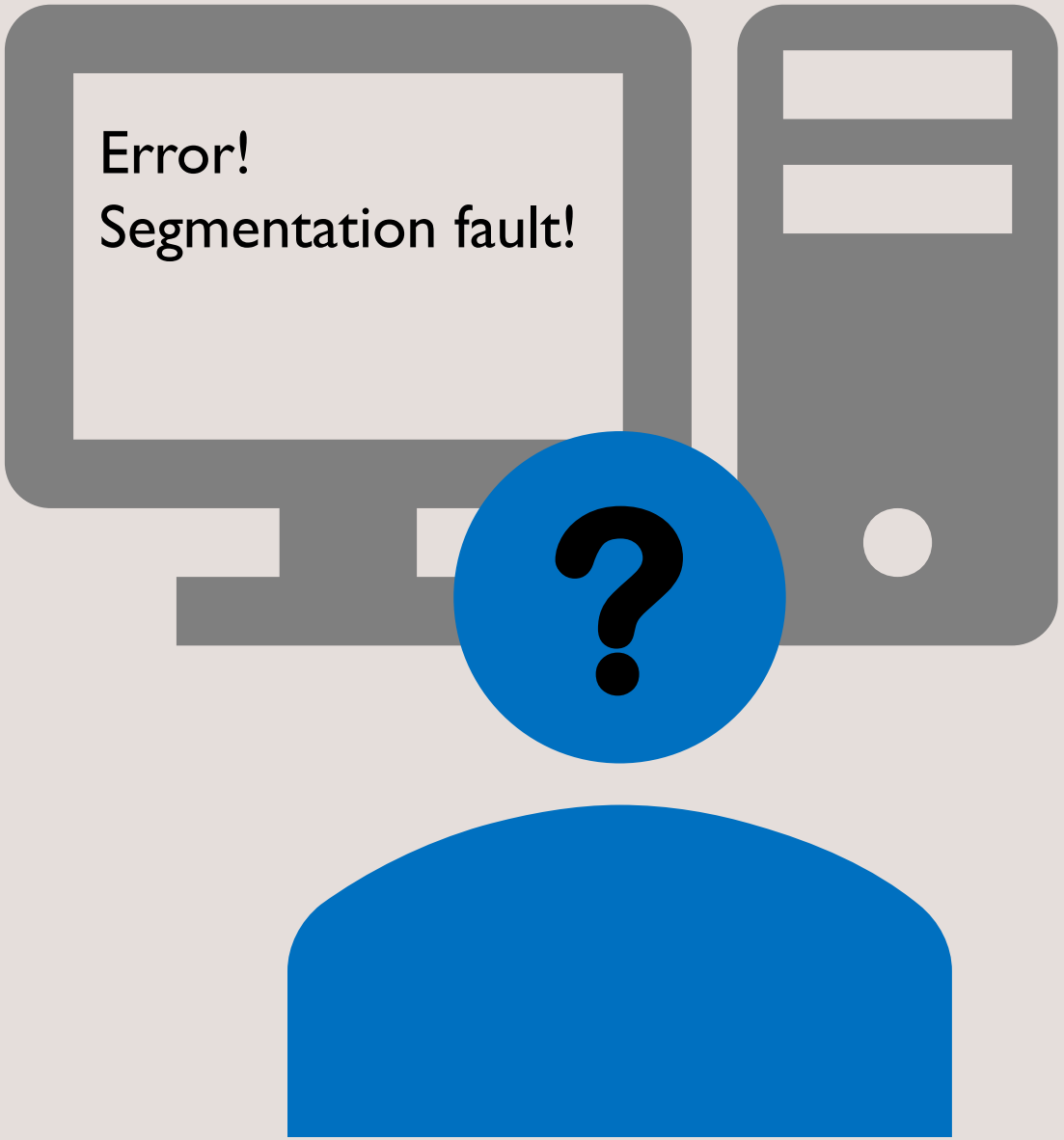
MEU PROGRAMA NÃO FUNCIONA! E AGORA?

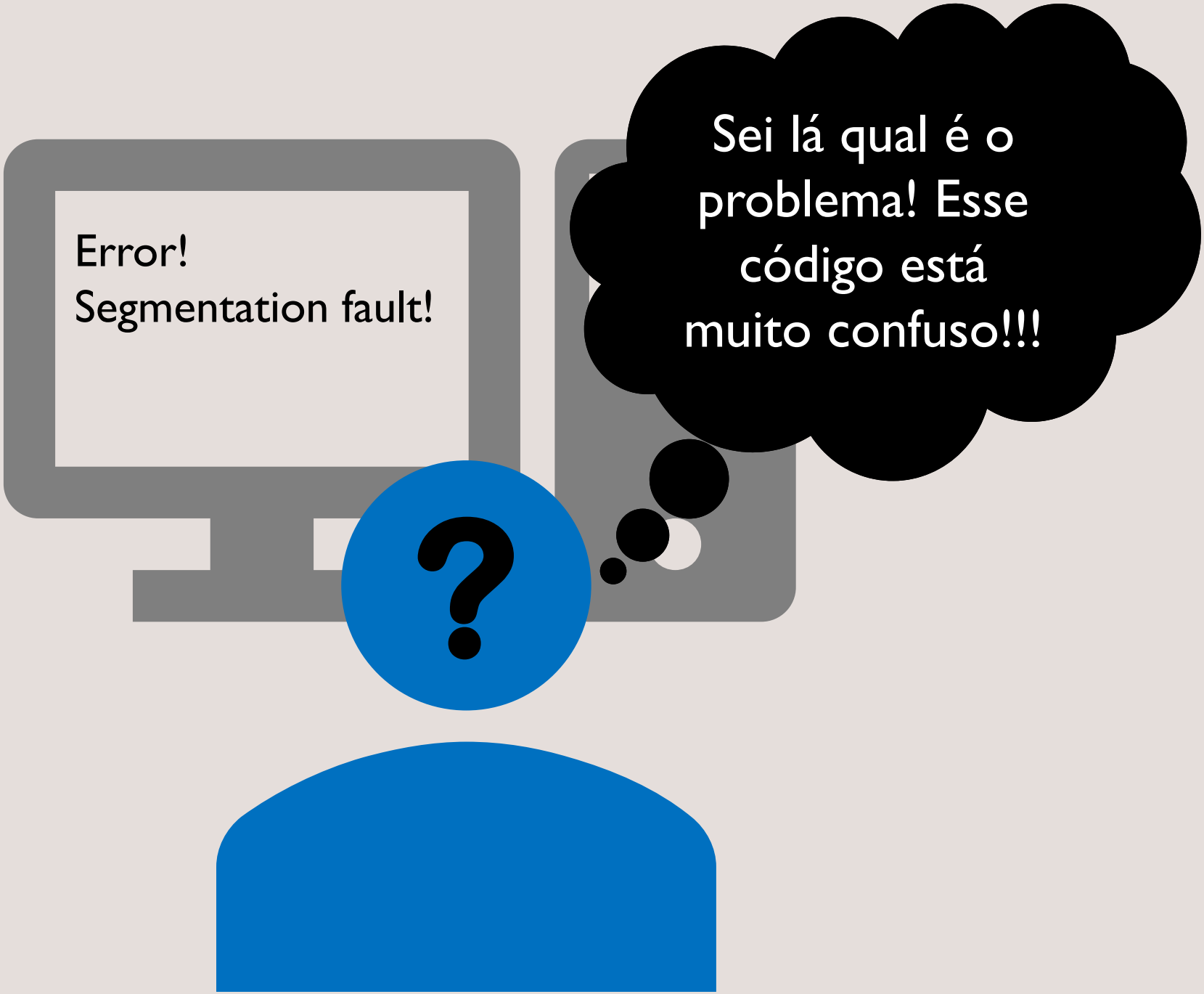
PROF. PAULO HENRIQUE PISANI
PROF. MONAEL PINHEIRO RIBEIRO
UNIVERSIDADE FEDERAL DO ABC (UFABC)

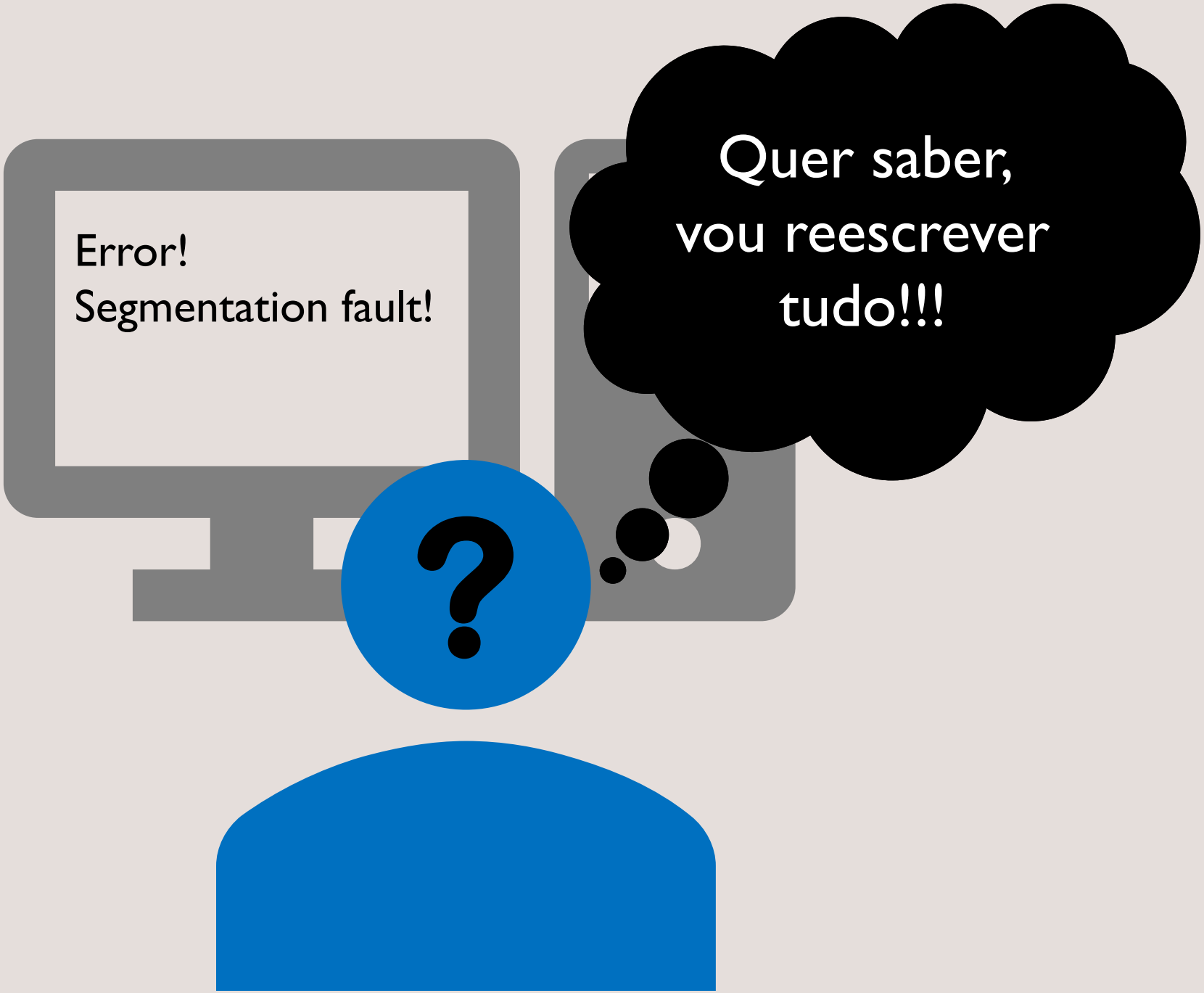
19/junho/2018

UFABC

IV SEMANA DO CMCC
Mercado e Inovação


Error!
Segmentation fault!
?


Error!
Segmentation fault!
Sei lá qual é o problema! Esse código está muito confuso!!!
?


Error!
Segmentation fault!
Quer saber, vou reescrever tudo!!!
?

Quer saber,

NNÃÃÃÃÃÃÃÃÃÃÃÃOOOOOO!!!!!!!!!!

Neste curso, vamos discutir técnicas para **evitar e encontrar erros/bugs**;

Muitas vezes, é **inviável** reescrever um sistema existente (além disso, **a própria reescrita pode gerar outros erros**…)

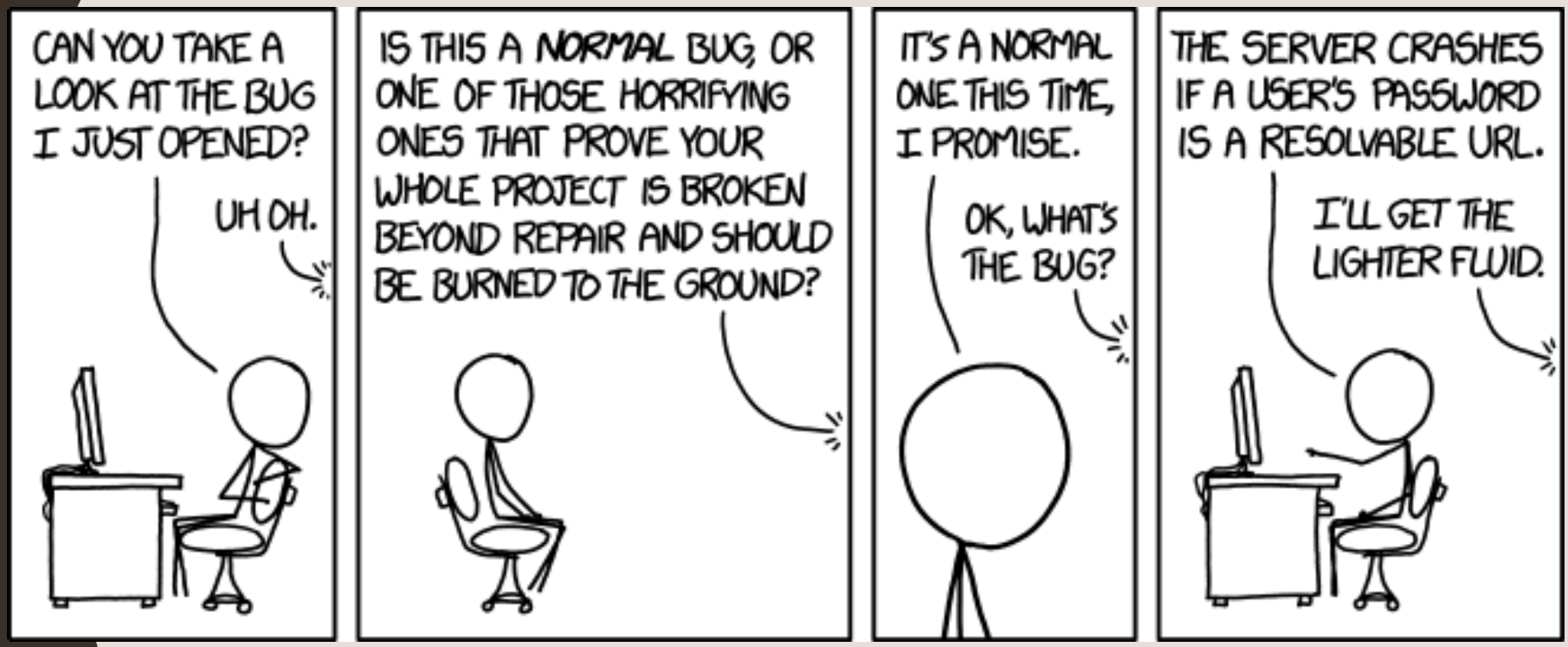
TÓPICOS

- **Medidas proativas -** Prevenção de erros/bugs:
  - "Programação defensiva"
  - Testes unitários
- **Medidas reativas -** Depuração:
  - "Raiz"
  - "Nutella"

PREVENÇÃO DE BUGS

PROGRAMAÇÃO DEFENSIVA, TESTES UNITÁRIOS

Material do curso:
http://professor.ufabc.edu.br/~paulo.pisani/abcdebug/



https://xkcd.com/1700/

VAMOS NOS DEFENDER PRIMEIRO!

- **Programação defensiva (Blunden, 2012)**: medidas proativas para evitar erros;

- Quando programar, você deve se defender de seus erros e de erros de outros programadores!

PROGRAMAÇÃO DEFENSIVA

- Princípio fundamental:
  - **Permitir o uso/teste de componentes/módulos do programa de forma independente!**

- Alta coesão; Exemplo a seguir
- Baixo acoplamento;
- Limitar escopo de variáveis (evitar variáveis globais);
- Evitar efeitos colaterais;
- Verificar validade dos argumentos;
- Verificar tamanho/limites dos tipos de dados;
- Inicialize todas as variáveis;
- Evite valores "hardcoded";
- Programa deve falhar de forma "elegante".

EXEMPLO

- Escreva um método que converta um número decimal (int) em binário;

Solução 1

```
public static void convBinarioPrint(int numero) {
    if (numero == 0) return;

    convBinarioPrint(numero / 2);
    System.out.print(numero % 2);
}
```

Chamada do método: `convBinarioPrint(25);`

Solução 2

```
public static String convBinarioStr(int numero) {
    if (numero == 0) return "";

    return convBinarioStr(numero / 2) + (numero % 2);
}
```

Chamada do método: `System.out.print(convBinarioStr(25));`

EXEMPLO

- Escreva um método que converta um número decimal (int) em binário;

Solução 1

```
public static void convBinarioPrint(int numero) {
    if (numero == 0) return;

    convBinarioPrint(numero / 2);
    Syste
}
```

Chamada

Qual solução é melhor?

Solução 2

```
public static String convBinarioStr(int numero) {
    if (numero == 0) return "";

    return convBinarioStr(numero / 2) + (numero % 2);
}
```

Chamada do método: `System.out.print(convBinarioStr(25));`

PROGRAMAÇÃO DEFENSIVA

- Princípio fundamental:
  - **Permitir o uso/teste de componentes/módulos do programa de forma independente!**

- Alta coesão;
- Baixo acoplamento;
- Limitar escopo de variáveis (evitar variáveis globais);
- Evitar efeitos colaterais;
- Verificar validade dos argumentos;
- Verificar tamanho/limites dos tipos de dados;
- Inicialize todas as variáveis;
- Evite valores "hardcoded";
- Programa deve falhar de forma "elegante".

PROGRAMAÇÃO DEFENSIVA

- Nomes representativos para variáveis/parâmetros;
- Adotar um guia de estilo de codificação;
- Atualizar comentários *inline* (e.g. javadoc):
  - Programas sao atualizados com frequência e rapidamente a documentação pode tornar-se desatualizada.
- Código que necessita de muita documentação provavelmente precisa ser reescrito;
- Comentários não devem ser longos, assim tornam-se mais objetivos e facilitam sua atualização com o tempo.

Exemplo a seguir

EXEMPLO

O que esse método faz?

```
public static void processaVetor(int[] v) {
for (int i1 = 0; i1 < v.length-1; i1++) {
int aux1 = i1;
int aux2 = v[i1];
for (int i2 = i1+1; i2 < v.length; i2++) {
if (v[i2] < aux2) {
aux1 = i2;
aux2 = v[i2];}}
v[aux1] = v[i1];
v[i1] = aux2;}}
```

EXEMPLO

O que esse outro método faz?

```
public static void ordenaSelecao(int[] v) {
    for (int i = 0; i < v.length-1; i++) {
        int indiceMenor = i;
        int valorMenor = v[i];
        for (int k = i+1; k < v.length; k++) {
            if (v[k] < valorMenor) {
                indiceMenor = k;
                valorMenor = v[k];
            }
        }
        v[indiceMenor] = v[i];
        v[i] = valorMenor;
    }
}
```

PROGRAMAÇÃO DEFENSIVA

- Nomes representativos para variáveis/parâmetros;
- Adotar um guia de estilo de codificação;
- Atualizar comentários inline (e.g. javadoc):
  - Programas sao atualizados com frequência e rapidamente a documentação pode tornar-se desatualizada.
- Código que necessita de muita documentação provavelmente precisa ser reescrito;
- Comentários não devem ser longos, assim tornam-se mais objetivos e facilitam sua atualização com o tempo.

TESTES UNITÁRIOS

- Além de seguir os princípios da **programação defensiva**, testes unitários podem ajudar a encontrar erros antes de entregar um software ao cliente!
- **Teste unitário**: teste de cada componente de um sistema (geralmente automatizado).

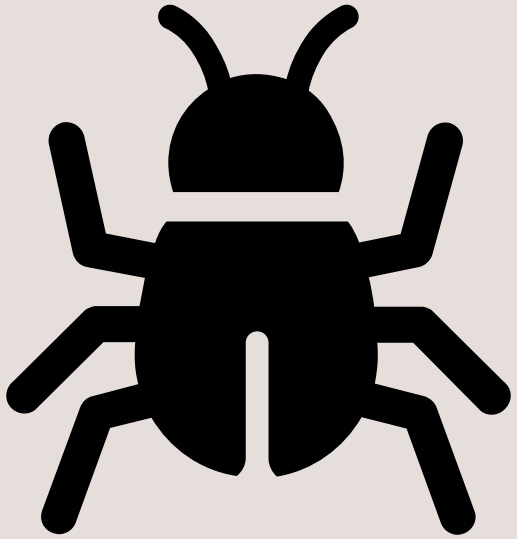
DEPURAÇÃO

Ok, perfeito, tudo isso parece muito legal.

Mas e quando encontro um erro em sistemas já existentes???

MAS É UM BUG MESMO?

- Primeiro temos que averiguar se é um erro/bug;
- Alguns problemas são na verdade resultado de uso incorreto;
  - Por exemplo, a função carregaDados não funciona com esse arquivo do Excel! (a função exige um arquivo CSV, mas foi usada com um arquivo XLS)



Bom, agora que sabemos que é um bug/erro… **vamos solucioná-lo!!!**

FONTES COMUNS DE BUGS

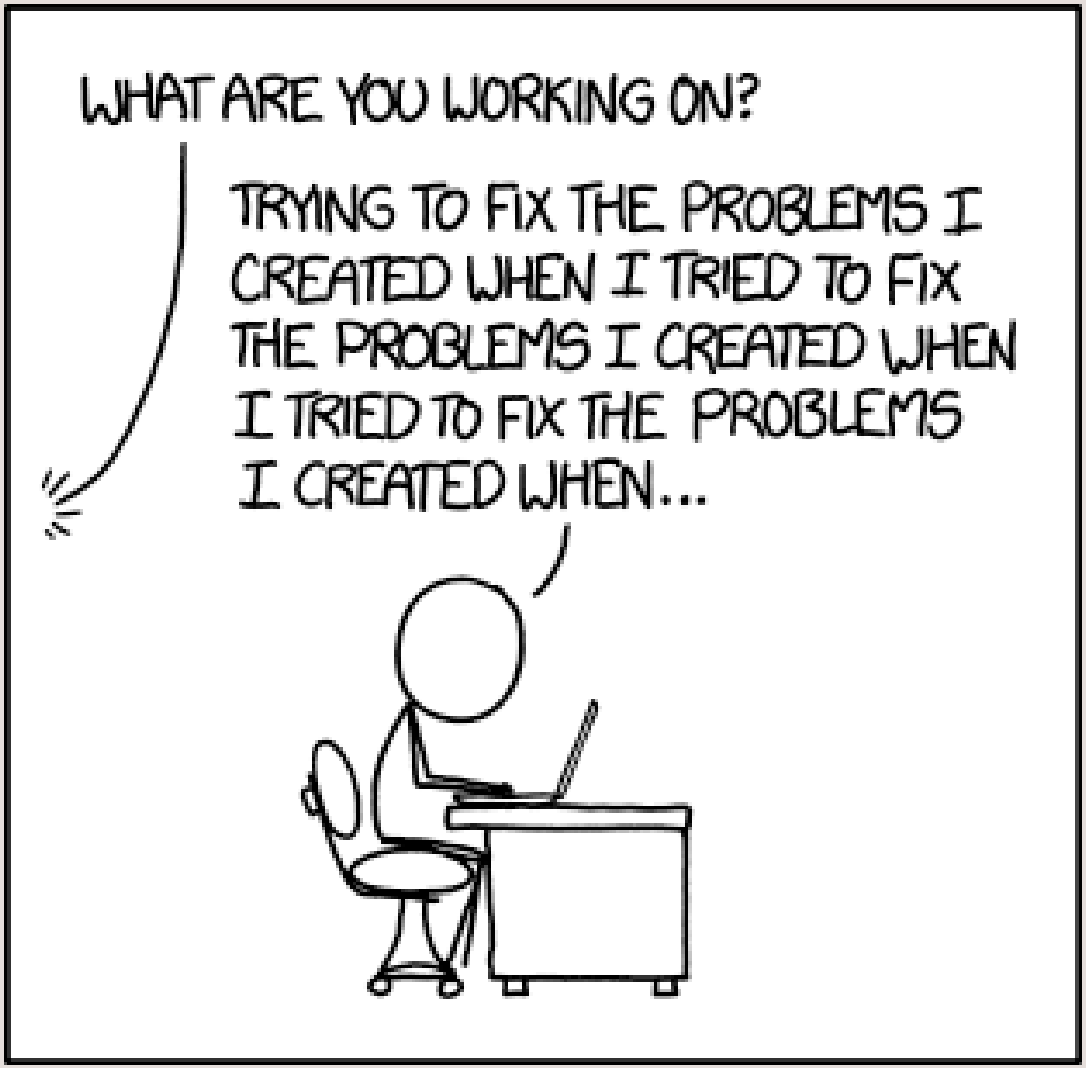
- Falta de inicialização de variáveis;
- Ponteiros inválidos;
- Problemas de sincronização de threads.

ESTRATÉGIAS

- **Primeiro: reproduzir o erro!**

- **Verificar mudanças recentes (sistemas de controle de versão são muito úteis nesse caso)**
  - **Estratégia incremental;**
  - **Estratégia da busca binária;**

- Quando encontrar o erro, aplique as correções aos poucos:
  - Se o problema continuar, haverá muitos suspeitos!



https://xkcd.com/1739/

DEPURAÇÃO "RAIZ"

PRINTF

- Incluir registros da sequência de execução (pode ser com printf mesmo!);
  - Dessa forma, podemos acompanhar a **sequência de prints** e **identificar a região do erro**;
  - Além de marcar **pontos chave**, os prints podem mostrar valores de **variáveis importantes** ao longo da execução.

PRINTF

- Exemplo: **programa atletas**;
  - Primeiro passo: verificar em que parte o programa para;
  - Depois: verificar se entrada está correta;
  - Isolar fonte do erro;
  - Corrigir!

PROGRAMA ATLETAS

Arquivo: atletas.c

Dados os nomes, sexo, altura e peso de 500 atletas calcular a média da altura dos atletas e a média do peso das atletas.

PROGRAMA ATLETAS

- Princípios de **programação defensiva** quebrados:
  - Valores "hardcoded";
  - Verificar validade de argumentos.

PROGRAMA SORVETERIA

Arquivo: sorveteria.c

PROGRAMA SORVETERIA

- Redirecionamento de entrada:
  - Facilita testes com muitas entradas/saídas;
  - Podemos usar um programa para ver as diferenças nas sáidas (diff, WinMerge, FC, etc).
- Princípios de **programação defensiva** quebrados:
  - Verificar tamanho/limites dos tipos de dados.

PROGRAMA LISTA LIGADA

Arquivo: listaligada.c

PROGRAMA LISTA LIGADA

- Princípios de **programação defensiva** quebrados:
  - Inicialize todas as variáveis;

DEPURAÇÃO RAIZ - LOGGING

- Registro de execução (*logging*);
  - Pode ser ativado/desativado;
  - O logging auxilia na depuração, pois serve como indicador da localização de erros.
- Sempre registrar:
  - Origem: Módulo/Função ou Classe/método
  - Tipo: Erro / Trace
  - Mensagem

DEPURAÇÃO "NUTELLA"

USANDO UM DEPURADOR

O QUE É UM DEPURADOR?

- Permite observar o funcionamento de um programa sem influenciar seu fluxo de execução;
  - Em C, um depurador comum é o GDB;
  - Geralmente, ambientes de desenvolvimento possuem depuradores integrados (Eclipse, Visual Studio, NetBeans, CodeBlocks, etc).

DEPURADOR

- Alguns recursos comuns oferecidos por depuradores:
  - Execução linha a linha;
  - Breakpoints;
  - Watches;
  - Pilha de chamadas (call stack).

BREAKPOINT

- **Interrompe** a execução do programa na linha especificada pelo ***breakpoint*** e retorna para o depurador;
- A partir de um breakpoint, podemos examinar os valores de variáveis, seguir linha a linha, por exemplo.

EXECUÇÃO LINHA A LINHA

- Start/Continue
- Step over (next line)
- Step into
- Step out
- Stop

WATCHES

- Mostra o valor de **variáveis** e **expressões**;
- Dessa forma, podemos **acompanhar mudanças em variáveis chave**, que podem indicar a origem de um bug.

PROGRAMA ORDENAÇÃO

Arquivo: ordenacao.c

PROGRAMA ORDENAÇÃO

- Princípios de **programação defensiva** quebrados:
  - Limitar escopo de variáveis (evitar variáveis globais);
  - Baixo acoplamento;

PROGRAMA ARVORE BINÁRIA DE BUSCA

Arquivo: abb.c

PROGRAMA ÁRVORE BINÁRIA DE BUSCA

- Princípios de **programação defensiva** quebrados:
  - Inicialize todas as variáveis.

PROGRAMA MISTÉRIO

Arquivo: misterio.c

VALGRIND

- Ferramenta de análise que pode detectar vazamentos de memória (memory leaks), acessos a áreas de memória indevidas, etc.

```
valgrind ./misterio.exe -v -leak-check=full
```

```
valgrind ./abb.exe -v -leak-check=full
```

MAS QUAL ESTRATÉGIA DEVO USAR?

DEPURAÇÃO: RAIZ VS NUTELLA

DEPURAÇÃO "RAIZ"

- printf
- printf
- printf, printf, printf
- Coloca para executar

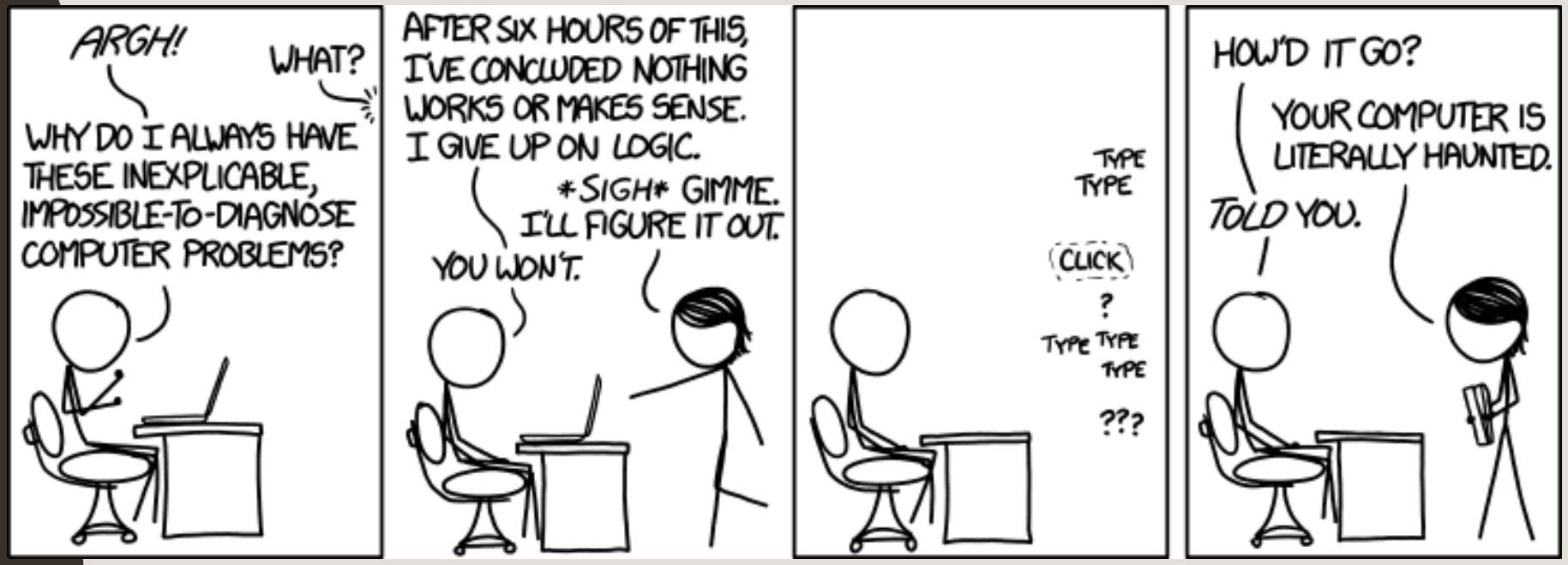
DEPURAÇÃO "NUTELLA"

- Breakpoints
- Watches
- Step over, step into, step out
- Coloca para "debugar"

Figura da esquerda: https://www.freeimages.com/photo/grandcomputer-2-1243223
Figura da direita: https://www.istockphoto.com/br/foto/3-d-moderno-computador-local-de-trabalho-gm480393652-68699345?irgwc=1&esource=AFF_IS_IR_SP_FreeImages_246195&asid=FreeImages&cid=IS

MAS QUAL ESTRATÉGIA DEVO USAR?

- Depuração "raiz":
  - Não precisa de depurador;
  - Geralmente mais rápida para testar (não precisa ir linha a linha).
- Depuração "nutella":
  - Requer acesso a um depurador;
  - Permite um controle melhor do programa (execução linha a linha, watches);
- Uma estratégia possível é uma combinação dos dois também:
  - Depuração "raiz" pode ser usada para encontrar a região do problema e então depuração "nutella" pode ser aplicada nesta região.



https://xkcd.com/1316/

REFERÊNCIAS

- Bill Blunden. Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code. Apress, 2012.
- Donald E. Knuth. The Art of Computer Programming. Addison-Wesley, 1973.
- Robert Sedgewick. Algorithms in C, Parts 1-4: Fundamental Algorithms, Data Structures, Sorting, Searching. Addison-Wesley Professional, 1997.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest; Clifford Stein. Introduction to Algorithms. MIT Press, 2009.