UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Testes e geração de casos de teste

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Introdução

- Modificar um programa é difícil
  - Mais do que implementá-lo inicialmente
  - Modificações em cadeia no código
  - Correções introduzem (novos) erros
- Como diminuir a chance de erros futuros?
  - Testar o código durante o desenvolvimento
  - O que é um erro no programa? E um teste?

Introdução

- O que é teste de software?
  - Atividade responsável por avaliar as capacidades de um programa, verificando que esse alcança os resultados previamente estabelecidos

> "Testing is the process of executing a program with the intent of finding errors."
>
> – Glenford J. Myers, The Art of Software Testing, p. 6

Introdução
Motivação

- Diminuir o número de erros ao cliente
  - Melhorar a qualidade do software
- Detectar problemas mais rapidamente
  - Minimizar o custo de correção
- Modelagem mais precisa
  - Pensar em possíveis testes (cenários) para o sistema ajudam a entender melhor o problema

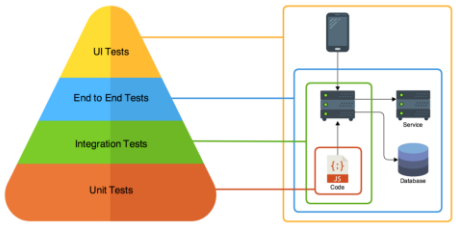
Introdução
Princípios

- Teste ≠ Debugging
  - Caso o teste encontre um erro, o processo de depuração pode ser usado para corrigi-lo
- Programa → paciente doente
  - Teste de sucesso → Problemas detectados
  - Teste sem sucesso → Nenhum problema
- Ponto de vista psicológico
  - Análise e Codificação são tarefas construtivas
  - Teste é uma tarefa "destrutiva"

Introdução
Tipos de testes

- Testes de unidade
  - Programação/codificação (módulo específico)
- Testes de integração
  - Projeto (diferentes módulos)
- Testes de validação
  - Requisitos
- Testes de sistema
  - Demais elementos

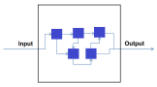
Introdução
Tipos de testes



Introdução
Tipos de testes

- Black-box
  - Pouco, ou nenhum, acesso ao código

- White-box
  - Conhecemos o código e o seu funcionamento



Testes de unidade

- White-box (no nosso caso)
  - Código feito para testar as classes
- Avalia a funcionalidade pelo contrato
  - Teste é como se fosse um código cliente
  - Pouco (de preferência zero) de lógica
- Entretanto, conhecendo a implementação é possível testar comportamentos específicos

Testes de unidade

- Trecho de código que chama outro trecho de código para verificar o comportamento apropriado de uma determinada hipótese
- Hipótese não validada (resultado incorreto), dizemos que o teste de unidade "falhou"
  - Objetivo é que todos os testes passem!
  - Resultados de acordo com o esperado

Testes de unidade
O que é uma unidade?

- Menor unidade de código testável
  - Método / bloco de código em um método
- Teste verifica uma hipótese para o método
  - As partes que utilizam o método devem ser testadas em outros casos de testes separados
- Diferentes aspectos podem ser testados
  - E/S, condições de contorno, exceções, ...

Testes de unidade
Casos de teste

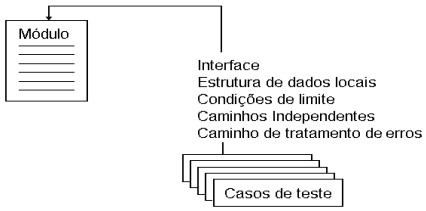
- Condição particular a ser testada
  - Valores de entrada
  - Restrições de execução
  - Resultado ou comportamento esperado

IEEE Standard 610 (1990) defines test case as follows:
1. A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
2. (IEEE Std 829-1983) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.

Testes de unidade
Casos de teste

- Refletem os requisitos que serão verificados
- Casos básicos
  - Positivo
    - Demonstrar que o requisito é atendido
  - Negativo
    - Requisito só é atendido sob certas condições
    - O que acontece em cenários com condições ou dados inaceitáveis, anormais ou inesperados?

Testes de unidade
Casos de teste



Testes de unidade
Casos de teste – Exemplo

- Programa para identificar triângulos
  - Entrada: 3 números inteiros (lados)
  - Saída: Equilátero, Isósceles, Escaleno
- Casos positivos
  - Quantos casos de teste para equilátero?
    - [5, 5, 5]
  - Quantos casos de teste para isósceles?
    - [3, 3, 4]; [3, 4, 3]; [4, 3, 3]
  - Quantos casos de teste para escaleno?
    - [3, 4, 6]; [3, 6, 4]; [4, 3, 6]

Testes de unidade
Casos de teste – Exemplo

- Casos negativos
  - Teste quando um dos lados é zero
  - Teste quando um dos lados é negativo
  - Teste verificando valores para triângulos válidos
    - Verificar diferentes permutações
    - [1, 2, 3]; [1, 3, 2]; [2, 1, 3]; [2, 3, 1]; [3, 1, 2]; [3, 2, 1]

Testes de unidade
Características

- Reproduzível
  - Mesmos resultados sempre que é executado
- Independente
  - Testam apenas uma funcionalidade por vez
  - Não dependem de outros testes (ordem)
- Completo
  - Maior cobertura possível do código

Testes de unidade
Vantagens

- Permitem a utilização de ferramentas que validam o código por condições fail/pass
- Podem ser feitos pelo implementador
- Ajudam a entender e manter o código
- Falhas detectadas durante as alterações
  - Se o código muda, o teste começa a falhar

Framework

- Automatização dos testes de unidade
  - Agilizar a verificação após mudanças
  - Evitar um trabalho tedioso (caro) → Falhas
- Doctest: https://github.com/onqtam/doctest
  - Light, fast, single-header, free, feature-rich, ...
- Outras opções:
  - Catch2: https://github.com/catchorg/Catch2
  - GoogleTest: https://github.com/google/googletest

Framework

- Funcionamento baseado em asserções
- Diferentes níveis de severidade
  - REQUIRE / CHECK / WARNING
- Métodos auxiliares
  - Condições
    - `CHECK(thisReturnsTrue());`
  - Exceções
    - `CHECK_THROWS_AS(func(), std::exception);`

https://github.com/onqtam/doctest/blob/master/doc/markdown/assertions.md

Framework

- Criar um arquivo de teste (!)
  - Geralmente um para cada classe
- Criar um método para o teste (test case)
  - Criar o cenário do teste
  - Executar a operação sendo testada
  - Conferir o resultado retornado

https://github.com/onqtam/doctest/blob/master/doc/markdown/tutorial.md

Exemplo

```
#ifndef FACTORIAL_H
#define FACTORIAL_H

int factorial(int number);

#endif
```

```
#include "Factorial.hpp"

int factorial(int number) {
  if (number <= 1) {
    return number;
  } else {
    return factorial(number - 1) * number;
  }
}
```

```
g++ -c Factorial.cpp
```


Exemplo

Essa flag define o main que executará os testes. Deve aparecer em um único arquivo em todo código.

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"

#include "Factorial.hpp"

TEST_CASE("Teste Factorial - Casos Base") {
  CHECK(factorial(1) == 1);
  CHECK(factorial(2) == 2);
}

TEST_CASE("Teste Factorial - Casos Gerais") {
  CHECK(factorial(3) == 6);
  CHECK(factorial(5) == 120);
  CHECK(factorial(10) == 3628800);
}
```

```
g++ -o main_test TesteFactorial.cpp Factorial.o
```


Exemplo

```
[doctest] doctest version is "2.3.2"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases:      2 |      2 passed |      0 failed |      0 skipped
[doctest] assertions:      5 |      5 passed |      0 failed |
[doctest] Status: SUCCESS!
```

Exemplo

- E os possíveis casos excepcionais?

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"

#include "Factorial.hpp"

TEST_CASE("Teste Factorial - Casos Base") {
  CHECK(factorial(0) == 1);
  CHECK(factorial(1) == 1);
  CHECK(factorial(2) == 2);
}

TEST_CASE("Teste Factorial - Casos Gerais") {
  CHECK(factorial(3) == 6);
  CHECK(factorial(5) == 120);
  CHECK(factorial(10) == 3628800);
}

TEST_CASE("Teste Factorial - Casos excepcionais") {
  CHECK_THROWS_AS(factorial(-2), ExcecaoEntradaNegativa);
}
```

Corrigir o código!

Criar a exceção!

https://wandbox.org/permlink/1iceX9kDvB9TPt6h

Exemplo

```
[doctest] doctest version is "2.3.2"
[doctest] run with "--help" for options
===============================================================================
TesteFactorial.cpp:6:
TEST CASE:  Teste Factorial - Casos Base

TesteFactorial.cpp:7: ERROR: CHECK( factorial(0) == 1 ) is NOT correct!
  values: CHECK( 0 == 1 )

===============================================================================
TesteFactorial.cpp:18:
TEST CASE:  Teste Factorial - Casos excepcionais

TesteFactorial.cpp:19: ERROR: CHECK_THROWS_AS( factorial(-2), ExcecaoEntradaNegativa ) did NOT throw at all!

===============================================================================
[doctest] test cases:      3 |      1 passed |      2 failed |      0 skipped
[doctest] assertions:      7 |      5 passed |      2 failed |
[doctest] Status: FAILURE!
```


Cobertura de código

- Medida do grau que o código do programa é executado dado um conjunto de testes
- Quanto maior a cobertura, menor a chance do código conter erros não detectados

> If you make a certain level of coverage a target, people will try to attain it. The trouble is that high coverage numbers are too easy to reach with low quality testing.
>
> – Martin Fowler

https://en.wikipedia.org/wiki/Code_coverage
https://martinfowler.com/bliki/TestCoverage.html

Cobertura de código
Critérios básicos (white-box)

- Declaração
  - Testes que avaliam todas as linhas do código
  - Testes simples, porém pobres
- Decisões (branches)
  - Avaliar diferentes caminhos condicionais
- Condições
  - Parada, valores inválidos, valores limite, ...

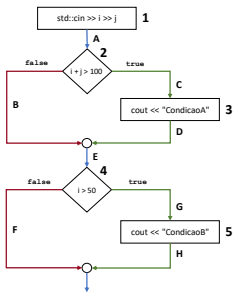
Exemplo

- Quantos testes são necessários para cobrir todas as declarações (statement) e decisões (branches)?

```
int main() {
  int i, j;
  std::cin >> i >> j;
  if (i + j > 100) {
    std::cout << "CondicaoA" << std::endl;
  }
  if (i > 50) {
    std::cout << "CondicaoB" << std::endl;
  }
  return 0;
}
```


Exemplo

- Statement Coverage
  - Encontrar o menor número de caminhos que cobre todos os nós (números).
  - 1A-2C-3D-E-4G-5H
- Branch Coverage
  - Encontrar o menor número de caminhos que cobre todas as arestas (letras).
  - 1A-2C-3D-E-4G-5H
  - 1A-2B-E-4F



https://www.istqb.guru/how-to-calculate-statement-branchdecision-and-path-coverage-for-istqb-exam-purpose/

Cobertura de código
Ferramentas

- gcov
  - Analisa número de vezes que cada linha de um programa é executada durante uma execução
  - Permite encontrar áreas do código que não são utilizadas ou que não são avaliadas nos testes
- LCOV
  - Formata os relatórios em arquivos html
  - Facilita identificar e verificar problemas

https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
http://ltp.sourceforge.net/coverage/lcov.php

Cobertura de código
Ferramentas – Passo-a-passo

1. Compilar todos os arquivos com o parâmetro "--coverage" (saída arquivos '.gcno').

```
$ g++ -c --coverage Factorial.cpp
$ g++ --coverage -o TesteFactorial TesteFactorial.cpp Factorial.o
```

2. Execute o arquivo executável (saída arquivos '.gcda').

```
$ ./TesteFactorial
```

3. Gerar os relatórios de cobertura (saída arquivos '.gcov').

```
$ mkdir coverage
$ mv *.gcno *.gcda coverage/
$ gcov -lpr *.cpp -o coverage/
$ mv *.gcov coverage/
```

4. Gerar o relatório em html*.

```
$ lcov --no-external --capture  --directory . --output-file coverage/coverage.info
$ genhtml coverage/coverage.info --output-directory coverage
```

*Parece que o LCOV não é compatível com GCC 8.0.

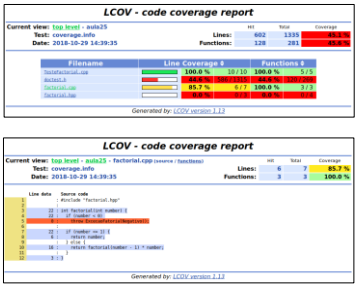
Cobertura de código
Ferramentas – gcov

```
File 'Factorial.cpp'
Lines executed:85.71% of 7
Creating 'Factorial.cpp.gcov'

File 'Factorial.hpp'
Lines executed:0.00% of 3
Creating 'Factorial.hpp.gcov'

File 'TesteFactorial.cpp'
Lines executed:100.00% of 10
Creating 'TesteFactorial.cpp.gcov'

File 'doctest.h'
Lines executed:44.56% of 1315
Creating 'doctest.h.gcov'

Lines executed:45.09% of 1335
```

```
-:    0:Source:Factorial.cpp
-:    0:Programs:2
-:    1:#include "Factorial.hpp"
-:    2:
22:    3:int factorial(int number) {
22:    4:  if (number < 0)
#####:    5:    throw ExcecaoEntradaNegativa();
-:    6:
22:    7:  if (number <= 1) {
 6:    8:    return number;
-:    9:  } else {
16:   10:    return factorial(number - 1) * number;
-:   11:  }
 3:   12:}
```

Factorial.cpp.gcov

Cobertura de código
Ferramentas – LCOV



Modularização

- Múltiplos "mains"
  - Um para os testes
  - Um para o código principal
- g++ não deixa compilar, como resolver?
  - Separar código principal, testes, coberturas, ...
  - Fazer um makefile que cuida de cada caso
    - Criar múltiplos targets

Modularização

```
. project
├── Makefile
├── build
│   └── [objects]
├── coverage
├── include
│   └── modulo1
│       ├── modlc1.hpp
│       └── modlc2.hpp
└── src
    ├── main.cpp
    ├── main_test.cpp
    └── modulo1
        ├── modlc1.cpp
        └── modlc2.cpp
└── tests
    └── modulo1
        └── testmodlc1.cpp
├── third_party
│   └── doctest.h
```

Considerações finais

- Bons testes cobrem a maioria dos (ou todos os) fluxos possíveis de execução
  - Nenhuma relação com a qualidade dos testes
  - Podemos ter 100% de cobertura e o código ainda ter erros de lógica (exemplo Fatorial)
- Um teste por método, no mínimo!

Considerações finais

- Casos base
  - Onde seu código com certeza funciona
- *Corner cases*
  - Entradas especiais (pouco esperadas, válidas)
  - Ordenar um vetor com 1 elemento
- Valores inválidos (seja defensivo)