

Programação e Desenvolvimento de Software 2

Testes, geração de casos de teste e teste de unidade

Prof. Julio Cesar S. Reis
julio.reis@dcc.ufmg.br

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 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 o alcance de resultados previamente estabelecidos

“Testing is the process of executing a program with the intent of finding errors.”

- Glenford. F. 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 e de forma antecipada
 - Minimizar o custo de correção
- Modelagem mais precisa
 - Pensar em possíveis testes (cenários) para o sistema ajudam a entender melhor o problema

Introdução

Motivação

Table 7-5. Hours to fix bug based on introduction point

		STAGE FOUND			
Stage Introduced	Requirements	Coding/Unit Testing	Integration	Beta Testing	Post-product Release
Requirements	1.2	8.8	14.8	15.0	18.7
Coding/Unit testing	NA	3.2	9.7	12.2	14.8
Integration	NA	NA	6.7	12.0	17.3

NA = Not applicable because cannot find a bug before it is introduced

<https://blog.fullstory.com/what-we-learned-from-google-code-reviews-arent-just-for-catching-bugs/>

Introdução

Princípios

- **Teste \neq Debugging**
 - Caso o teste encontre um erro, o processo de depuração pode ser usado para corrigi-lo
- **Programa \rightarrow Paciente Doente**
 - Teste de sucesso \rightarrow Problemas detectados
 - Teste sem sucesso \rightarrow Nenhum problema
- **Ponto de vista psicológico**
 - Análise e Codificação são tarefas construtivas
 - Teste é uma tarefa “destrutiva”

Tipos de Testes

- Testes de unidade
 - Programação/codificação (módulo específico)
 - Nível de classe
- Testes de integração
 - Projeto (diferentes módulos)
- Testes de validação
 - Requisitos
- Testes de sistemas
 - Demais Elementos

Tipos de Testes

- Outros tipos de testes
 - Instalação
 - Segurança
 - Regressão
 - Performance
 - Usabilidade
 - [...]
 - Black-box vs White-box

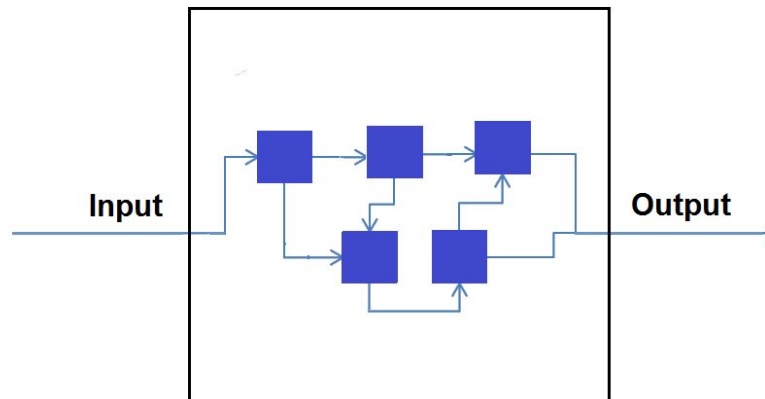
Black-box v. White-box

Black-box. Pouco, ou zero, acesso ao código.
Por exemplo, podemos testar um sistema já pronto com usuários.



White-box testing

- Conhecemos o código e como o mesmo funciona. Podemos encaixar e manipular vários módulos do mesmo



Testes de unidade

- Testes de Unidade (White-box)
 - Nosso foco
 - Código feito para testar as classes
- Parece cíclico
 - Não é. Sabendo do contrato sabemos como usar os objetos de módulos
 - O teste é um código cliente
 - Pouca ou quase zero lógica

Teste 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
 - O objetivo é que todos os testes passem!
 - Resultados de acordo com o esperado

Testes de Unidade

O que é uma unidade?

- Menor unidade de classe testável
 - Método/bloco de código de 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 especiais ou dados inaceitáveis, anormais ou inesperados?

Testes de Unidade

Exemplo I

- 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

Exemplo I

- 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

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

- A automatização dos testes de unidade
 - Agilizar a verificação após mudanças
 - Evitar um trabalho tedioso (caro) → Falhas
- Doctest: <https://github.com/onqtam/doctest>
 - Estamos usando nas VPLs
 - 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);`

Framework - Macros doctest

- Existe uma série de macros no doctest
- A maioria é descrita aqui
- <https://github.com/onqtam/doctest/blob/master/doc/markdown/assertions.md>

Framework

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

Exemplo Simples

- Vamos fazer um código que gera um fatorial. Estilo o primeiro VPL

```
#ifndef PDS2_FAT_H  
#define PDS2_FAT_H  
  
int fatorial(int);  
  
#endif
```

Exemplo Simples

- O código está incompleto
- Apenas para compilar

```
#include "fatorial.h"

int fatorial(int n) {
    return n;
}
```


○ Teste

- Abaixo temos um teste simples
- Vamos entender o mesmo

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o fatorial") {
    CHECK(fatorial(1) == 1);
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(10) == 3628800);
}
```

Entendendo o nosso código

- Um teste de unidade é representado por um código C++
 - Temos algumas macros definidas pela biblioteca doctest (outras funcionam de forma similar)
- Por isso iniciamos com o include

Entendendo o nosso código

```
#include "doctest.h"  
#include "fatorial.h"
```

← Include doctest e código além do código que será testado

```
TEST_CASE("Testando o fatorial") {  
    CHECK(fatorial(2) == 2);  
    CHECK(fatorial(3) == 6);  
    CHECK(fatorial(4) == 24);  
    CHECK(fatorial(10) == 3628800);  
}
```

Test Cases

- Cada Test Case foca em uma funcionalidade. Geralmente método

```
#include "doctest.h"  
#include "fatorial.h"
```

```
TEST_CASE("Testando o fatorial") {  
    CHECK(fatorial(2) == 2);  
    CHECK(fatorial(3) == 6);  
    CHECK(fatorial(4) == 24);  
    CHECK(fatorial(10) == 3628800);  
}
```



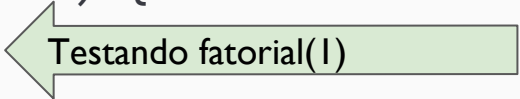
Caso de teste

Test Cases

- A macro CHECK verifica se o resultado é igual ao esperado

```
#include "doctest.h"  
#include "fatorial.h"
```

```
TEST_CASE("Testando o fatorial") {  
    CHECK(fatorial(2) == 2);  
    CHECK(fatorial(3) == 6);  
    CHECK(fatorial(4) == 24);  
    CHECK(fatorial(10) == 3628800);  
}
```



Executando

- Vamos usar um main a moda antiga
- Compilar tudo e rodar

```
$ g++ -std=c++14 *.cpp -o main  
$ ./main
```

■ Parece que deu erro

```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options
=====
testes.cpp:4:
TEST CASE:  Testando o fatorial

testes.cpp:6: ERROR: CHECK( fatorial(3) == 6 ) is NOT correct!
  values: CHECK( 3 == 6 )

testes.cpp:7: ERROR: CHECK( fatorial(4) == 24 ) is NOT correct!
  values: CHECK( 4 == 24 )

testes.cpp:8: ERROR: CHECK( fatorial(10) == 3628800 ) is NOT correct!
  values: CHECK( 10 == 3628800 )

=====
[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:     4 |      1 passed |      3 failed |
[doctest] Status: FAILURE!
```

■ Alguns testes passam

```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options

=====

testes.cpp:4:
TEST CASE: Testando o fatorial


testes.cpp:6: ERROR: CHECK( fatorial(3) == 6 ) is NOT correct!
  values: CHECK( 3 == 6 )

testes.cpp:7: ERROR: CHECK( fatorial(4) == 24 ) is NOT correct!
  values: CHECK( 4 == 24 )

testes.cpp:8: ERROR: CHECK( fatorial(10) == 3628800 ) is NOT correct!
  values: CHECK( 10 == 3628800 )

=====

[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:     4 |      1 passed |      3 failed |
[doctest] Status: FAILURE!
```



■ Outros testes falham.

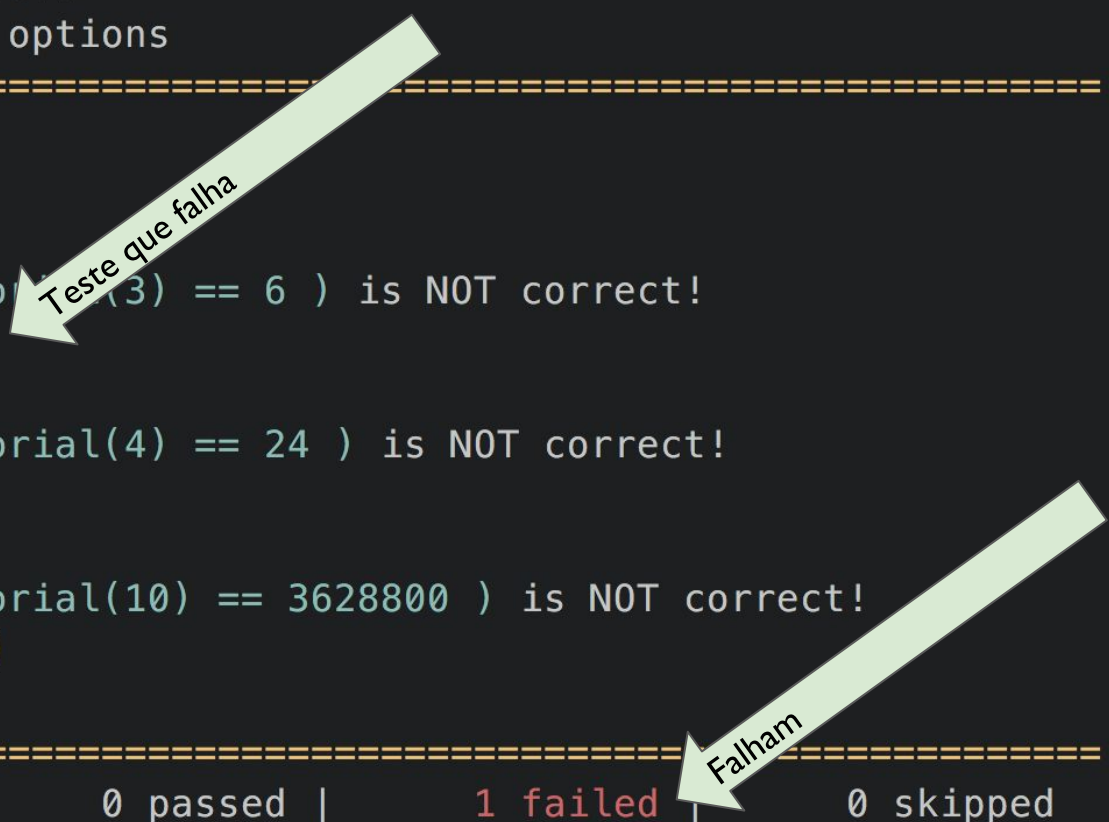
```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options
=====
testes.cpp:4:
TEST CASE: Testando o fatorial

testes.cpp:6: ERROR: CHECK( fatorial(3) == 6 ) is NOT correct!
values: CHECK( 3 == 6 )

testes.cpp:7: ERROR: CHECK( fatorial(4) == 24 ) is NOT correct!
values: CHECK( 4 == 24 )

testes.cpp:8: ERROR: CHECK( fatorial(10) == 3628800 ) is NOT correct!
values: CHECK( 10 == 3628800 )

=====
[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:     4 |      1 passed |      3 failed |
[doctest] Status: FAILURE!
```



Saída completa

```
$ ./main -s
```

■ Use a opção -s

```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options
=====
testes.cpp:4:
TEST CASE: Testando o fatorial

testes.cpp:5: SUCCESS: CHECK( fatorial(2) == 2 ) is correct!
  values: CHECK( 2 == 2 )

testes.cpp:6: ERROR: CHECK( fatorial(3) == 6 ) is NOT correct!
  values: CHECK( 3 == 6 )

testes.cpp:7: ERROR: CHECK( fatorial(4) == 24 ) is NOT correct!
  values: CHECK( 4 == 24 )

testes.cpp:8: ERROR: CHECK( fatorial(10) == 3628800 ) is NOT correct!
  values: CHECK( 10 == 3628800 )

=====
[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:     4 |      1 passed |      3 failed |
[doctest] Status: FAILURE!
```

Vamos corrigir o programa

- Nova versão do código
- Parece ok?
 - Ainda temos erro. `fatorial(0)`;

```
#include "fatorial.h"

int fatorial(int n) {
    if (n <= 1) return n;
    return n * fatorial(n-1);
}
```

Rodando testes novamente

- Parece que tudo está executando corretamente
- Qual foi o problema?

```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options
=====
[doctest] test cases:      1 |      1 passed |      0 failed |      0 skipped
[doctest] assertions:     4 |      4 passed |      0 failed |
[doctest] Status: SUCCESS!
```

Nosso Teste

- Nunca vai testar fatorial(0)
- Deixa o código com erro

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o fatorial") {
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(4) == 24);
    CHECK(fatorial(10) == 3628800);
}
```

Teste Completo

- Dois casos. Um para o zero
 - Especial
- Outro geral. Valores comuns do fatorial

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("1: fatorial of 0 is 1 (corner case)") {
    CHECK(fatorial(0) == 1);
}

TEST_CASE("2: factorials of 1 and higher are computed (caso geral)") {
    CHECK(fatorial(1) == 1);
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(10) == 3628800);
}
```

Valores inválidos

- Como que o código se comportaria com o fatorial(-20)?
- Ainda tem erro. Precisamos sinalizar que um valor inválido foi passado

Exceções

- Frequentemente uma função não consegue realizar a operação com uma dada entrada
- Ou o estado de um objeto é inválido
- Para sinalizar tal problema fazemos uso de exceções
- Interrompem o código
- Vimos este assunto na nossa última aula

Código Final

```
#include <stdexcept>

#include "fatorial.h"

int fatorial(int n) {
    if (n < 0) {
        throw std::invalid_argument("Não existe fatorial de n < 0");
    }
    if (n <= 1) {
        return 1;
    }
    return n * fatorial(n-1);
}
```

Teste Final

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o caso especial") {
    CHECK(fatorial(0) == 1);
}

TEST_CASE("Testando o fatorial geral") {
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(4) == 24);
    CHECK(fatorial(10) == 3628800);
}

TEST_CASE("Testando o caso invalido") {
    CHECK_THROWS(fatorial(-1));
}
```

Teste Final

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o caso especial") {
    CHECK(fatorial(0) == 1);
}

TEST_CASE("Testando o fatorial geral") {
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(4) == 24);
    CHECK(fatorial(10) == 3628800);
}

TEST_CASE("Testando o caso válido") {
    CHECK_THROWS(fatorial(-1));
}
```

Verifica que lança uma exceção

Problemas ao usar doctest

- Temos dois mains, um do doctest
- Um main do código principal
- O g++ não deixa compilar tal caso
- Gerenciar testes, coberturas, gdb etc etc
- Como resolver?

Problemas ao usar doctest

- Temos dois mains, um do doctest
- Um main do código principal
- O g++ não deixa compilar tal caso
- Gerenciar testes, coberturas, gdb etc etc
- Como resolver?
 - Fazer um makefile que cuida de cada caso
 - Ou utilizar um já pronto

Usando o DocTest em projetos maiores

- Lembrando da nossa estrutura de projeto
- Adicionamos pastas para os testes

. raiz do projeto	
---- Makefile	
---- build/	[diretório]
----	[objetos compilados]
---- third_party/	[bibliotecas de outras pessoas]
---- doctest.h	
---- include/	[diretório]
---- src/	[diretório com código]
---- classe1.cpp	
---- tests/	[diretório com código]
---- test_classe1.cpp	

Cobertura de código

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

Cobertura de código

- 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, ...

Cobertura de código

Ferramentas

- **gcov**
 - é a ferramenta para C++ que verifica a cobertura
 - analisa o 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 relatórios em arquivos .html
 - Facilita identificar e verificar problemas

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
```

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

Exemplo Cobertura

- A linha abaixo indica que executamos todas as linhas do arquivo.
- Ou seja, cobrimos todos os casos!

```
$ g++ --coverage -std=c++14 *.cpp -o main  
$ ./main  
$ gcov fatorial.cpp  
File 'fatorial.cpp'  
Lines executed:100.00% of 6  
fatorial.cpp:creating 'fatorial.cpp.gcov'
```

Exemplo Cobertura

- Se apagarmos o último teste
 - Aquele do valor negativo
- Não cobrimos mais tudo

```
File 'fatorial.cpp'  
Lines executed:83.33% of 6  
fatorial.cpp:creating 'fatorial.cpp.gcov'
```

Cobertura de Testes

- A Cobertura é um bom sinal que cobrimos todo o código
 - Ou boa parte do mesmo
- Porém
 - Não fala nada da qualidade dos testes
 - Podemos ter 100% de cobertura com erros de lógica ainda (exemplo Fatorial)

Escrevendo bons testes

- Bons testes cobrem a maioria dos (ou todos os) fluxos possíveis de execução
- Teste diferentes formas de escrever a mesma função (overloading)
- Além de diferentes implementações de uma mesma interface
 - Um teste por classe, no mínimo

Dicas Para Escrever Testes

- Teste caso base
 - Onde seu algoritmo com certeza funciona
- Teste os *corner cases*
 - Entradas especiais
 - Ordenar um vetor com 1 elemento
- Teste os valores inválidos

Metodologias de Desenvolvimento

- Quando devemos iniciar a fase de testes?
 - Depois de terminada a codificação?
 - Não é uma boa ideia! Por que?
- Desenvolvimento Orientado por Testes (TDD)
 - Foco deve ser no requisito, não no código!
 - Interface → Comportamento
 - Vamos ver com mais detalhes nas próximas aulas...

Exercício

- Como testar peças de xadrez? Isto é, temos uma interface `Peca` com o método `boolean podeMover(int x, int y);`
- Diversas peças implementa a mesma.
- Uma classe `Tabuleiro` com um método `void move(Peca &peca, int x, int y);`

Exercício

- Precisamos de testes para cada classe que implementa a interface Peca
- Além de um teste para Tabuleiro
 - O mesmo que conhece o tamanho