

# **Qualidade de software: defeitos e explosão combinatória**

**Qualidade de Software**

André Koscianski

Michel Soares

**Editora Novatec**

## Mito:



**"Um software de qualidade não tem defeitos ( '*bugs*' ) "**

## Mito:



**"Um software de qualidade não tem defeitos ( '*bugs*' ) "**

## Verdade:

**Qualidade depende de diversos fatores,  
que incluem  
o preço que se dispõe pagar para reduzir o número de defeitos.**



## Observe esta rotina:

```
double fun1 (bool flag) {  
    int n = 7 * 14 / 2 + 1;  
    if (flag) {  
        while (--n > 2)  
            ;  
    } else  
        n = 3;  
    return 3.14 * n;  
}
```

Quantos valores diferentes ela pode retornar?

Veja **a mesma rotina** escrita de um modo mais direto:

```
double fun1 (bool flag) {  
    int n = 7 * 14 / 2 + 1;  
    if (flag) {  
        while (--n > 2)  
            ;  
    } else  
        n = 3;  
    return 3.14 * n;  
}
```

```
double fun1_simples (bool flag) {  
    return flag ? 6.28 : 9.42;  
}
```

ou seja, neste caso:

- Só há **dois** valores possíveis de entrada.
- Então, só há **duas** maneiras diferentes de executar a rotina

(isso independe de quão complicado o código possa parecer...)

Em consequência, só há dois valores possíveis de saída.

## Outro exemplo: quantos valores diferentes a rotina a seguir pode retornar?

```
double fun2 (bool flag1, bool flag2) {
```

## Resposta:

```
double fun2 (bool flag1, bool flag2) {  
    if ( flag1 &&  flag2) return 1.0;  
    if ( flag1 && !flag2) return 2.0;  
    if (!flag1 &&  flag2) return 3.0;  
    if (!flag1 && !flag2) return 4.0;  
}
```

Há quatro entradas possíveis.

Logo, **podem haver no máximo** quatro saídas possíveis.



**Observe este último exemplo: quantas possibilidades há neste caso?**

```
double fun3 (char c) {
```

## No máximo 256. Veja:

```
double fun3 (char c) {  
    return (double) c;  
}
```

*256 saídas possíveis*

```
double fun3 (char c) {  
    return (double) (c / 2);  
}
```

*128 saídas possíveis*

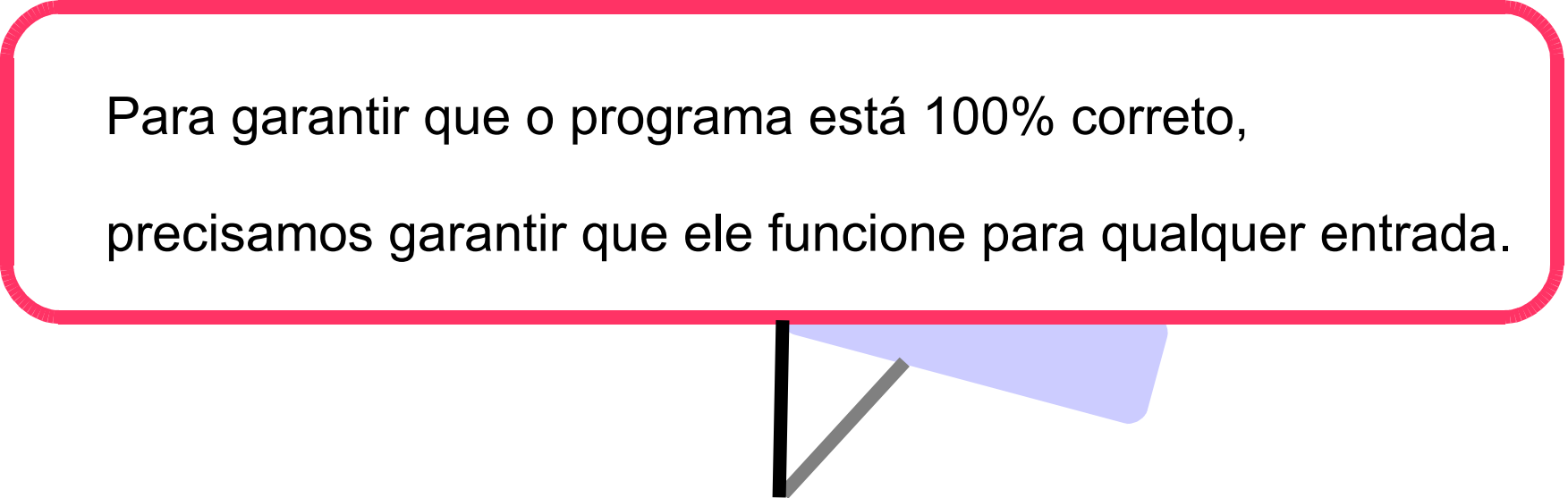
```
double fun3 (char c) {  
    return 1.234;  
}
```

*1 única saída possível*

## Por que o número de entradas é crítico?

Cada diferente entrada representa uma possibilidade de execução.

Para garantir que o programa está 100% correto,  
precisamos garantir que ele funcione para qualquer entrada.



O trecho a seguir vem de um programa real.

```
if (!_traj.thecollisiongrid2 (_x0 + (float) v1.x,  
                             _y0 + (float) v1.y)) {  
  
    d = (float) v1.length ();  
  
    _traj.prepare (_x0, _y0,  
                  _x0 + (float) v1.x,  
                  _y0 + (float) v1.y,  
                  12, d, 7, 25  
                  );
```

**Pergunta:** para quais valores de entrada o método destacado funciona corretamente?

## Isto é: quantas entradas diferentes são possíveis?

```
_traj.prepare (  
    _x0                // float, 32 bits  
    ,_y0                // float, 32 bits  
    ,_x0 + (float) v1.x // float, 32 bits  
    ,_y0 + (float) v1.y // float, 32 bits  
    ,12                // int  , 32 bits  
    ,d                  // float, 32 bits  
    ,7                  // int  , 32 bits  
    ,25                 // int  , 32 bits  
);
```

Esta rotina recebe  $8 \times 32 = 256$  bytes = 1024 bits.

Isso totaliza  $2^{1024}$  entradas diferentes <sup>(1)</sup>.

*(1) isto sem falar do estado interno do objeto "\_traj" !!*

## Como garantir o funcionamento da rotina à 100% ?

Suponha que um computador testasse a rotina, gerando todas as entradas possíveis.

Se esse computador gerasse 1 bilhão de testes por segundo,

depois de 1 ano,

teríamos testado apenas cerca de  $3^{16}$  combinações !

Estaríamos ainda muito, muito longe de  $2^{1024}$  ...



**Testar um programa inteiro é ainda mais difícil**

**do que testar uma simples (!) rotina ...**



## Uma analogia:



Uma dessas pessoas tem gripe e você deve descobrir qual delas !



## O problema é insolúvel?

Não exatamente.

A complexidade combinatória sempre nos atormentará.

Mas podemos buscar tratá-la.

veja o próximo exemplo.

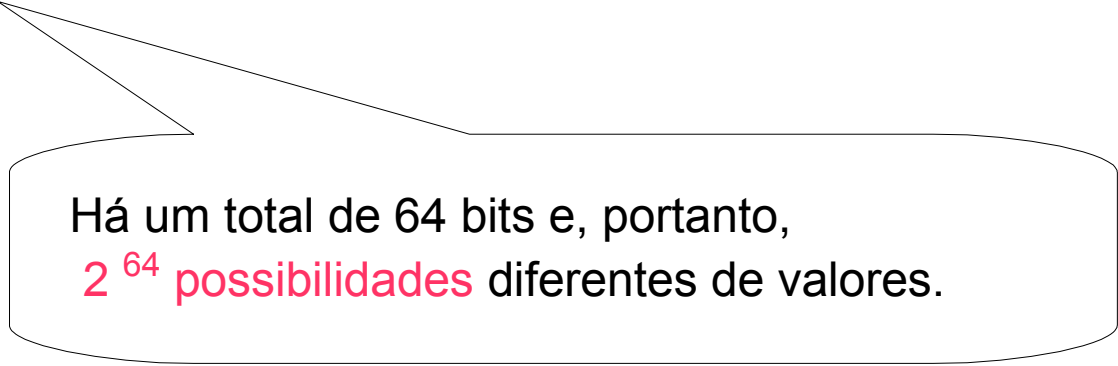
## Este é um caso tratável:

Suponha que este código use inteiros de 32 bits:

...

```
x = y / (z - fun2 (x));
```

...



Há um total de 64 bits e, portanto,  
 $2^{64}$  possibilidades diferentes de valores.

### Entretanto:

A falha acontece em menos de um bilionésimo de todas combinações ( $2^{-32}$ ).

Ainda melhor: os casos problemáticos podem ser resolvidos em uma linha de código .

## Solução possível:

Por exemplo, modifique o código assim:

```
...  
if (0 == (aux = (z - fun2 (x))))  
    socorro ();  
  
x = y / aux;  
...
```



## Porém:

No exemplo acrescentamos código para proteger o código.

O programa ficou mais seguro, mas também menos simples.

## Porém:

No exemplo acrescentamos código para proteger o código.

O programa ficou mais seguro, mas também menos simples.

### Perguntas importantes:

- O que fazer se tivermos 10000 linhas de código?
- Se o código que protege o código contiver erros, quem nos salvará?

## Onde estamos:

- o problema, em toda sua generalidade, permanece em aberto;
- não faltam exemplos práticos de defeitos, indo do vulgar até software militar;
- a regra "evitem todo mal e todos os defeitos" pode ser pregada em certos contextos, mas não é realística em engenharia;
- o custo de remoção de defeitos é inversamente proporcional aos defeitos restantes;
- e preço é um componente determinante da qualidade!

O que fazer?



Bem , milagres não existem.



## Passado o cortejo do santo: e agora?

As próximas transparências são bastante carregadas.  
Isso é não-didático, mas aqui é proposital.

Cada uma leva a uma longa lista de assuntos.

Você não achava que algumas transparências  
acabariam com os problemas de software.

Achava?



## Algumas técnicas:

**Planejar e projetar cuidadosamente.**

Isso inclui requisitos e **cronograma** de implementação.

Por exemplo: você inicia o almoço pensando em *spaghetti alla carbonara* e, faltando meia hora para colocar a mesa, decide trocar por filé *parmigiana*.

Resultado: terminará comendo angu !

Estranhamente muitos fabricam software assim e reclamam dos resultados...

**Construa o código cuidadosamente.**

Isso inclui:

- empregar um controle de versões;
- fazer controle e rastreabilidade de requisitos;
- aplicar técnicas de inspeção;
- definir e seguir padrões em equipe;
- utilizar técnicas como tratamento de exceções, bibliotecas como Electric Fence...

**Permita** que o código seja construído cuidadosamente.

Isso não será possível se o cronograma e o planejamento não forem honrados.

## Algumas técnicas:

**Use o óbvio: a engenharia vista nos bancos de escola.**

Por exemplo, aplique o que **Wirth** e **Knuth** nos ensinaram há décadas, como verificar pré e pós-condições de cada sub-rotina ou cada trecho de código.

Compre urgente livros desses autores, na remota hipótese de você não os ter lido ainda !!!

**Aplique a preguiça onde ela for útil.**

Use bibliotecas e código pronto **sempre** que possível.

Mas, **não** tenha preguiça de estocar e documentar cuidadosamente esses artefatos!

**Analise as possibilidades de implementação.**

Considere a plataforma utilizada. Por exemplo, trocar C++ por C# <sup>(1)</sup> e com isso ganhar garbage collection, arrays verificados, type checking, checked { }  
Obviamente isso abrange compiladores, bibliotecas, SGBDs, etc..

(1) Disponível gratuitamente nas implementações GNU e Mono, em Linux e Windows.  
Tudo isto para explicar que a linguagem não é proprietária...

## Algumas técnicas:

### Pesquise

Não é preciso tornar-se pesquisador na acepção do termo: apenas informe-se.

Exemplo: **toda a documentação do CMMI está disponível, gratuitamente, na Internet.**

### Invista

Problemas mais críticos exigem, de regra, mais recursos.

Ferramentas, por exemplo: existem várias.

Geradores de casos de teste não são ferramentas de ficção científica !

Consultorias são bastante desejáveis e podem ser encontradas na iniciativa privada.

Uma parceria com uma Universidade também pode ser um grande negócio.

Técnicas como Especificação Formal, ou a Interpretação Abstrata (P. Cousot) são menos acessíveis mas apresentam resultados relevantes.