Quem testa
nossos testes ? ? ?

Centeno

Ricardo Gaete

ThoughtWorks®

Vamos **conversar** um pouco sobre . . .

Teste unitário

TDD

Mutation Test

Coverage

Teste unitário

TDD

CI/CD

Coverage

Mutation

Sobre **testes** unitários

- É a **menor parte testável** do seu código
- **Não** serve para **provar** que sua **aplicação funciona**
  - **Garante** que seu código **mantenha** o mesmo **comportamento** após alguma alteração
- Ajuda a **melhorar o design** do seu código

Desenvolvimento guiado por **testes**

- Pequenos ciclos de repetições, onde para cada funcionalidade do sistema um **teste** é criado **antes**
    - **RED** -> **GREEN** -> **Refactoring**
- Código mais limpo e flexível
- Segurança no refactoring e na correção de bugs, pois podemos ver o que estamos ou não afetando

Cobertura de **testes**

- **Métrica** utilizada para medir a **quantidade** de **código** que foi **testado**
- Uma alta taxa de cobertura de testes não indica que ele está bem testado.
    - Mas **uma taxa** muito **baixa** definitivamente **indica** que os seus testes **não são suficientes**
- Ou seja, **cuidado**! A cobertura de testes pode nos dá uma falsa sensação de que está tudo bem, quando não está!

“

Talk is cheap.
Show me the code.

- Linus Torvalds

R$ 17,00

Se comprar **um** produto

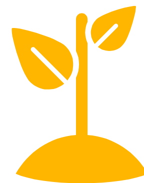
R$ 15,00

Se comprar **20** ou mais produtos

R$ 15,00

Se possuir um **cupom** de desconto

Quem inventou essa de Testes de Mutação ?



- Foi proposto por Richard Lipton, como um estudante, em **1971**
- A primeira implementação de uma ferramenta de teste de mutação foi feita por Timothy Budd, em **1980**
- A ideia era **localizar** e expor as **deficiências** nos conjuntos de testes
- Se apresentou como a melhor maneira de **medir a qualidade** dos nossos testes

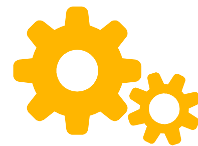
Afinal, o que é Teste de Mutação ?

- Uma mutação é um pequena **modificação** no nosso código
- Esta modificação gera uma versão **mutante** do nosso código

```
if ( amount >= limit ) {
    return amount * 2;
}
```

```
if ( amount > limit ) {
    return amount * 2;
}
```

Funcionamento básico do Teste de Mutação

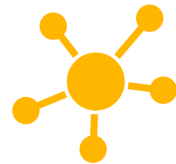


- **Introduz falhas** (mutações) na implementação do código
- **Executa** os testes unitários normalmente **para cada mutação**
- Verificar se os **testes unitários passaram**
- Exibe o resultado das mutações
  - Mutações **mortas**
  - Mutações **sobreviventes**
  - Mutações **others (timeout, error bitecode, etc)**

Porque utilizar Testes de Mutação

- **Qualidade** dos meus **testes**, criando suítes de teste efetivas
- Mostra quanto pode-se confiar numa suíte de teste
- Ajuda na **refatoração** e na criação de testes mais eficientes
- Verificar se alguma implementação está bem testada
- Propósito "**educacional**"

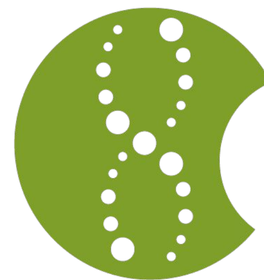
Como utilizar Testes de Mutação



pitest.org

infection-github.io

stryker-mutator.io



ortask.com/mutator

Principais mutações

Conditionals Boundary Mutator

- A mutação de **limite de condicionais** substitui os operadores relacionais com o seu limite equivalente

Original	<	<=	>	>=
Mutação	<=	<	>=	<

Principais mutações

Conditionals Boundary Mutator

```
if ( amount > limit ) {
    return amount * 2;
}
```

- A mutação de **limite de condicionais** substitui os operadores relacionais com o seu limite equivalente

```
if ( amount >= limit ){
    return amount * 2;
}
```

Original	<	<=	>	>=
Mutação	<=	<	>=	<

Principais mutações

Negate Conditionals Mutator

- A mutação de **negação de condicionais** irá alterar todos os condicionais encontrados

Original	==	!=	<=	>=	<	>
Mutação	!=	==	>	<	>=	<=

Principais mutações

Negate Conditionals Mutator

- A mutação de **negação de condicionais** irá alterar todos os condicionais encontrados

```
if ( amount == limit ) {
    return amount * 2;
}
```

```
if ( amount != limit ){
    return amount * 2;
}
```

Original	==	!=	<=	>=	<	>
Mutação	!=	==	>	<	>=	<=

Principais mutações

Remove Conditionals Mutator

- A mutação de **remoção de condicionais** irá remover todas as declarações condicionais
- Garante que as instruções que seguem a condição sejam **sempre executadas**
- A mutação irá alterar apenas as verificações de igualdade (ex: ==, ! =)

Principais mutações

Remove Conditionals Mutator

- A mutação de **remoção de condicionais** irá remover todas as declarações condicionais
- Garante que as instruções que seguem a condição sejam **sempre executadas**
- A mutação irá alterar apenas as verificações de igualdade (ex: ==, ! =)

```
if ( amount == 20 ) {
    return amount * 2;
}

if ( true ){
    return amount * 2;
}
```

```
if( amount == 20 ){
    return amount * 2;
} else {
    return amount;
}

if( false ){
    return amount * 2;
} else {
    return amount;
}
```

Principais mutações

Math Mutator

- A mutação **matemática** substitui as operações aritméticas binárias para aritmética de inteiros ou de ponto flutuante com outra operação.

Original	+	-	*	/	%	&	|	^	<<	>>	>>>
Mutação	-	+	/	*	*	|	&	&	>>	<<	<<

Principais mutações

Math Mutator

- A mutação **matemática** substitui as operações aritméticas binárias para aritmética de inteiros ou de ponto flutuante com outra operação.

```
price = amount + taxes;

price = amount - taxes;
```

Original	+	-	*	/	%	&	|	^	<<	>>	>>>
Mutação	-	+	/	*	*	|	&	&	>>	<<	<<

Principais mutações

Increments Mutator

- A mutação de **incrementos** irá alterar incrementos e decrementos
- Também irá modificar incrementos e decrementos de atribuição de variáveis locais
- Ele irá substituir incrementos com decrementos e vice-versa.

Principais mutações

Increments Mutator

- A mutação de **incrementos** irá alterar incrementos e decrementos
- Também irá modificar incrementos e decrementos de atribuição de variáveis locais.
- Ele irá substituir incrementos com decrementos e vice-versa.

```
public int increaseCounter(int i) {
    i++;
    return i;
}

public int increaseCounter(int i) {
    i--;
    return i;
}
```

Principais mutações

Return Values Mutator

- A mutação **valor de retorno** altera os valores de retorno do método.

Original	Mutação
`boolean`	Substitui um retorno `true` por `false`. E vice-versa
`int` `byte` `short`	Substitui um retorno `0` por `1`. Nos outros casos retorna sempre `0`
`long`	Incrementa `+1` no retorno
`float` `double`	Substitui o valor original por `-(x+1.0)`
`Object`	Retorna sempre `null`

Principais mutações

Return Values Mutator

```java
public Passenger createPassenger() {
    return new Passenger();
}
```

- A mutação **valor de retorno** altera os valores de retorno do método.

```java
public Passenger createPassenger() {
    new Passenger();
    return null;
}
```

Original	Mutação
`boolean`	Substitui um retorno `true` por `false`. E vice-versa
`int` `byte` `short`	Substitui um retorno `0` por `1`. Nos outros casos retorna sempre `0`
`long`	Increment `+1` no retorno
`float` `double`	Substitui o valor original por `-(x+1.0)`
`Object`	Retorna sempre `null`

Principais mutações

Void Method Call Mutator

- A mutação de **chamada de métodos *void*** remove as chamadas de métodos *void*
- As chamadas de construtores não são consideradas chamadas de métodos void. Para isto existe um mutador específico
- Também existe uma outra mutação para métodos não void

Principais mutações

Void Method Call Mutator

- A mutação de **chamada de métodos *void*** remove as chamadas de métodos void
- As chamadas de construtores não são consideradas chamadas de métodos void. Para isto existe um mutador especifico
- Também existe uma outra mutação para métodos não void

```java
public void addTaxes(Price price) {
    // does something
}
public Price getPrice() {
    Price price = new Price();
    addTaxes(price);
    return price;
}
```

```java
public void addTaxes(Price price) {
    // does something
}
public Price getPrice() {
    Price price = new Price();
    return price;
}
```

Assim como o mundo, as mutações têm **problemas**

- Testes de mutação **demoram muito**. Mas muito mesmo!
- Eles são **computacionalmente** muito **pesados**
- Limitado aos operadores de mutação
- Útil para teste unitários, mas **não substitui** os **testes funcionais**
- Em alguns casos, onde a mutação **equivale ao código original**, é gerado um **falso positivo**:
  - while a == b | while a >= b

100 Classes

10 Testes unitários

2ms Tempo por cada teste

2s Tempo testes unitários (100 x 10 x 2ms)

2s Tempo testes unitários (100 x 10 x 2ms)

8 Mutantes por classe = **800** Mutantes

26m Tempo total (800 x 2s)

20s Tempo testes unitários (**1000** x 10 x 2ms)

8 Mutantes por classe = **8.000** Mutantes

1d20h Tempo total (8.000 x 20s)

20s Tempo testes unitários (**1000** x 10 x 2ms)

8 Mutantes por classe = **8.000** Mutantes

~~**1d20h** Tempo total (8.000 x 20s)~~

2m40s Tempo total (8.000 x 10 x 2ms)

Porém, nem tudo está perdido. Há uma **solução**

- Primeiramente, escrever **testes unitários** que sejam **rápidos**
- **Não** executar o teste de mutação **toda vez** que roda a suíte de testes
- Análise incremental : **Rodar** o teste **só nas partes que sofreram alterações**

> Talk is cheap.
> Show me the code.
>
> - Linus Torvalds

Dúvidas? Perguntas?

Quer falar com a gente?

- @centeno | centeno@diegocenteno.com
- @rgaete | ricardo.gaete@gmail.com