UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Introdução ao teste de software

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

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Introdução

- Modificar um programa é difícil
  - Algumas vezes mais do que implementá-lo inicialmente
  - Podem gerar modificações em cadeia no código
  - Alterações/correções podem resultar em (novos) erros
- Como diminuir a chance de problemas 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 se esse alcança os resultados esperados
  - O que fazer para tentar **quebrar** o programa?

> "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

- Detectar problemas mais rapidamente
  - Minimizar o custo (impacto) de detecção/correção
- Diminuir o número de erros ao usuário final
  - Melhorar a qualidade percebida do software

> "Program testing can be used to show the presence of bugs, but never to show their absence!"
>
> – Edsger W. Dijkstra

Introdução
Motivação

- Modelagem mais precisa
  - Pensar em possíveis casos de testes (cenários de avaliação) para o sistema ajudam a entender melhor o problema (entradas, saídas)
- Testar ≠ Depurar
  - Diferentes papeis: Vândalo → Detetive
  - Caso o teste encontre um erro (quebrou o programa), o processo de depuração deve ser usado para consertá-lo

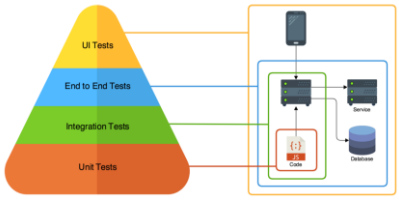
Teste de software
Princípios

- Programa → paciente doente
  - Teste de sucesso → Problemas detectados
  - Teste sem sucesso → Nenhum problema (será?)

- Ponto de vista psicológico
  - Análise e Codificação são tarefas construtivas
  - Teste seria uma tarefa "destrutiva" (mas não é!)

Teste de software
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



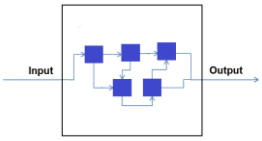
Fonte: https://medium.com/@nathankpeck/microservice-testing-introduction-347d2f74095e

Fonte: https://martinfowler.com/bliki/TestPyramid.html

Teste de software
Métodos de testes

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

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



https://www.careerist.com/insights/a-guide-to-white-box-black-box-and-gray-box-testing

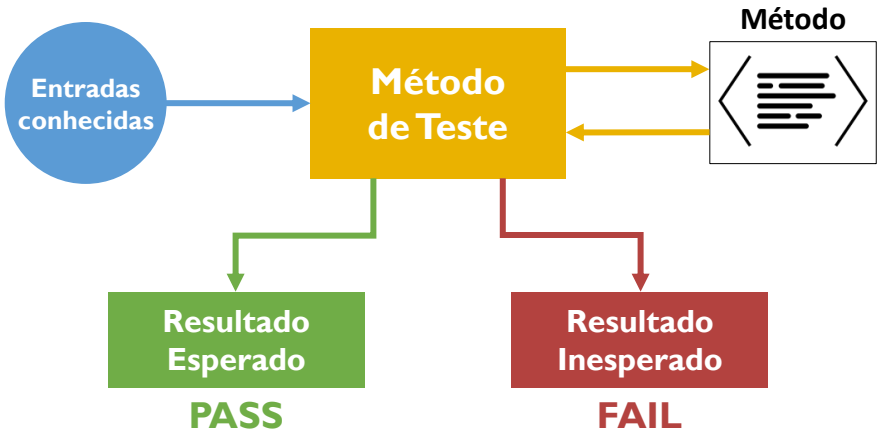
Testes de unidade

- O teste é um trecho de código feito pelo desenvolvedor e que chama outro trecho para verificar o comportamento apropriado de uma determinada hipótese (verdadeira)
- Quando a hipótese não é validada (resultado incorreto), dizemos que o teste de unidade "falhou" (achou um erro)
  - Objetivo é que todos os testes passem!
  - Resultados de acordo com o esperado

Testes de unidade
O que é uma unidade?

- Menor bloco de código testável
  - Método / trecho de código em um método
- Teste verifica uma hipótese para um método
  - As demais partes que utilizam o método devem ser testadas em outros casos de testes separados e com suas próprias hipóteses
- Diferentes aspectos podem ser testados
  - E/S (resultado esperado), exceções, ...

Testes de unidade



Testes de unidade

- White-box (no nosso caso)
  - Código feito para testar as classes/métodos
  - Uma vez que conhecemos a implementação é possível testar comportamentos (trechos) específicos, mas cuidado com isso!
- Focar em avaliar a funcionalidade pelo contrato
  - Teste é como se fosse um código cliente
  - Pouco (de preferência zero) de lógica

Testes de unidade
Casos de teste

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.

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

Adaptado de https://livrozilla.com/doc/557974/casos-de-teste

Testes de unidade
Casos de teste

- Casos básicos
  - **Positivos**
    - "Caminho feliz" → utilizar dados e condições esperados
    - Demonstrar que o requisito é atendido nesses casos
  - **Negativos**
    - Condição ou dados inaceitáveis, anormais ou inesperados
    - Requisito só deveria ser atendido sob a condição esperada

Testes de unidade
Casos de teste

- Casos base
  - Onde seu código com certeza funciona
- *Edge (corner) cases* → *Boundary conditions*
  - Entradas especiais (pouco esperadas, porém válidas)
  - Ordenar um vetor com 1 elemento
- Valores inválidos (não seja defensivo no teste!)
  - Lançamento de possíveis exceções

Testes de unidade
Casos de teste – Exemplo 1

- Como testar a função abaixo?

```
int max(int a, int b)
```

- Possíveis casos
  - a < b `(1, 2)` → **2**
  - a = b `(7, 7)` → **7**
  - a > b `(4, 3)` → **4**

Testes de unidade
Casos de teste – Exemplo 1

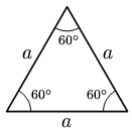
- Relacionamento a e b:
  - a < b
  - a = b
  - a > b

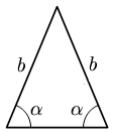
- Valores para a:
  - a = 0
  - a > 0
  - a < 0
  - a = maior valor inteiro
  - a = menor valor inteiro
- Valores para b:
  - b = 0
  - b > 0
  - b < 0
  - b = maior valor inteiro
  - b = menor valor inteiro

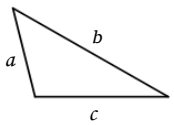
https://sttp.site/chapters/testing-techniques/boundary-testing.html

Testes de unidade
Casos de teste – Exemplo 2

- Método 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 2

- Casos negativos
  - Teste quando um dos lados é zero (ou todos)
  - Teste quando um dos lados é negativo (ou todos)
  - 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
- Isolado/Independente
  - Testam apenas uma funcionalidade por vez
  - Não dependem de outros testes (ordem)
- Completo
  - Maior cobertura possível do código (o que é isso?)

Fast
Isolated
Repeatable
Self-validating
Timely

https://github.com/tekguard/Principles-of-Unit-Testing
https://medium.com/@qaschool/conhe%C3%A7a-os-princ%C3%ADpios-f-i-r-s-t-aplicados-em-testes-de-unidade-999c9d2a00d

Cobertura de código

- Métrica **quantitativa** que avalia o 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 (partes não verificadas)

> "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)

- Cobertura de **declarações** (statements)
  - Testes que avaliam todas as linhas do código
  - Testes simples, porém pobres
- Cobertura de **decisões** (branches)
  - Avaliar diferentes caminhos condicionais
- Cobertura de condições
  - Parada, valores inválidos, valores limite, ...

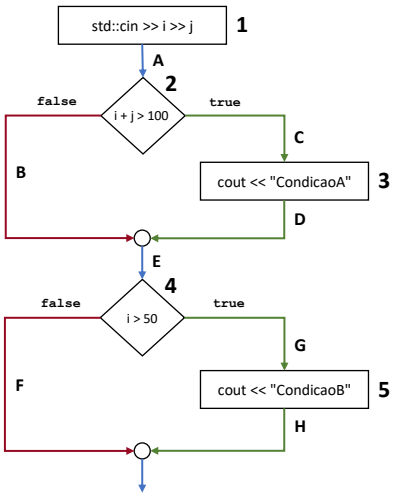
Cobertura de código
Exemplo

- Quantos casos de testes são necessários para cobrir
  - Todas as declarações (statements)?
  - Todas as 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;
}
```


Cobertura de código
Exemplo

- Cobertura de **declarações**
  - Encontrar o menor número de caminhos que cobre <u>todos</u> os <u>vértices</u> (números).
  - 1A-2C-3D-E-4G-5H
- Cobertura de **decisões**
  - Encontrar o menor número de caminhos que cobre <u>todas</u> as <u>arestas</u> (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/

Testes de unidade

- Quando devemos iniciar a fase de testes?
  - Apenas depois de terminada a codificação?
  - Não é uma boa ideia! Por que?
- Test Driven Development (TDD)
  - Foco deve ser no requisito, não no código!
  - Interface → Comportamento
- Prática extremamente recomendada
  - Código se adapta ao teste, não o contrário!

Testes de unidade

Test Driven Development (TDD)

- Escrever o teste antes do código
  - Depois fazer o código que faz o esperado
  - Quando o teste passar, melhorar o código
- Abordagem incremental
  - Pequenos passos (testes) ajudam a alcançar um resultado final de qualidade (projeto)
- Maneira de se desenvolver, não de testar!

Fonte: https://www.codica.com/blog/test-driven-development-benefits/

Testes de unidade
Framework

- Automatização dos testes de unidade
  - Agilizar a verificação após mudanças
  - Evitar um trabalho longo e tedioso (imperfeito)
- 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

Testes de unidade
Framework

- Funcionamento baseado em **asserções**
- Diferentes níveis de severidade
  - REQUIRE / CHECK / WARNING
- Métodos auxiliares
  - Condições
    - `CHECK(thisReturnsTrue());`
    - `CHECK(getNumStudents() == 30);`
  - Exceções
    - `CHECK_THROWS_AS(func(), std::exception);`

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

Testes de unidade
Framework

- Criar um arquivo de teste (!)
  - Geralmente um para cada classe
- Criar os métodos de teste (test cases)
  - 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

Testes de unidade
Framework – Exemplo

Factorial.hpp

```
#ifndef FACTORIAL_H
#define FACTORIAL_H

int factorial(int n);

#endif
```

Factorial.cpp

```
#include "Factorial.hpp"

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

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

Testes de unidade
Framework – Exemplo

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

TesteFactorial.cpp

```
#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);
}
```

Adicionando a biblioteca necessária.

Código que vamos testar.

Definindo um **Caso de Teste** com diferentes verificações.

Hipóteses sendo avaliadas.

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


Testes de unidade
Framework – 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!
```

Testes de unidade
Framework – 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!

Wandbox

Testes de unidade
Framework – 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
Ferramentas

- Encontrar áreas do código que não são executadas
  - Facilita a identificação e resolução de problemas
- gcovr
  - Analisa o número de vezes que linhas de código e decisões são percorridas durante uma execução e gera um relatório
  - Pode gerar relatórios em arquivos html
- Outras ferramentas: gcov/LCOV

https://gcovr.com/en/stable/index.html
https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
http://ltp.sourceforge.net/coverage/lcov.php

Cobertura de código
Ferramentas – gcovr

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 o relatório em html*.

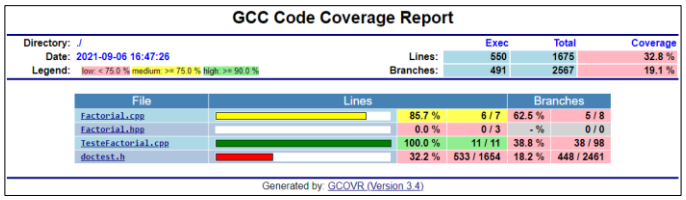
```
$ gcovr -r . --html --html-details -o relatorio.html
```

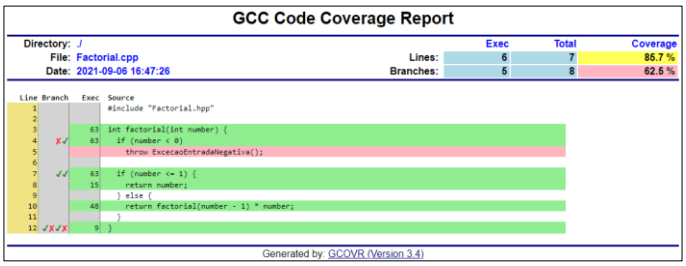
*Veja mais opções e detalhes no próprio site do gcovr.
https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html#Instrumentation-Options

Cobertura de código
Ferramentas – gcovr

```
------------------------------------------------------------------------------
                           GCC Code Coverage Report
Directory: .
------------------------------------------------------------------------------
File                                       Lines    Exec  Cover   Missing
------------------------------------------------------------------------------
Factorial.cpp                                  7       6    85%   5
Factorial.hpp                                  3       0     0%   6,8-9
TesteFactorial.cpp                            11      11   100%
doctest.h                                   1654     533    32%   491,666, ...
------------------------------------------------------------------------------
TOTAL                                       1675     550    32%
------------------------------------------------------------------------------
```


Cobertura de código
Ferramentas – gcovr





Modularização

- Como fazer para rodar todos os testes?
- Múltiplos "mains"
  - Um para executar os testes
  - Um para executar o código principal
- g++ não deixa compilar, como resolver?
  - Separar código principal, testes, cobertura, ...
  - Adaptar o makefile para cuidar de cada caso
    - Criar múltiplos targets

Modularização

```
. project
├── Makefile
├── build
│   └── [objects]
├── coverage
├── include
│   └── modulo1
│       ├── mod1c1.hpp
│       └── mod1c2.hpp
└── src
    │   ├── main.cpp
    │   └── main_test.cpp
    ├── modulo1
    │   ├── mod1c1.cpp
    │   └── mod1c2.cpp
    └── tests
    │   └── modulo1
    │       └── testmod1c1.cpp
    ├── third_party
    │   └── doctest.h
```

Considerações finais

- Testes de Unidade
  - Trecho de código que valida uma hipótese
  - Isolar pequenas partes e verificar corretude individualmente
- Permitem a utilização de ferramentas de automatização que validam o código por checagem de hipóteses (fail/pass)
- Podem ser feitos pelo próprio desenvolvedor
  - Ajudam a entender e manter o código
- Detecção rápida de falhas resultante de alterações
  - Se o código muda erroneamente, os testes deveriam falhar

Considerações finais

- Bons testes cobrem a maioria dos (ou todos) fluxos
  - Um teste por método, no mínimo!
  - Nenhuma relação com a qualidade dos testes
  - Podemos ter 100% de cobertura e ainda ter erros de lógica
- Cobertura de testes (≠ cobertura de código)
  - Métrica **qualitativa** que visa medir a eficácia dos testes perante os requisitos testados, avaliando se os casos de testes são "bons"