры программного продукта

Уровни тестирования

Модульное тестирование (Unit testing). Unit – «наименьший» фрагмент создаваемого кода. C++ / Java – class, C – функция

Интеграционное тестирование – тестирование подсистемы или целой системы. Выявляет ошибки связей между частями кода.

Системное тестирование

- Функциональное тестирование
- Юзабилити тестирование
- Тестирование безопасности
- Тестирование локализации
- Тестирование производительности
- …

Приемочное тестирование

Невозможно тестировать всё!

```
int bench( int j)
{
     j -= 1;              // Error! Should be j += 1;
     j /= 30000;
     return j;
}
```

int: -32768 <= j <= 32767

Только 4 входных значения из 65536 позволят найти ошибку

Классы эквивалентности

Пример:

0 – 16 не нанимать

16 – 18 частичная занятость

18 – 55 полная занятость

55 – 99 не нанимать

Полное покрытие: тестовые данные от 0 до 100

Покрытие классов эквивалентности: 4 теста

- 10 - не нанимать
- 17 – частичная занятость
- 40 – полный рабочий день
- 80 – не нанимать

Классы эквивалентности

Ожидания:

Если один набор данных из класса эквивалентности обнаруживает ошибку, то и все остальные наборы данных из этого же класса эквивалентности приведут к ошибке

Если один набор данных из класса эквивалентности **НЕ** обнаруживает ошибку, то ни один другой набор данных из этого же класса эквивалентности вероятно не приведет к ошибке.

Классы эквивалентности

Резюме:

Тестирование классов эквивалентности – техника, использующаяся для сокращения количества тестов до управляемого числа с сохранением разумного покрытия функционала тестами

Граничные значения

16 лет – что является правильным ответом ?

Пример:

0 – 15 не нанимать

16 – 17 частичная занятость

18 – 54 полная занятость

55 – 99 не нанимать

Наборы данных:

```
{-1,0,1}
{14,15,16}
{17,18,19}
{54,55,56}
{98,99,100}
```

Библиотека ctest

```
#include <sum.h>
#include <ctest.h>


CTEST(arithmetic_suite, simle_sum)
{
    // Given
    const int a = 1;
    const int b = 2;

    // When
    const int result = sum(a, b);

    // Then
    const int expected = 3;
    ASSERT_EQUAL(expected, result);
}
```


https://github.com/bvdberg/ctest

Библиотека ctest

```
$ make test
make[1]: Nothing to be done for `all'.
TEST 1/27 suite1:test1 [OK]
TEST 2/27 suite1:test2 [FAIL]
  ERR: mytests.c:12  expected 1, got 2
TEST 3/27 suite2:test1 [FAIL]
  ERR: mytests.c:16  expected 'foo', got 'bar'
TEST 4/27 suite3:test3 [OK]
TEST 5/27 memtest:test1 [OK]
  LOG: memtest_setup() data=0x107f488a0 buffer=0x0
  LOG: __ctest_memtest_test1_run()  data=0x107f488a0  buffer=0x7fed79800000
  LOG: memtest_teardown() data=0x107f488a0 buffer=0x7fed79800000
TEST 6/27 memtest:test3 [SKIPPED]
TEST 7/27 memtest:test2 [FAIL]
  LOG: memtest_setup() data=0x107f488a0 buffer=0x7fed79800000
  LOG: __ctest_memtest_test2_run()  data=0x107f488a0  buffer=0x7fed79800000
  ERR: mytests.c:53  shouldn't come here
TEST 8/27 fail:test1 [FAIL]
  ERR: mytests.c:61  shouldn't come here
TEST 9/27 weaklinkage:test1 [OK]
  LOG: __ctest_weaklinkage_test1_run()
TEST 10/27 weaklinkage:test2 [OK]
  LOG: __ctest_weaklinkage_test2_run()
TEST 11/27 nosetup:test1 [OK]
  LOG: __ctest_nosetup_test1_run()
  LOG: nosetup_teardown()
TEST 12/27 ctest:test_assert_str [FAIL]
  ERR: mytests.c:98  expected 'foo', got 'bar'
TEST 13/27 ctest:test_assert_equal [FAIL]
  ERR: mytests.c:103  expected 123, got 456
TEST 14/27 ctest:test_assert_not_equal [FAIL]
  ERR: mytests.c:108  should not be 123
TEST 15/27 ctest:test_assert_null [FAIL]
  ERR: mytests.c:114  should be NULL
TEST 16/27 ctest:test_assert_not_null_const [OK]
TEST 17/27 ctest:test_assert_not_null [FAIL]
```

https://github.com/bvdberg/ctest

Покрытие кода. Code coverage

Покрытие кода — мера, используемая при тестировании программного обеспечения. Она показывает процент, насколько исходный код программы был протестирован (отражает отношение строчек, задействованных в тестах, ко всем строчкам исходного кода).

Основные способы измерения покрытия кода:

Покрытие операторов — каждая ли строка исходного кода была выполнена и протестирована

Покрытие условий — каждая ли точка решения (вычисления истинно ли или ложно выражение) была выполнена и протестирована

Покрытие путей — все ли возможные пути через заданную часть кода были выполнены и протестированы

Покрытие кода. Code coverage

Основные способы измерения покрытия кода (продолжение):

Покрытие функций — каждая ли функция программы была выполнена

Покрытие вход/выход — все ли вызовы функций и возвраты из них были выполнены

Покрытие значений параметров — все ли типовые и граничные значения параметров были проверены