Boas práticas

Código Limpo

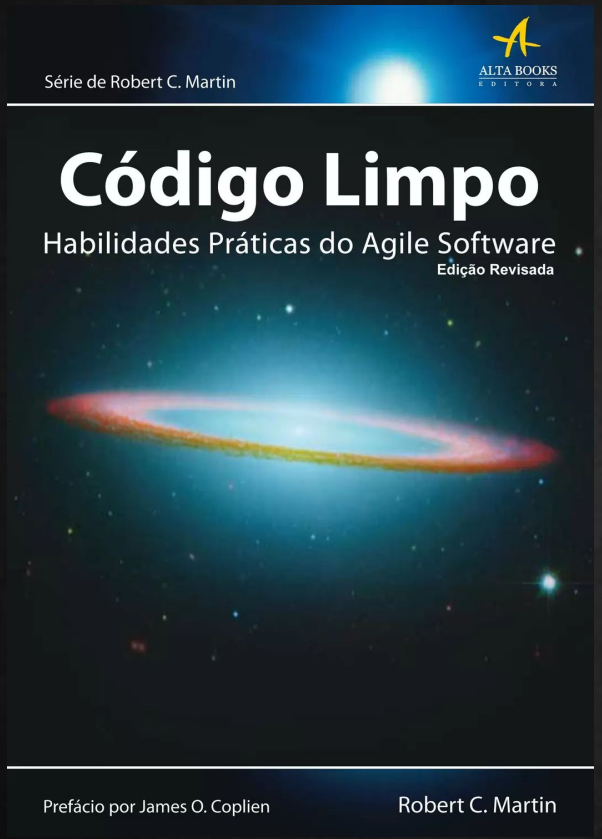
Gleryston Matos

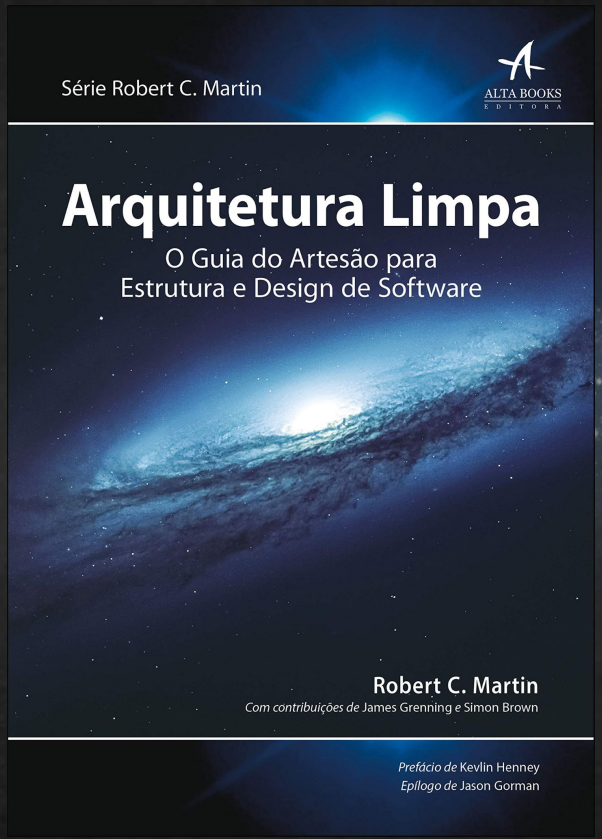
Código Limpo

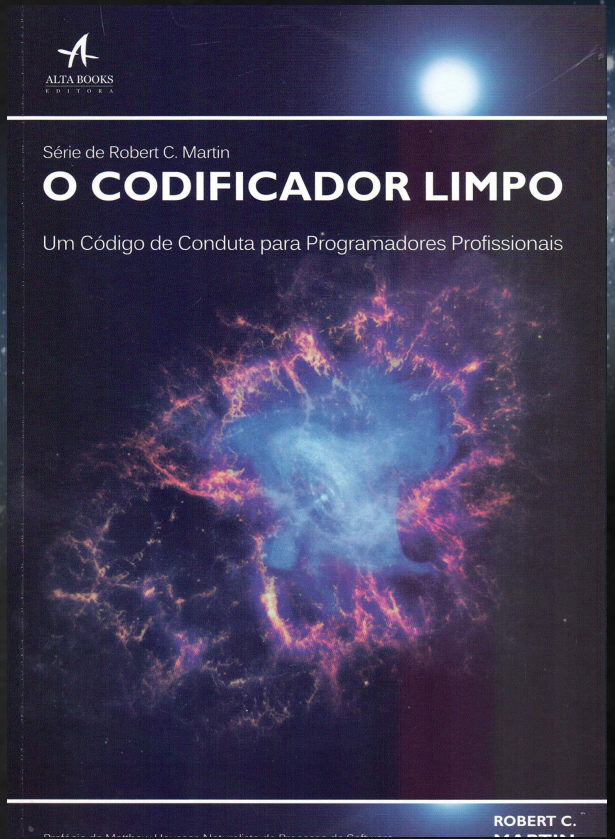
Gleryston Matos:

1. Graduado em sistemas e informação.
2. + 9 anos atuando como desenvolvedor de software.
3. Líder técnico na Fortes Tecnologia.
4. Membro e organizador do DUG-CE.
5. Escritor de artigos no medium.
6. Informações:
    - GitHub: https://github.com/GlerystonMatos
    - Medium: https://medium.com/@glerystonmatos
    - Linkedin: https://www.linkedin.com/in/glerystonmatos
    - Youtube: https://www.youtube.com/user/Glerystonmatos

DUG-CE:
Grupo: https://t.me/DUGCE
Canal: https://www.youtube.com/c/DUGCE-CANAL


Série de Robert C. Martin
ALTA BOOKS
EDITORA
Código Limpo
Habilidades Práticas do Agile Software
Edição Revisada
Prefácio por James O. Coplien
Robert C. Martin


Série Robert C. Martin
ALTA BOOKS
EDITORA
Arquitetura Limpa
O Guia do Artesão para
Estrutura e Design de Software
Robert C. Martin
Com contribuições de James Grenning e Simon Brown
Prefácio de Kevlin Henney
Epílogo de Jason Gorman


ALTA BOOKS
EDITORA
Série de Robert C. Martin
O CODIFICADOR LIMPO
Um Código de Conduta para Programadores Profissionais
ROBERT C.

Código Limpo

Robert Cecil Martin:

Robert Cecil Martin, também conhecido como "Uncle Bob" (Tio Bob em português), é uma grande personalidade da comunidade de desenvolvimento de software e métodos ágeis, atuando na área desde 1970.

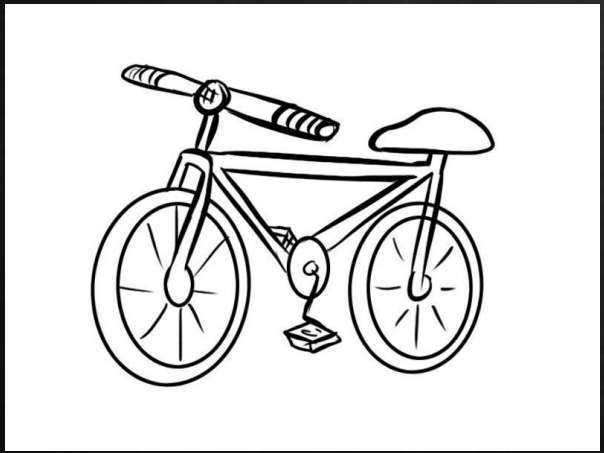
Atualmente é consultor internacional e autor de vários livros abordando o tema.

Uncle Bob foi um dos 17 signatários originais do Manifesto Ágil em 2001.

Antes de começar

Antes de começar temos que entender um ponto importante, quando falamos de código limpo, para que possamos escrever um bom código, vamos precisar de duas coisas inicialmente, conhecimento dos princípios, padrões e práticas que um bom profissional precisa ter e muito trabalho duro e prática desses conhecimentos.

É que nem aprender a andar de bicicleta…



Antes de começar

Código limpo é independente de linguagem.

Para reforçar isso vou usar exemplos em Delphi, C# e Java e mostrar como estes conceitos são independente de linguagem

Mesmo assim o nosso foco maior será em Delphi, pois o público alvo dessa apresentação tem maior domínio desta linguagem

Antes de começar

1 - Um código ruim pode gerar prejuízos?

Manutenção, Novas funcionalidades, Falência.

2 - O custo de um código confuso?

Produtividade, Complexidade, Bugs. *“Nunca é só um If”*

3 - O grande replanejamento do projeto!

Vamos recomeçar e fazer direito. *“While (true) do”*

3 - Ter atitude. A culpa é minha?

Requisitos? Prazo apertado? Pedidos do gerente? Exigências dos clientes?

4 - Um grande dilema.

Fazer bagunça mais entregar no prazo.

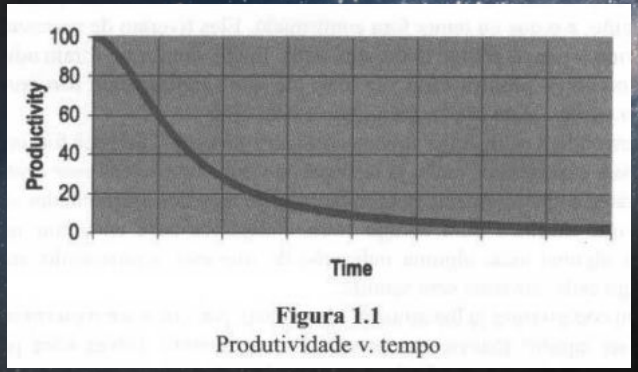


Figura 1.1
Produtividade v. tempo

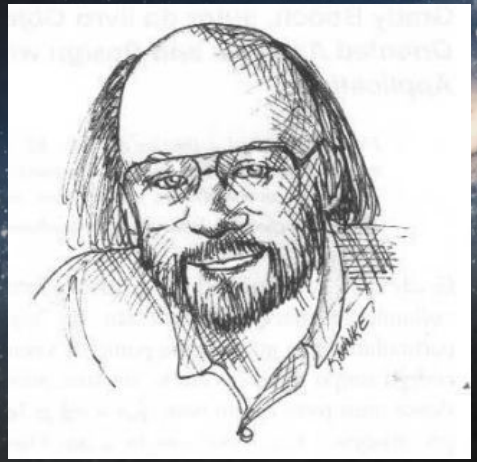
O que é código limpo?

Temos muitas respostas para essa pergunta, vou listar algumas de alguns nomes conhecidos da comunidade…

O que é código limpo?

Bjarne Stroustrup, criador do C++ e autor do livro:
A linguagem de programação C++

Gosto do meu código elegante e eficiente. A lógica deve ser direta para dificultar o encobrimento de bugs, as dependências mínimas para facilitar a manutenção. O tratamento de erros completo de acordo com uma estratégia clara e com desempenho próximo do mais eficiente de modo a não incitar as pessoas a tornarem o código confuso com otimizações sorrateiras. O código limpo faz bem apenas uma coisa.



O que é código limpo?

Grady Booch, autor do livro: Análise e design orientado a objetos com aplicativos

Um código limpo é simples e direto. Ele é tão bem legível quanto uma prosa bem escrita. Ele jamais torna confuso o objetivo do desenvolvedor, em vez disso, ele está repleto de abstrações claras e linhas de controle objetivas.

O que é código limpo?

O “grande” Dave Thomas, fundador da OTI, o pai da estratégia Eclipse

Além de seu criador, um desenvolvedor pode ler e melhorar um código limpo. Ele tem teste de unidade e de aceitação, nomes significativos, ele oferece apenas uma maneira, e não várias, de se fazer uma tarefa, possui poucas dependências, as quais são explicitamente declaradas e oferecem um API mínimo e claro. O código deve ser inteligível já que dependendo da linguagem, nem toda informação necessária pode ser expressa no código em si.

O que é código limpo?

Michael Feathers, autor do livro: Trabalhando efetivamente com código legado

Eu poderia listar todas as qualidades que vejo em um código limpo, mas há uma predominante que leva a todas as outras. Um código limpo sempre parece que foi escrito por alguém que se importava. Não há nada de óbvio no que se pode fazer para torná-lo melhor. Tudo foi pensado pelo autor do código, e se tentar pensar em algumas melhoras, você voltará ao início, ou seja, apreciando o código deixado para você por alguém que se importa bastante com essa tarefa.

O que é código limpo?

Ron Jeffries, autor do livro: Programação extrema instalada e aventuras de programação extrema em c#

Nestes anos recentes, comecei e quase finalizei, com as regras de beck sobre código simples. Em ordem de prioridade, são:

1 - Efetue todos os testes.

2 - Sem duplicação de códigos.

3 - Expressa todas as ideias do projeto que estão no sistema.

4 - Minimiza o número de entidades, como classes, métodos, funções e outras do tipo.

O que é código limpo?

Ward Cunningham, criador do conceito de “WikiWiki”, criador do Fit, co-criador da programação extrema (eXtreme programming). Incentivador dos padrões de projeto. Líder da smalltalk e da OO. Pai de todos aqueles que se importam com código.

Você sabe que está criando um código limpo quando cada rotina que você lê se mostra como o que você esperava. Você pode chamar de código belo quando ele também faz parecer que a linguagem foi feita para o problema.

O que é código limpo?

Resumindo:

01 - Eficiente

02 - Simples

03 - Direto ao ponto

04 - Mínimas dependências

05 - Sem duplicação

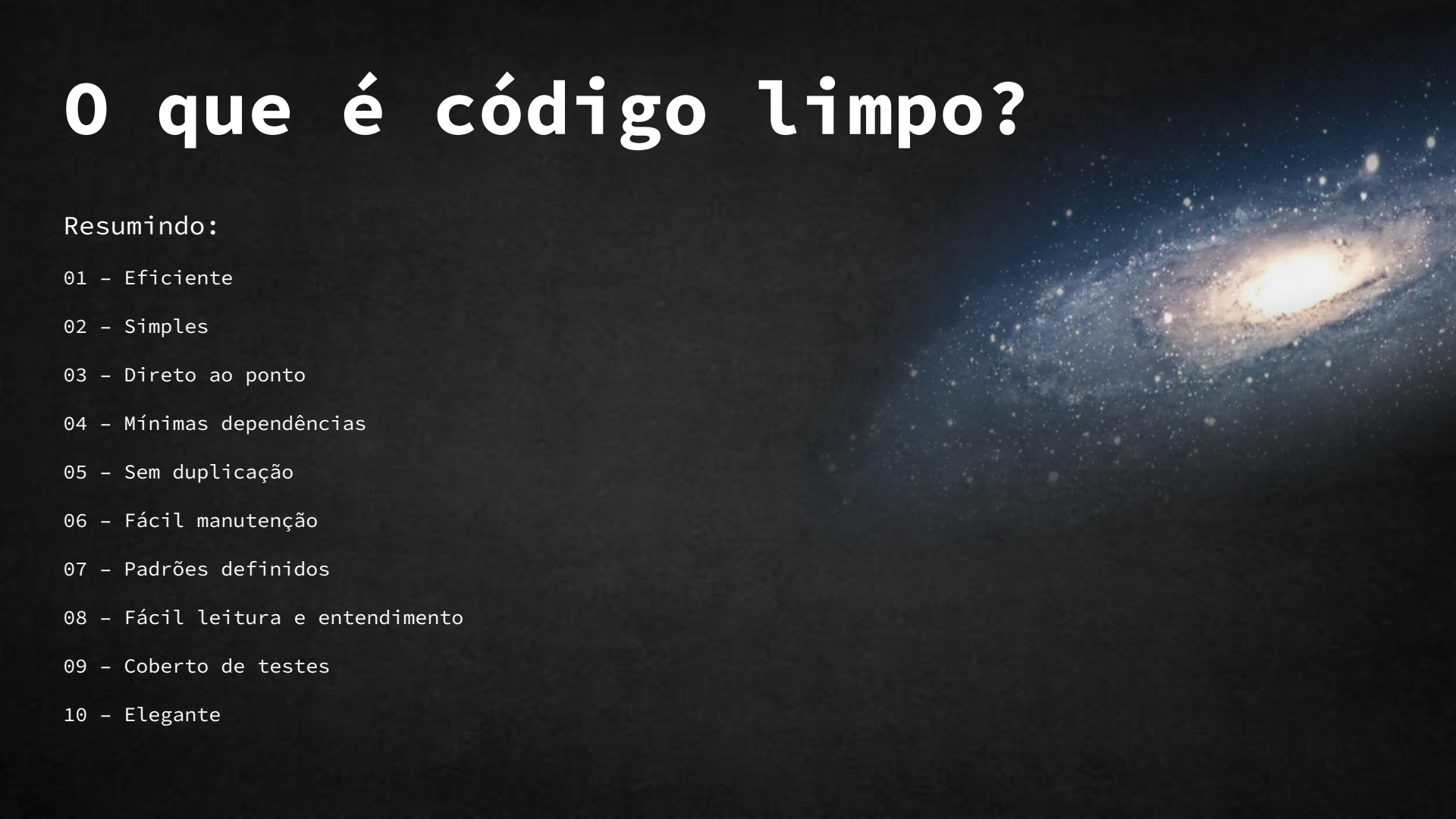
06 - Fácil manutenção

07 - Padrões definidos

08 - Fácil leitura e entendimento

09 - Coberto de testes

10 - Elegante



Nomes significativos

Faz parte do nosso dia a dia dar nomes às coisas, variáveis, funções, classes, parâmetros, pacotes e arquivos.

Essas nomenclaturas influenciam na compreensão do código, e por este motivo temos que escolher bons nomes, que representem bem os seus objetivos.

Vamos ver algumas regras que podem nos ajudar nesta tarefa.

Nomes significativos

Use nomes que revelem seu propósito:

O nome de uma variável, função ou classe deve indicar o porquê ela existe e o que ela faz.

```
var
  d: integer; // Tempo decorrido em dias
```

Nomes significativos

Use nomes que revelem seu propósito:

O nome de uma variável deve indicar o porquê ela existe e o que ela faz.

```
var
  d: integer; // Tempo decorrido em dias
```

```
tempoDecorridoEmDias: integer;
diasDesdeACriacao: integer;
diasDesdeAModificacao: integer;
IdadeDoArquivoEmDias: integer;
```

Nomes significativos

Use nomes que revelem seu propósito:

Escolher nomes que revelem seu propósito pode facilitar bastante o entendimento e a alteração do código. Qual o propósito deste código?

```
function ObterEntao(): TList<TIntegerArray>;
var
  i: integer;
  lista: TList<TIntegerArray>;
begin
  lista := TList<TIntegerArray>.Create();
  for i := 0 to aLista.Count - 1 do
  begin
    if (aLista[i][0] = 4) then
    begin
      lista.Add(aLista[i]);
    end;
  end;

  Result := lista;
end;
```

Nomes significativos

Use nomes que revelem seu propósito:

Escolher nomes que revelem seu propósito pode facilitar bastante o entendimento e a alteração do código. Qual o propósito deste código?

```
function ObterEntao(): TList<TIntegerArray>;
var
  i: integer;
  lista: TList<TIntegerArray>;
begin
  lista := TList<TIntegerArray>.Create();
  for i := 0 to aLista.Count - 1 do
  begin
    if (aLista[i][0] = 4) then
    begin
      lista.Add(aLista[i]);
    end;
  end;

  Result := lista;
end;
```

1. Que tipos de coisas estão em aLista?
2. Qual a importância de um item na posição zero na aLista?
3. Qual a importância do valor 4?
4. Como eu usaria a lista retornada?

Nomes significativos

```
function ObterEntao(): TList<TIntegerArray>;
var
  i: Integer;
  lista: TList<TIntegerArray>;
begin
  lista := TList<TIntegerArray>.Create();
  for i := 0 to aLista.Count - 1 do
  begin
    if (aLista[i][0] = 4) then
    begin
      lista.Add(aLista[i]);
    end;
  end;

  Result := lista;
end;
```

1. Que tipos de coisas estão em aLista?

3. Qual a importância do valor 4?

```
function obterCelulasSinalizadas(): TList<TIntegerArray>;
var
  indice: Integer;
  celulas: TIntegerArray;
  celulasSinalizadas: TList<TIntegerArray>;
begin
  celulasSinalizadas := TList<TIntegerArray>.Create();
  for indice := 0 to Pred(tabuleiroJogo.Count) do
  begin
    celulas := tabuleiroJogo[indice];
    if (celulas[VALOR_STATUS] = SINALIZADO) then
    begin
      celulasSinalizadas.Add(celulas);
    end;
  end;

  Result := celulasSinalizadas;
end;
```

2. Qual a importância de um item na posição zero na aLista?

4. Como eu usaria a lista retornada?

Nomes significativos

```
function obterCelulasSinalizadas(): TList<TIntegerArray>;
var
  indice: Integer;
  celulas: TIntegerArray;
  celulasSinalizadas: TList<TIntegerArray>;
begin
  celulasSinalizadas := TList<TIntegerArray>.Create();
  for indice := 0 to Pred(tabuleiroJogo.Count) do
  begin
    celulas := tabuleiroJogo[indice];
    if (celulas[VALOR_STATUS] = SINALIZADO) then
    begin
      celulasSinalizadas.Add(celulas);
    end;
  end;

  Result := celulasSinalizadas;
end;
```

```
function obterListaCelulasSinalizadas(): TList<TCelulas>;
var
  indice: Integer;
  celulas: TCelulas;
  celulasSinalizadas: TList<TCelulas>;
begin
  celulasSinalizadas := TList<TCelulas>.Create();
  for indice := 0 to Pred(tabuleiroJogo.Count) do
  begin
    celulas := tabuleiroCampoMinado[indice];
    if (celulas.EstaSinalizada) then
    begin
      celulasSinalizadas.Add(celulas);
    end;
  end;

  Result := celulasSinalizadas;
end;
```

1. Que tipos de coisas estão em aLista?
2. Qual a importância de um item na posição zero na aLista?
3. Qual a importância do valor 4?
4. Como eu usaria a lista retornada?

Nomes significativos

Evite informações erradas:

Devemos evitar passar dicas falsas que confundam o sentido do código.

Palavras com significados podem ser diferente dos que queremos

```
listaDeContas: array of TConta;
```

Nomes significativos

Evite informações erradas:

Devemos evitar passar dicas falsas que confundam o sentido do código.

Palavras com significados podem ser diferente dos que queremos

```
listaDeContas: array of TConta;
```

```
listaDeContas: TList<TConta>;
```

Nomes significativos

Evite informações erradas:

Devemos evitar passar dicas falsas que confundam o sentido do código.

Palavras com significados podem ser diferente dos que queremos.

```
listaDeContas: array of TConta;
```

```
listaDeContas: TList<TConta>;
```

```
arrayDeContas01: array of TConta;
```

Nomes significativos

Faça distinções significativas:

```
procedure CopiarCaracteres(a1: array of Char; a2: array of Char);
var
  i: Integer;
begin
  for i := 0 to Length(a1) - 1 do
  begin
    a2[i] := a1[i];
  end;
end;
```

Nomes significativos

Faça distinções significativas:

```
procedure CopiarCaracteres(a1: array of Char; a2: array of Char);
var
  i: Integer;
begin
  for i := 0 to Length(a1) - 1 do
  begin
    a2[i] := a1[i];
  end;
end;
```

```
procedure CopiarCaracteres(origem: array of Char; destino: array of Char);
var
  indice: Integer;
begin
  for indice := 0 to Pred(Length(origem)) do
  begin
    destino[indice] := origem[indice];
  end;
end;
```

Nomes significativos

Faça distinções significativas:

```
procedure ExemploGrupos;
begin
  TGrupo.ObterGruposAtivo;
  TGrupo.ObterGruposAtivos;
  TGrupo.ObterGruposAtivosInfo;
end;
```

Nomes significativos

Use nomes pronunciáveis:

```
type
  TDtaRcrd102 = class
  public
    fPssStr: string;
    function ObterYmdHms: TDateTime;
  end;
```

Nomes significativos

Use nomes pronunciáveis:

```
type
  TDtaRcrd102 = class
  public
    fPssStr: string;
    function ObterYmdHms: TDateTime;
  end;
```

```
type
  TFuncoesData = class
  public
    fPesquisa: string;
    function ObterDataHora: TDateTime;
  end;
```

Nomes significativos

Use nomes passíveis de busca:

```
procedure Exemplo08;
var
  s: double;
  i: Integer;
  j: Integer;
  t: array of Integer;
begin
  for i := 0 to 34 do
  begin
    s := s + (t[j] * 4) / 5;
  end;
end;
```

Nomes significativos

Use nomes passíveis de busca:

```
procedure Exemplo08;
var
  soma: double;
  indice: Integer;
  segundoIndice: Integer;
  diasDeTarefasReais: Integer;
  semanasDeTarefasReais: double;
  estimativaDeTarefa: array of Integer;
const
  NUMERO_DE_TAREFAS = 34;
  DIAS_REAIS_POR_DIA_IDEAL = 4;
  DIAS_DE_TRABALHO_POR_SEMANA = 5;
begin
  for indice := 0 to NUMERO_DE_TAREFAS do
  begin
    diasDeTarefasReais := estimativaDeTarefa[segundoIndice] * DIAS_REAIS_POR_DIA_IDEAL;
    semanasDeTarefasReais := diasDeTarefasReais / DIAS_DE_TRABALHO_POR_SEMANA;
    soma := soma + semanasDeTarefasReais;
  end;
end;
```

Nomes significativos

Evite codificações:

Não codifique informações do escopo ou tipos em nomes no código, isso simplesmente adiciona uma tarefa extra de decifração ao nosso trabalho.

Os nomes codificados são raramente pronunciáveis.

```
type
  TDtaRcrd102 = class
  public
    fPssStr: string;
    function ObterYmdHms: TDateTime;
  end;
```

Nomes significativos

A notação húngara:

A Notação húngara, criada por Charles Simonyi, visa a facilitar o reconhecimento do tipo de variável num programa. O nome foi dado a partir de uma brincadeira comum entre os primeiros a conhecer a notação que a achavam estranha, fazendo o seguinte comentário: "É tão estranho que até parece húngaro".

Nomes significativos

A tabela abaixo indica os tipos de indicadores mais utilizados na Notação húngara:

Nome	Descrição
s	String
sz	Aponta o primeiro caracter da terminação zero da string
st	Ponteiro da string, o primeiro byte é contado dos caracteres
h	handle (título)
msg	Message
fn	function (usada com pointer)
c	char (8 bits)
by	unsigned char (byte or uchar - 8 bits)
n	Int
b	Boolean (verdadeiro ou falso)
f	Flag (boolean, logical).
u	integer
w	Word
ch	Char, com texto ASCII
l	long int (32 bits)
dw	unsigned long int (dword - 32 bits)

Nomes significativos

Antigamente quando trabalhávamos com linguagens com limite de tamanho para nomes, violávamos essa regra quando necessário. O Fortran forçava codificações ao tornar a primeira letra uma indicação para o tipo. Versões do BASIC só permitiam uma letra mais um dígito. A notação húngara inovou essas limitações. Havia compiladores que não verificavam os tipos e os programadores precisavam de ajuda para lembrar dos tipos.

Esse cenário mudou, hoje não temos essas mesmas limitações em linguagens mais modernas, dessa forma a notação húngara e outras formas de convenção de tipos são basicamente obstáculos.

Eles dificultam a alteração do nome ou do tipo de uma variável, função ou classe, dificultam a leitura do código e criam a possibilidade de que o sistema de codificação induza o leitor ao erro.

```
var
  TelefoneInteger: string;
  TelefoneString: Integer;
```

Nomes significativos

Nomes de classes:

Classes e objetos devem ter nomes com substantivos, como Cliente, Pagina e Endereço.

Nomes de classes não devem ser verbos.

```
type
  TCliente = class
  end;

  TPagina = class
  end;

  TEndereco = class
  end;
```

Nomes significativos

Nomes de métodos (functions e procedures):

Os nomes de métodos devem ter verbos, como os exemplos:

```
procedure EnviarPagamento;
begin
end;

procedure Salvar;
begin
end;

procedure GerarFatura;
begin
end;
```

Nomes significativos

Nomes de métodos (functions e procedures):

Quando construtores estiverem sobrecarregados, devemos usar métodos factory estáticos com nomes que descrevem os parâmetros.

```
type
  TComplex = class
  public
    constructor Create(numero: double);
    class function ApartirDeNumeroReal(numero: Real): TComplex;
  end;
```

Para obrigar o uso dos novos métodos podemos tornar os construtores privados.

Nomes significativos

Adicione contextos significativos:

Existem poucos nomes que são significativos por si só, por este motivo precisamos usar nomes que façam parte do contexto para o leitor do código.

```
var
  firstName, lastName, street, city, state, zipcode: string;
  // uma melhor solução
  addrFirstName, addrLastName, adrState: string;

  // uma melhor solução
type
  TAddres = class
  end;
```

Nomes significativos

Não adicione contextos desnecessários:

Geralmente nomes curtos são melhores, contanto que sejam claros.
Não adicione mais contexto a um nome, mais do que é necessário.

```
type
  TEndereco = class
  public
    function ObterRuaEndereco: string;
    function ObterCidadeEndereco: string;
  end;
```

Nomes significativos

*“**Qualquer um** consegue escrever código que um computador entende.*
__Bons programadores__ escrevem código que humanos entendem”

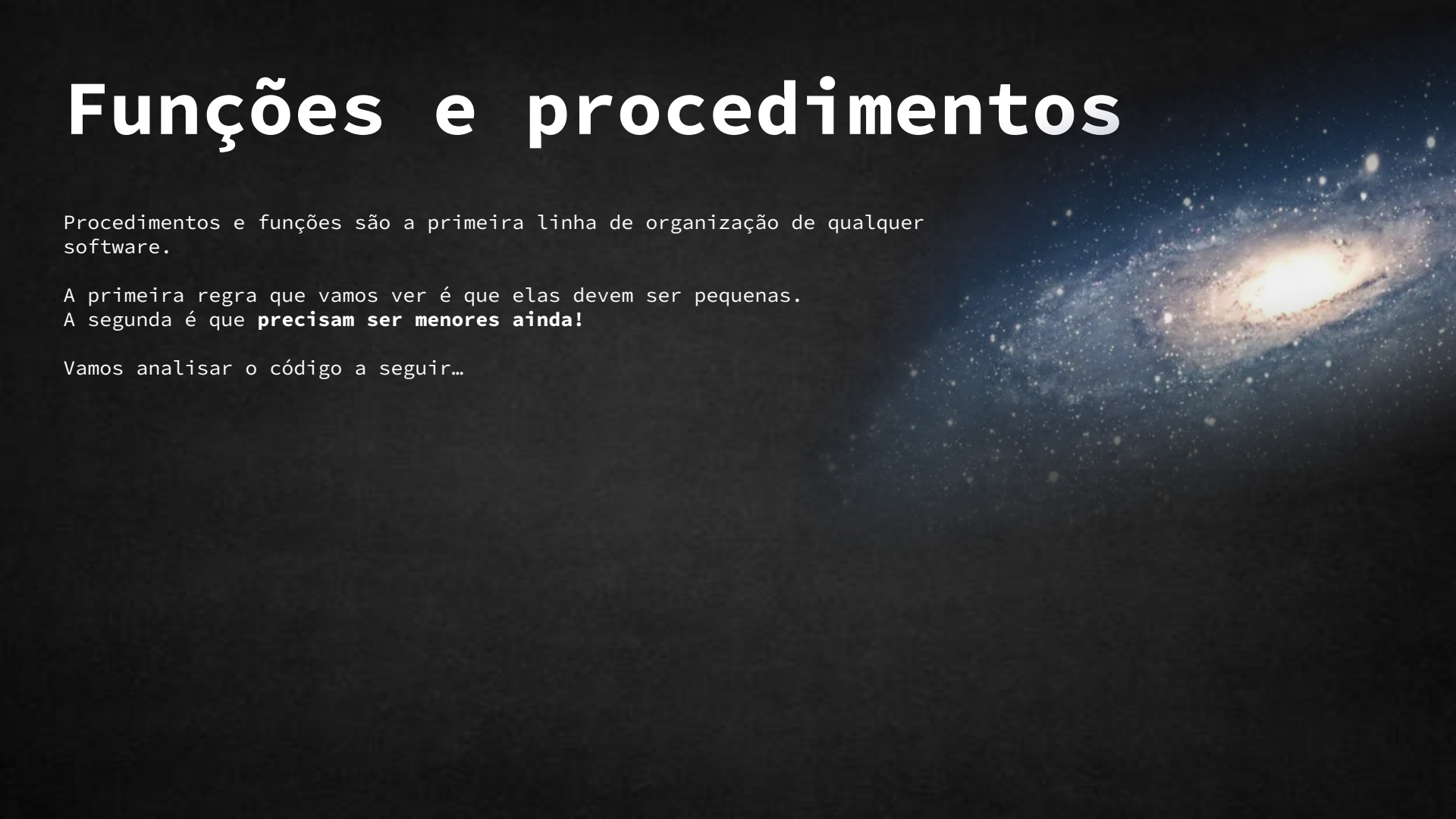
Martin Fowler

Funções e procedimentos

Procedimentos e funções são a primeira linha de organização de qualquer software.

A primeira regra que vamos ver é que elas devem ser pequenas.
A segunda é que **precisam ser menores ainda!**

Vamos analisar o código a seguir…



Listagem 3-1
HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
              SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
```

Listagem 3-1 (continuação)
HtmlUtil.java (FitNesse 20070619)

```
    if (includeSuiteSetup) {
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
              SuiteResponder.SUITE_TEARDOWN_NAME,
              wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

Listagem 3-2
HtmlUtil.java (refatorado)

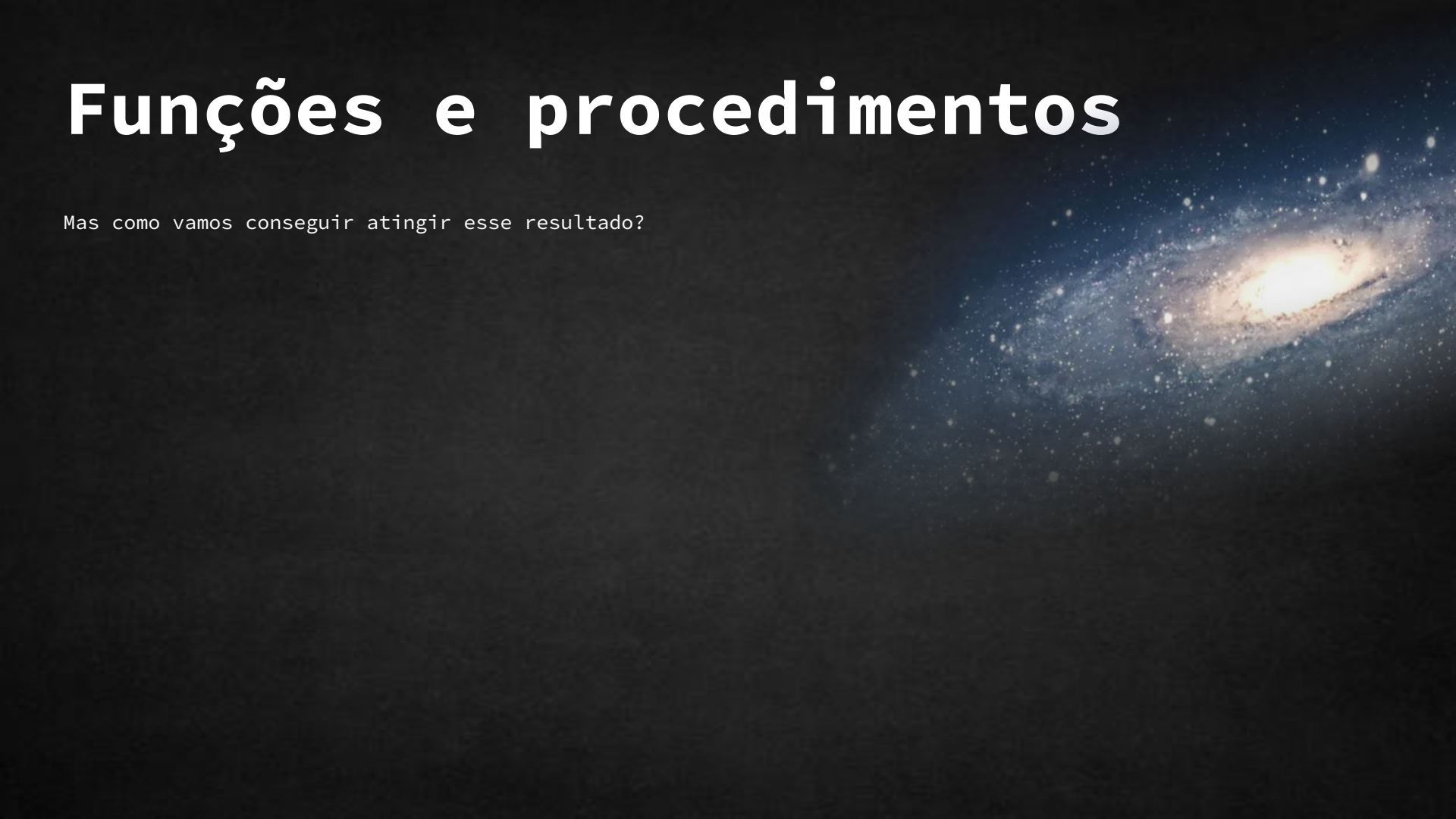
```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }

  return pageData.getHtml();
}
```

Listagem 3-3
HtmlUtil.java (refatorado novamente)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Funções e procedimentos

Mas como vamos conseguir atingir esse resultado?



Funções e procedimentos

Vamos analisar alguns pontos:

Os blocos if, else, while e outros devem ter apenas uma linha.

O nível de indentação de uma função deve ser de, no máximo, um ou dois níveis.

As funções devem fazer uma coisa, devem fazê-la bem.
Devem fazer apenas ela.

```
procedure Salvar;
begin
  if (JaEstaCadastrado) then
    Incluir
  else
    Editar
end;
```

Como saber se uma função faz apenas uma coisa?

Algumas perguntas podem nos ajudar:

A função faz o que o nome dela indica?

A função pode ser quebrada em seções?

Posso extrair uma nova função de dentro desta?
(O nome da nova função reafirma a anterior)

Funções e procedimentos

```
procedure TfrmCdBasicLight.RestoreGridLayoutFromDisk;
begin
  {mais código aqui...}
  // procura a seção de layout no INI
  LayoutSection := SectionName + '.LastColumnSet';
  if not IniFileHandle.SectionExists(LayoutSection) then
         Exit;
  ColumnsOrders := TStringList.Create;
  // para cada coluna válida da grid adiciona sua ordem à coleção
  for I := 0 to dgrData.Columns.Count - 1 do
  begin
      {mais código aqui}
  end;
  // organiza a lista de colunas pelo seu indice
  ColumnsOrders.Sort;
  for I := 0 to ColumnsOrders.Count - 1 do
  begin
         GridColumn := TColumn(ColumnsOrders.Objects[I]);
         if StrToInt(ColumnsOrders[I]) < dgrData.Columns.Count then
         GridColumn.Index := StrToInt(ColumnsOrders[I]);
  end;
  {Configurar a grid e a query de acordo com o layout salvo pelo usuário}
  dgrData.SortColumnInfo.SortedFieldName := IniFileHandle.ReadString(LayoutSection,
         dgrData.Name + '.SortedFieldName', '');
  dgrData.SortColumnInfo.LastSortedFieldName := IniFileHandle.ReadString(LayoutSection,
         dgrData.Name + '.LastSortedFieldName', '');
  {mais código aqui...}
end;
```

Funções e procedimentos

O código deve ser lido como uma narrativa, de cima para baixo.

Queremos que o código seja lido de cima para baixo, como uma narrativa.

Queremos que cada função seja seguida pelas outras do próximo nível de modo que possamos ler o programa descendo um nível de cada vez, conforme percorremos a lista de funções.

Chamamos isso de *Regra Decrescente*.

Funções e procedimentos

1 - Para incluir o Setup e TearDown, incluímos Setup, o conteúdo da página de teste e os TearDown.

2 - Para incluir os Setup, incluímos o Setup do Suite, se for uma Suite, e o Setup regular.

3 - Para incluir o Setup do conjunto, pesquisamos na hierarquia pai a página "SuiteSetUp" e adicionamos uma instrução de inclusão com o caminho desta página.

4 - Para pesquisar os pais ... Vários níveis de abstração, normalmente gera confusão.

```
procedure IncludeSetupsAndTeardownsPage;
begin
  IncludeSetups;
  IncludeTestPageContent;
  IncludeTeardowns;
end;

procedure IncludeSetups;
begin
  if (IsSuite) then
    IncludeSuiteSetup;
  IncludeRegularSetup;
end;

procedure IncludeSuiteSetup;
var
  parentPage: TPage;
begin
  parentPage := FindParentSuitePage;
  // add include statement with the path of the parentPage
end;
```

Funções e procedimentos

Estrutura Switch (case)

```
function CalculatePay(e: TEmployee): TMoney;
begin
  case e.Kind of
    COMMISSIONED:
      Result := CalculateCommissionedPay(e);
    HOURLY:
      Result := CalculateHourlyPay(e);
    SALARIED:
      Result := CalculateSalariedPay(e);
  end;
end;
```

Esta função tem vários problemas:
Primeiro, ela é grande, é quando se adiciona novos tipos ela crescerá mais ainda.
Segundo, obviamente ela faz mais de uma coisa.
Terceiro, ela viola o princípio da responsabilidade única por haver mais de um motivo para alterá-la.
Quarto, ela viola o princípio de aberto fechado, pois precisa ser modificada sempre que novos tipos forem adicionados. Mas, provavelmente, o pior problema com essa função é a quantidade ilimitada de outras funções que terão a mesma estrutura.

Por exemplo, poderíamos ter:

```
function IsPayday(e: TEmployee; date: TDateTime): TMoney;
begin
end;

procedure DeliverPay(e: TEmployee; pay: TMoney);
begin
end;
```

Funções e procedimentos

Estrutura Switch (case)

A regra geral para estruturas switch (case) é que são aceitáveis se aparecerem apenas uma vez, como para a criação de objetos polifórmicos, e se estiverem escondidas atrás de uma relação de herança de modo que o resto do sistema não possa enxergá-la. É claro que cada caso é um caso e haverá vezes que não vamos respeitar uma ou mais partes dessa regra.

Funções e procedimentos

Usar nomes descritivos

Quanto menor e mais centralizada for a função, mais fácil será pensar em um nome que a descreva bem.

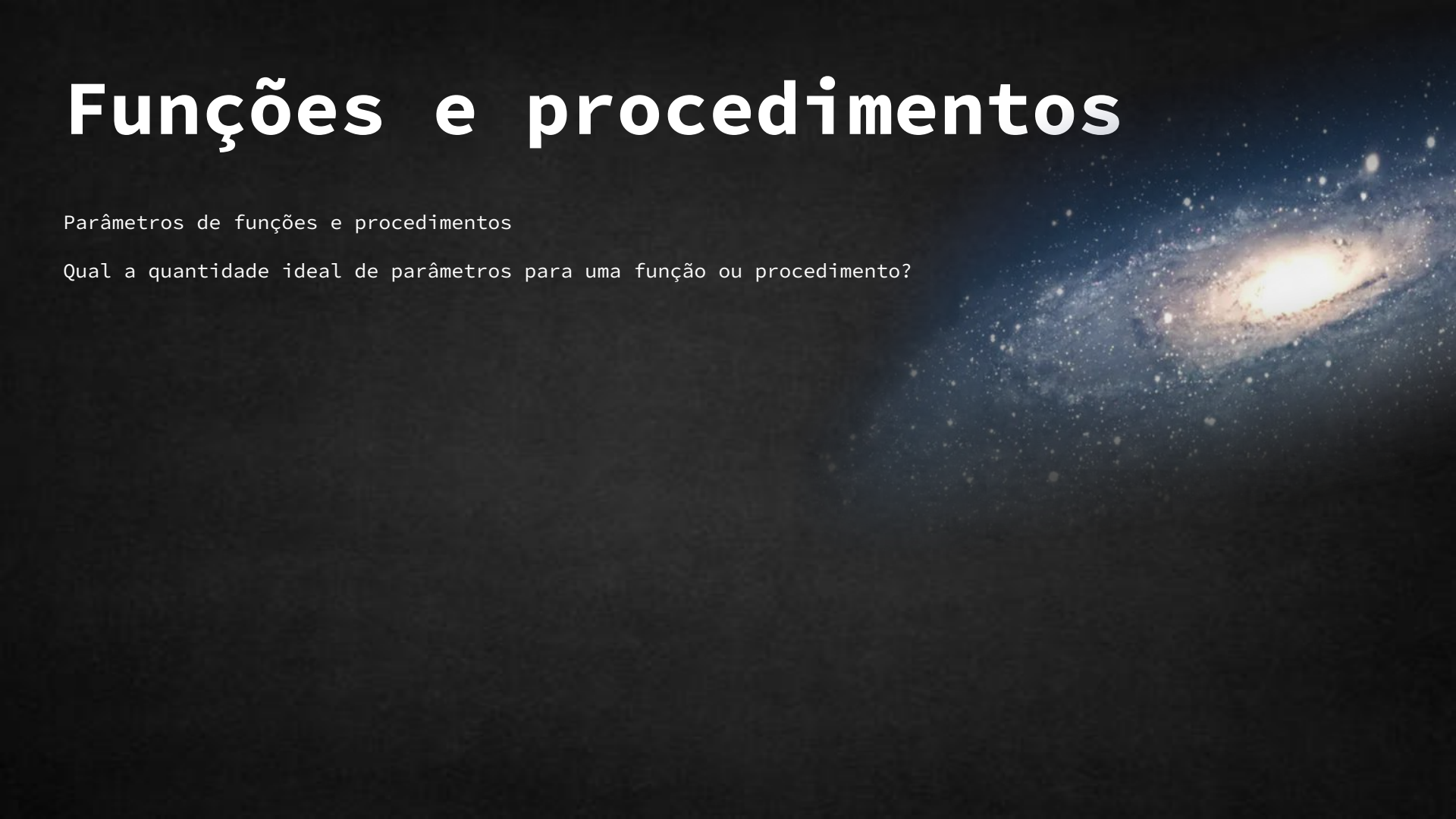
Não tenha medo de criar nomes extensos, porque eles são melhores do que um pequeno e enigmático nome.

```
3 referências
public IQueryable<EmpresaDto> ObterEmpresasPodemEmitirDocumentosClientesVinculadosUsuario(Guid idLicenciada, IList<string> documentos)
    => _empresaRepository.ObterEmpresasPodemEmitirDocumentosClientesVinculadosUsuario(idLicenciada, documentos)
    .ProjectTo<EmpresaDto>(_mapper.ConfigurationProvider);
```

Funções e procedimentos

Parâmetros de funções e procedimentos

Qual a quantidade ideal de parâmetros para uma função ou procedimento?

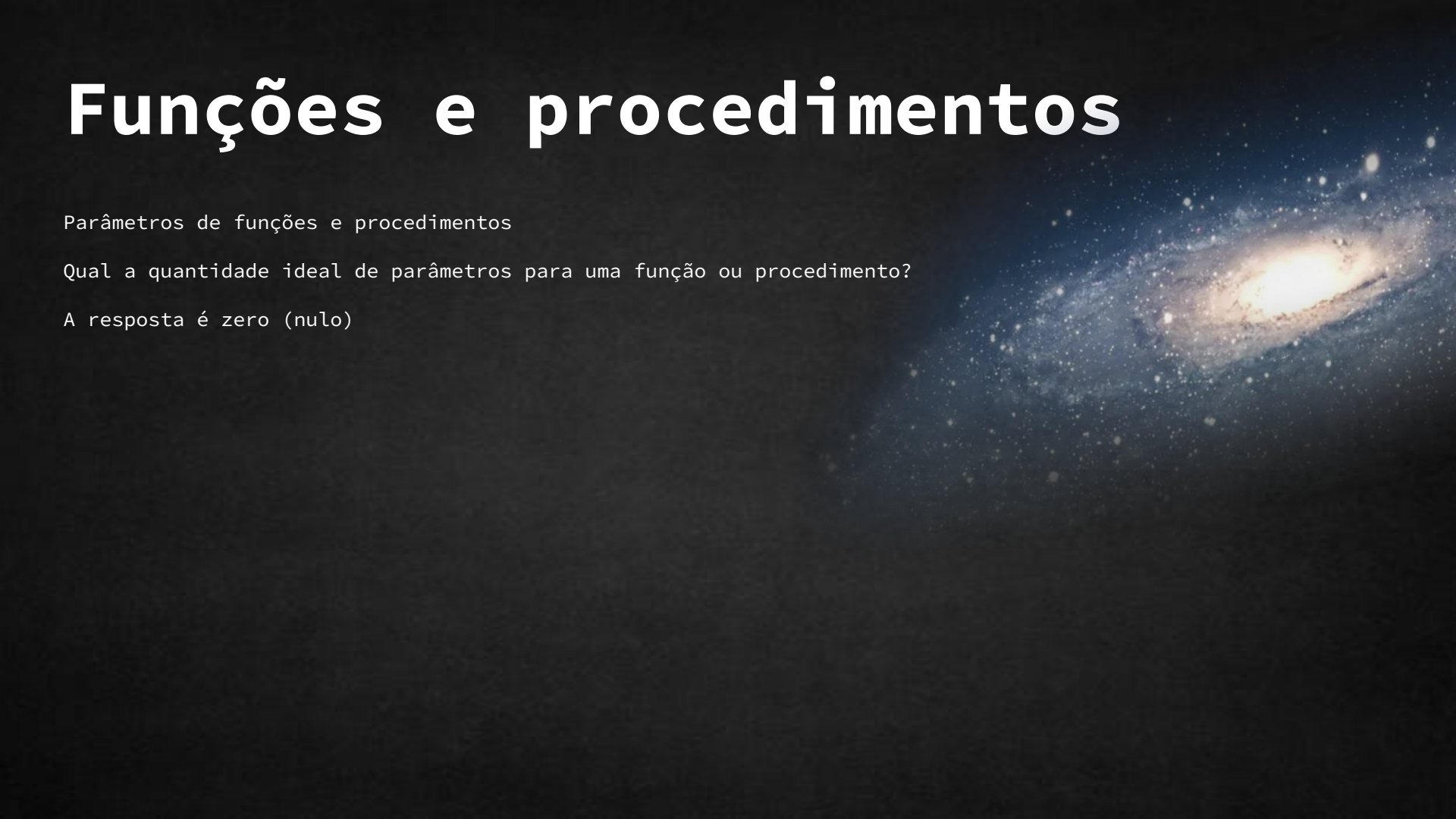


Funções e procedimentos

Parâmetros de funções e procedimentos

Qual a quantidade ideal de parâmetros para uma função ou procedimento?

A resposta é zero (nulo)



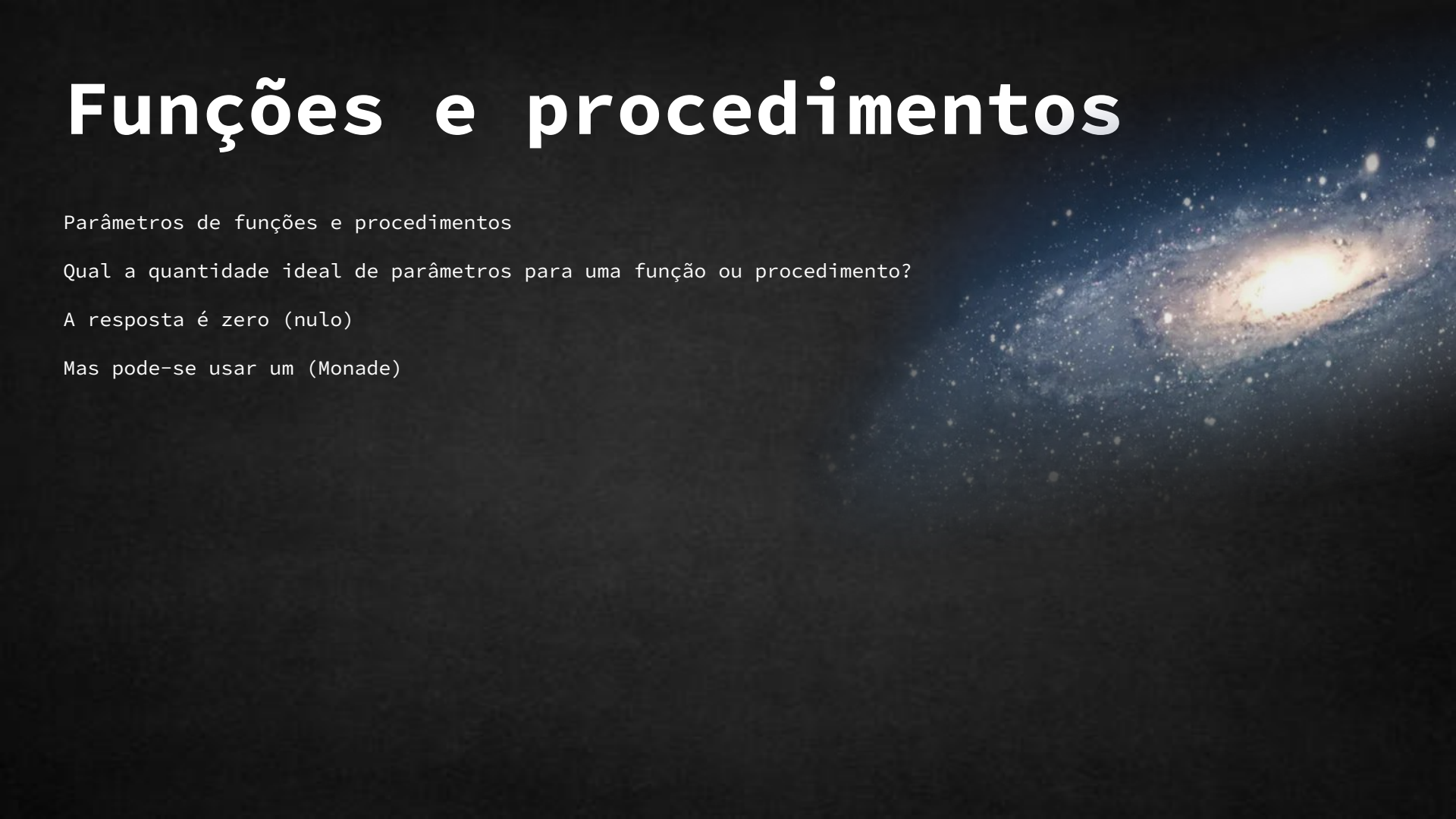
Funções e procedimentos

Parâmetros de funções e procedimentos

Qual a quantidade ideal de parâmetros para uma função ou procedimento?

A resposta é zero (nulo)

Mas pode-se usar um (Monade)



Funções e procedimentos

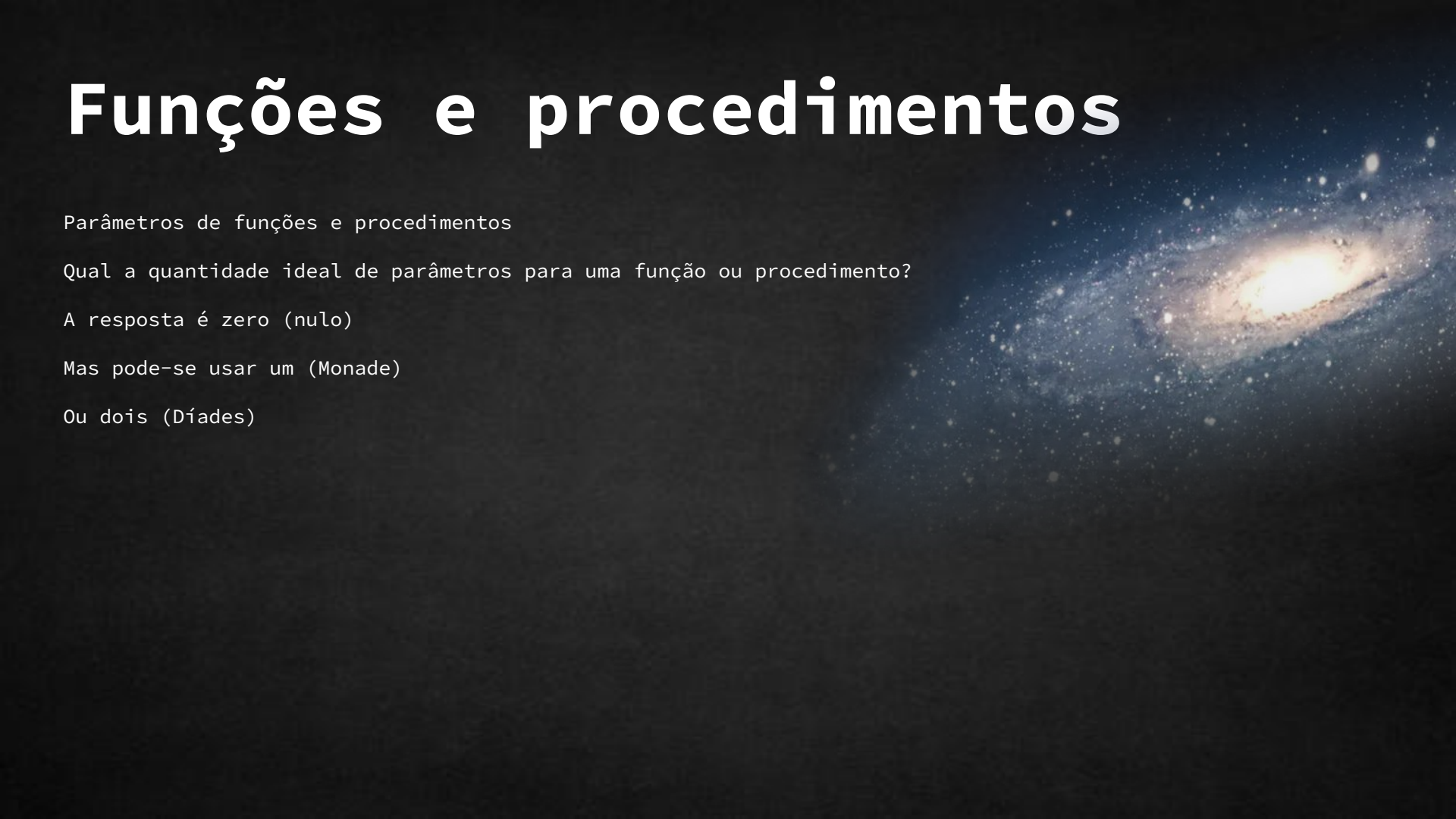
Parâmetros de funções e procedimentos

Qual a quantidade ideal de parâmetros para uma função ou procedimento?

A resposta é zero (nulo)

Mas pode-se usar um (Monade)

Ou dois (Díades)



Funções e procedimentos

Parâmetros de funções e procedimentos

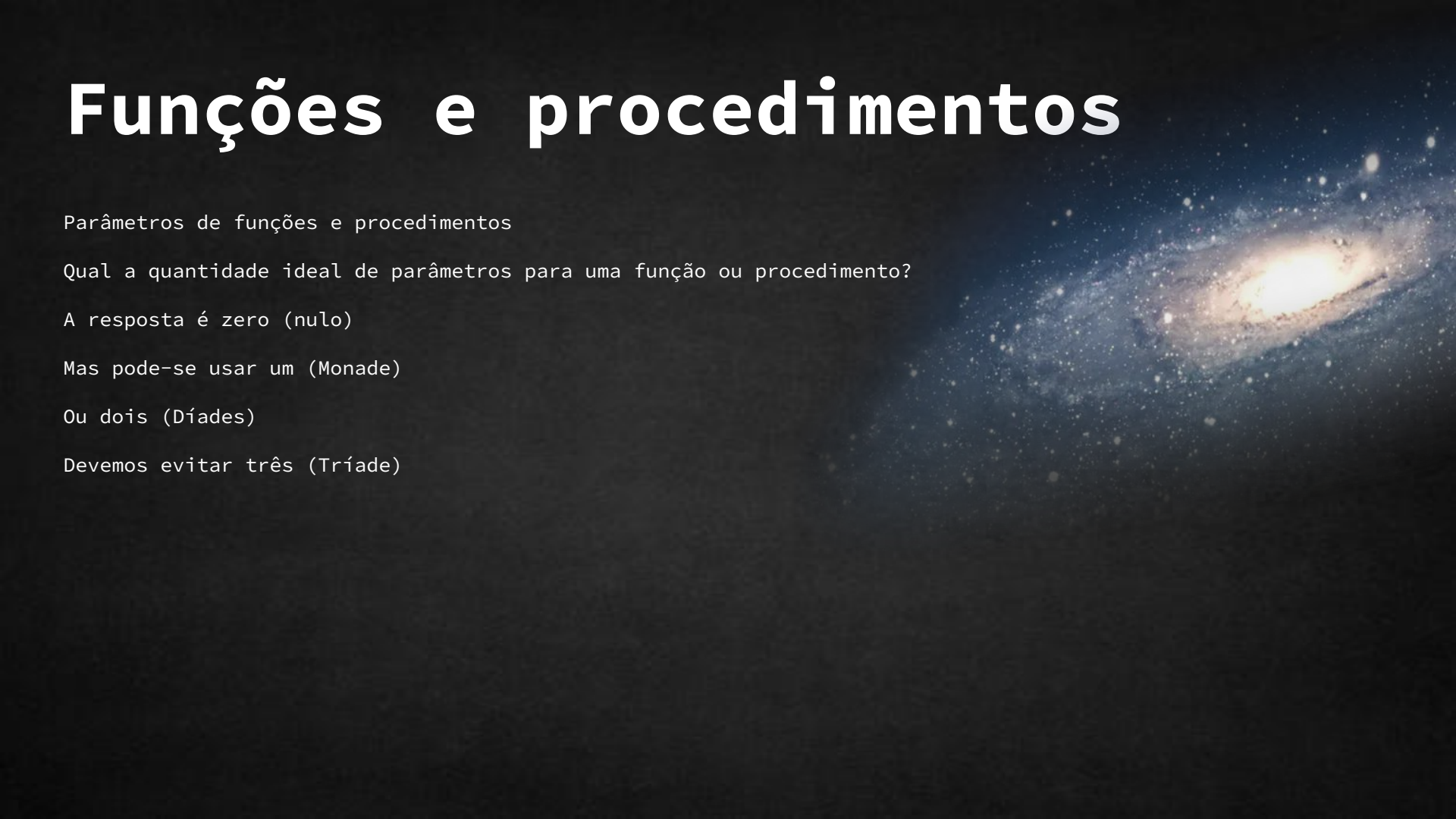
Qual a quantidade ideal de parâmetros para uma função ou procedimento?

A resposta é zero (nulo)

Mas pode-se usar um (Monade)

Ou dois (Díades)

Devemos evitar três (Tríade)



Funções e procedimentos

Parâmetros de funções e procedimentos

Qual a quantidade ideal de parâmetros para uma função ou procedimento?

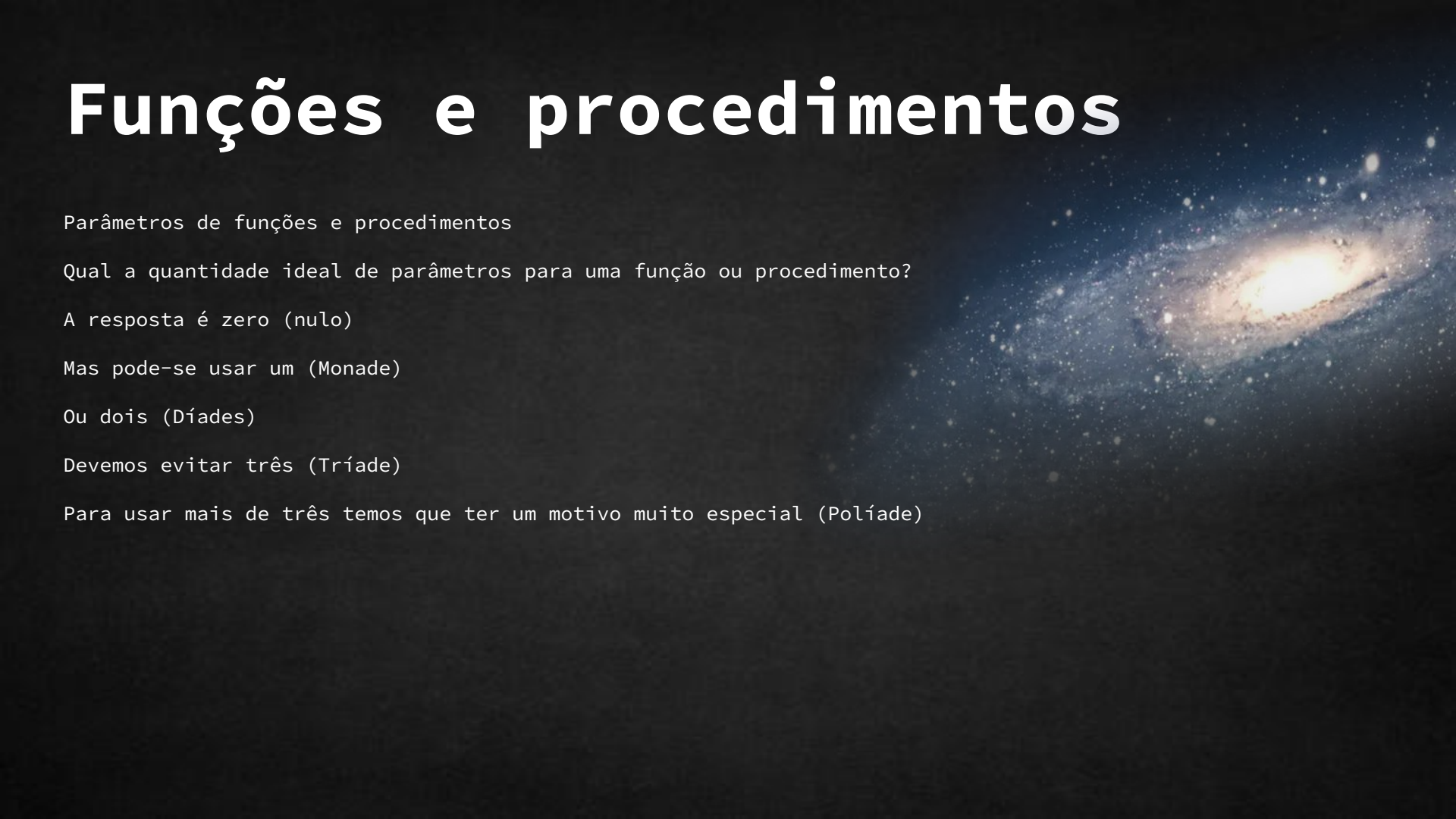
A resposta é zero (nulo)

Mas pode-se usar um (Monade)

Ou dois (Díades)

Devemos evitar três (Tríade)

Para usar mais de três temos que ter um motivo muito especial (Políade)



Funções e procedimentos

Parâmetros de funções e procedimentos

Qual a quantidade ideal de parâmetros para uma função ou procedimento?

A resposta é zero (nulo)

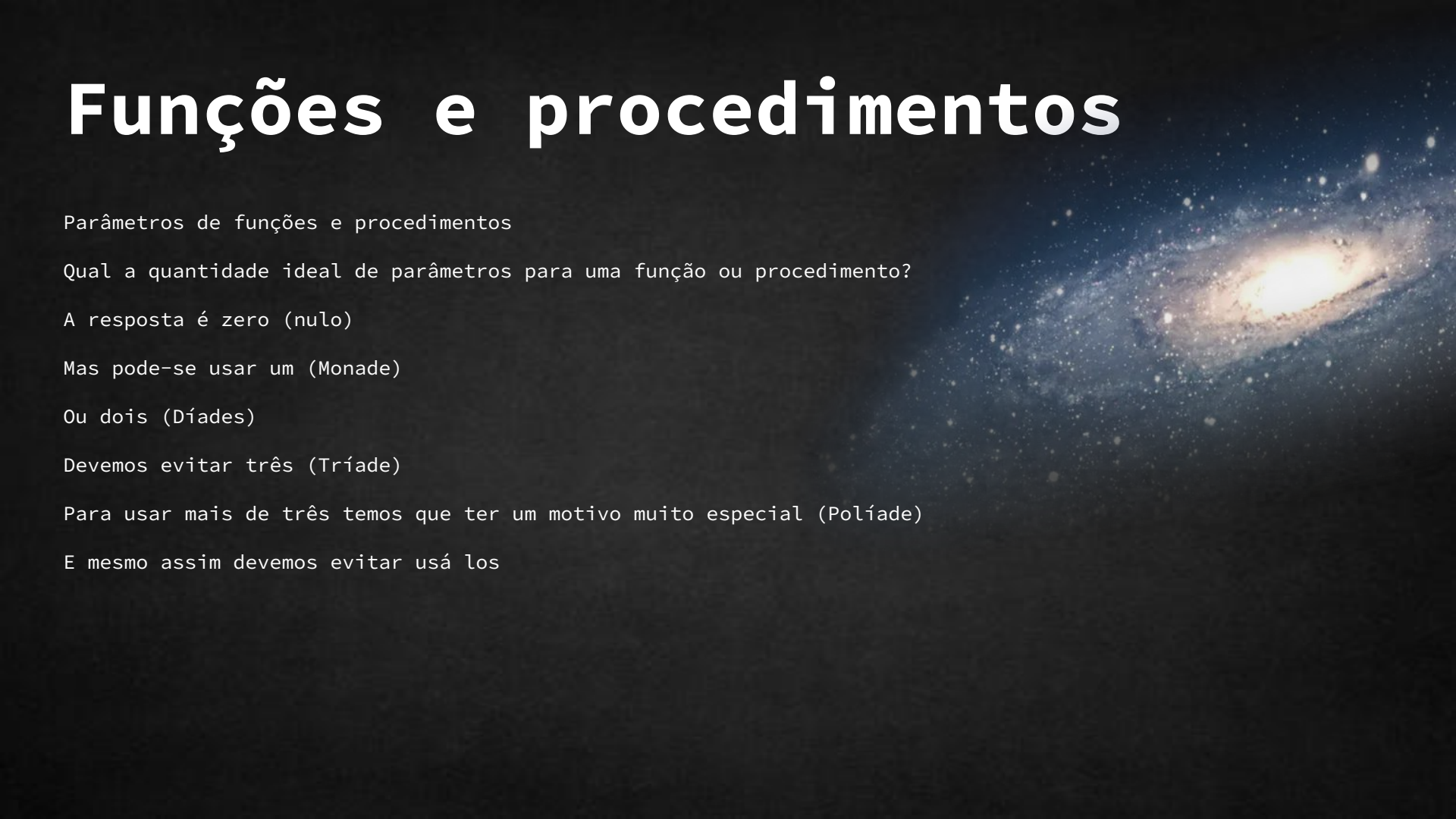
Mas pode-se usar um (Monade)

Ou dois (Díades)

Devemos evitar três (Tríade)

Para usar mais de três temos que ter um motivo muito especial (Políade)

E mesmo assim devemos evitar usá los



Funções e procedimentos

Parâmetros de funções e procedimentos

Qual a quantidade ideal de parâmetros para uma função ou procedimento?

A resposta é zero (nulo)

Mas pode-se usar um (Monade)

Ou dois (Díades)

Devemos evitar três (Tríade)

Para usar mais de três temos que ter um motivo muito especial (Políade)

E mesmo assim devemos evitar usá los

```
function AtualizaEnderecoCliente(const sCgcCpf, sEndereco, sBairro, sCep: String; const sCodCidade: String = '';
                                 const sDDDFone: String = ''; const sFone: String = ''; const sCgf: String = '';
                                 iTipoEnd: Integer = 0; const sTabela: String = 'CLI'; const sCondicaoContribuinteCliente: string ='';
                                 const sComplementoEndereco: string = ''; const sNumeroEndereco: string = ''): Boolean;
```

Funções e procedimentos

Parâmetros de funções e procedimentos

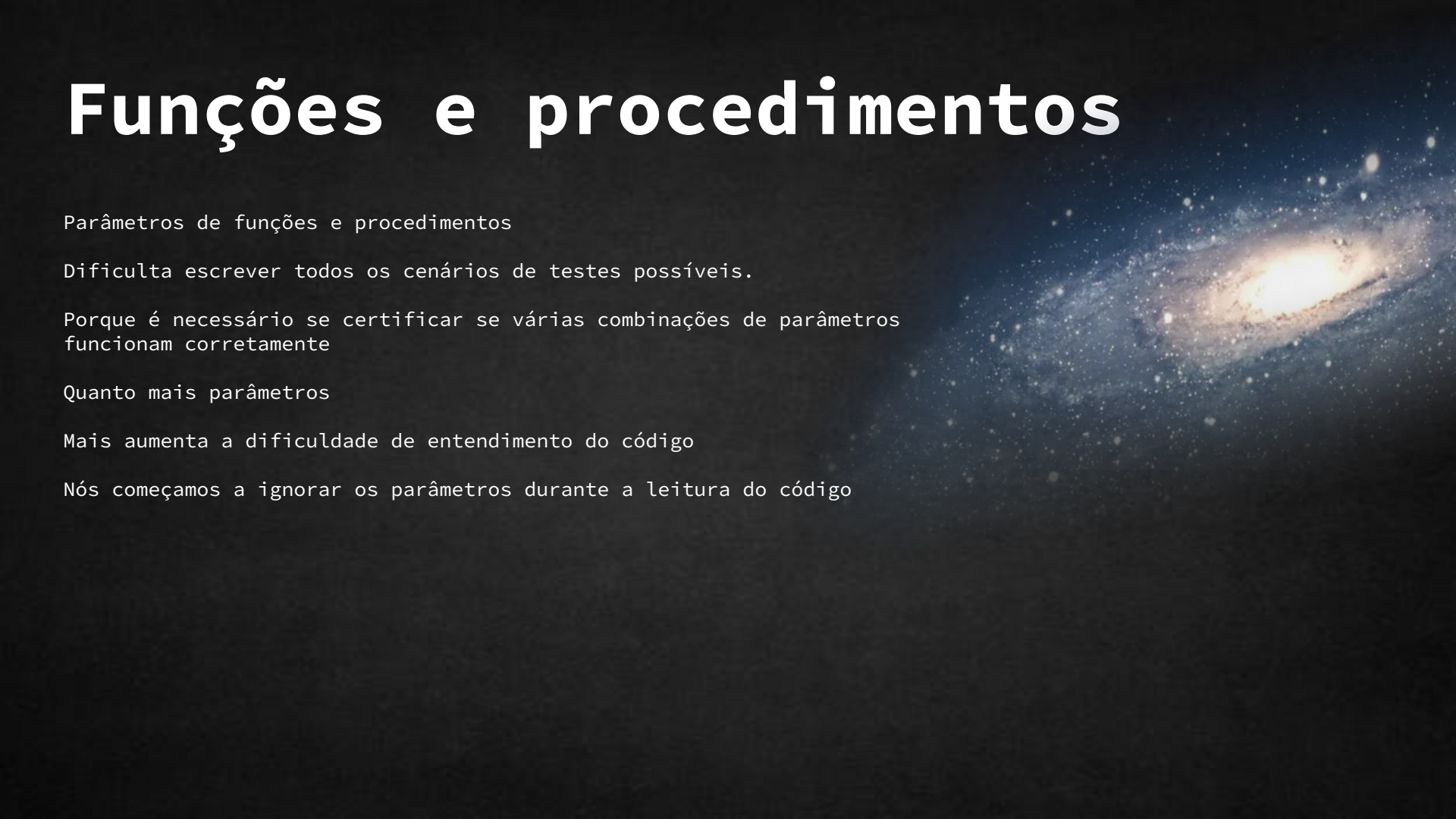
Dificulta escrever todos os cenários de testes possíveis.

Porque é necessário se certificar se várias combinações de parâmetros funcionam corretamente

Quanto mais parâmetros

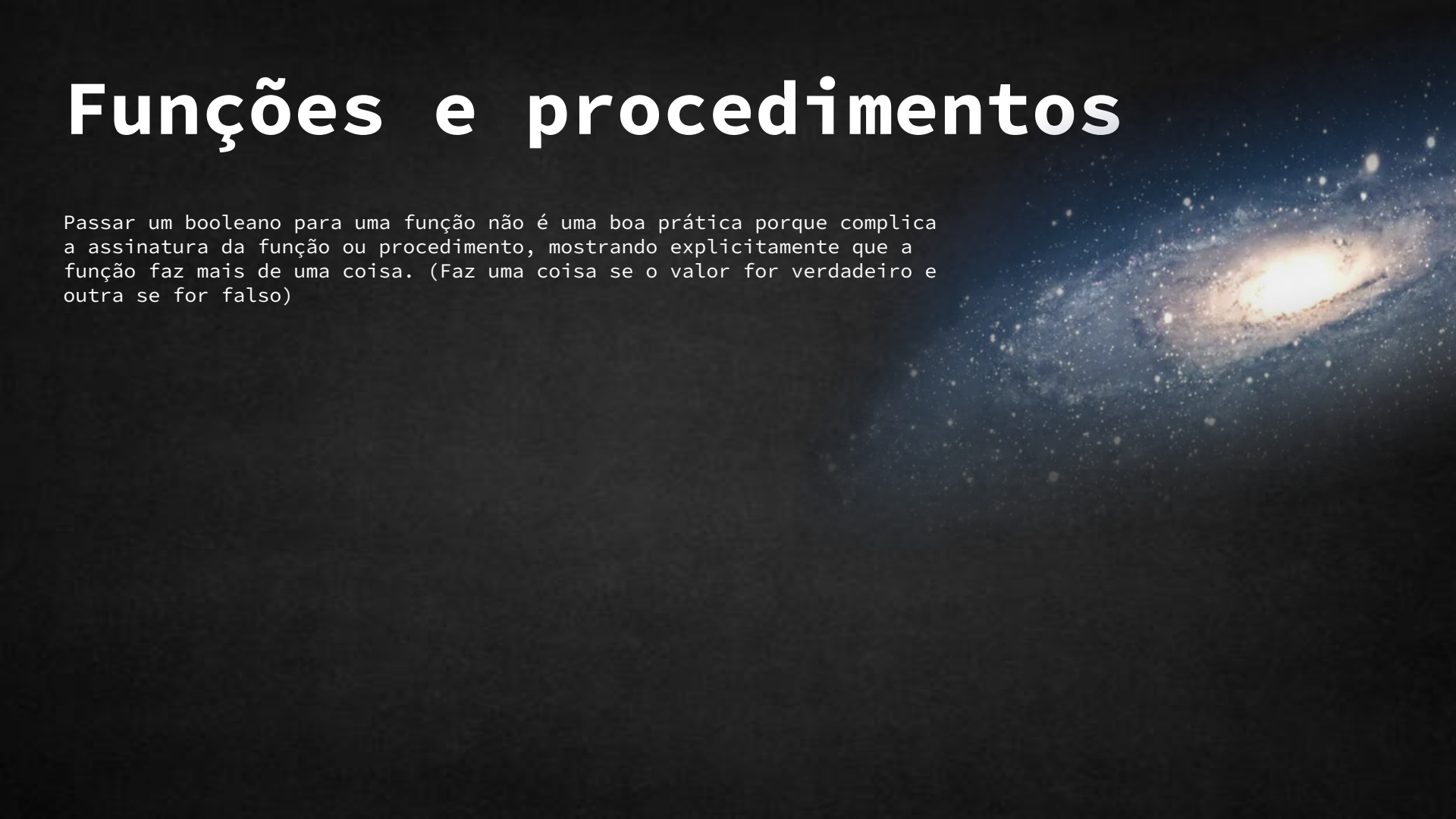
Mais aumenta a dificuldade de entendimento do código

Nós começamos a ignorar os parâmetros durante a leitura do código



Funções e procedimentos

Passar um booleano para uma função não é uma boa prática porque complica a assinatura da função ou procedimento, mostrando explicitamente que a função faz mais de uma coisa. (Faz uma coisa se o valor for verdadeiro e outra se for falso)



Funções e procedimentos

Passar um booleano para uma função não é uma boa prática porque complica a assinatura da função ou procedimento, mostrando explicitamente que a função faz mais de uma coisa. (Faz uma coisa se o valor for verdadeiro e outra se for falso)

```
procedure HabilitarEdicao(value: Boolean);
begin
  edtCodigo := value;
  edtNome := (not value);
end;
```

Funções e procedimentos

Passar um booleano para uma função não é uma boa prática porque complica a assinatura da função ou procedimento, mostrando explicitamente que a função faz mais de uma coisa. (Faz uma coisa se o valor for verdadeiro e outra se for falso)

```
procedure HabilitarEdicao();
begin
  edtCodigo := True;
  edtNome := False;
end;

procedure HabilitarVisualizacao();
begin
  edtCodigo := False;
  edtNome := True;
end;
```

Funções e procedimentos

Funções Díades

Funções com dois parâmetros não são tão ruins, mesmo que possam confundir muito a sua utilização na hora de usá-las, como por exemplo:

```
procedure assertEquals(expected: string; actual: string);
begin
end;
```

Funções e procedimentos

Funções Díades

São funções que teremos que usar em algum momento, por mais que tenhamos que pagar um preço por isso, porém podemos tirar proveito dos mecanismos disponíveis para transformá-las em mônades (um parâmetro), por exemplo:

```
type
  TStringListHelper = class helper for TStringList
    procedure WriteField(field: string);
  end;

procedure WriteField(var list: TStringList; field: string);
begin
end;

procedure Exemplo();
var
  field: string;
  list: TStringList;
begin
  WriteField(list, field);

  list.WriteField(field)
end;
```

Funções e procedimentos

Funções Tríades

Funções que recebem três parâmetros são consideravelmente mais difíceis de entender do que as díades.

A sugestão é pensar bastante antes de criar uma tríade.

Funções e procedimentos

Objetos como parâmetros

Quando uma função ou procedimento precisa ter mais de dois ou três parâmetros é provável que eles possam ser colocados em uma classe própria.

Por exemplo, esta função que vimos anteriormente, poderia ser simplificada pois os seus parâmetros poderiam ser colocados em uma classe cliente e na chamada da função seria passado apenas um parâmetro, sendo o objeto cliente.

```
function AtualizaEnderecoCliente(const sCgcCpf, sEndereco, sBairro, sCep: String; const sCodCidade: String = '';
                                 const sDDDFone: String = ''; const sFone: String = ''; const sCgf: String = '';
                                 iTipoEnd: Integer = 0; const sTabela: String = 'CLI'; const sCondicaoContribuinteCliente: string ='';
                                 const sComplementoEndereco: string = ''; const sNumeroEndereco: string = ''): Boolean;
```

Funções e procedimentos

Verbos e palavras-chave

Escolher bons nomes para funções pode ir desde explicar o propósito da função a ordem e a finalidade dos parâmetros.

```
// Combinação verbo/substantivo
procedure Escrever(nome: string);
begin
end;

// Informa que o nome é um campo
procedure EscreverCampo(nome: string);
begin
end;

// Informa a ordem dos parâmetros
function VerificarEsperadoIgualAtual(Esperado: string; Atual: string): Boolean;
begin
end;
```

Funções e procedimentos

Evite efeitos colaterais

Quando uma função ou procedimento promete fazer uma coisa, mas ela faz também outras coisas escondidas, acaba gerando efeitos colaterais na sua utilização.

Listagem 3-6
UserValidator.java

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

Funções e procedimentos

Evite efeitos colaterais

O nome dessa função poderia transmitir melhor o objetivo dela se nos mudarmos para o seguinte

checkPasswordAndInitializeSession

O novo nome deixa claro que a função verifica a senha e inicializa a sessão

Funções e procedimentos

Evite parâmetros de saídas

Parâmetros normalmente são interpretados como entrada

```
procedure Exemplo01(nome: string; out valor: string);
begin
end;
```

Funções e procedimentos

Evite parâmetros de saídas

Parâmetros normalmente são interpretados como entrada

```
procedure Exemplo01(nome: string; out valor: string);
begin
end;
```

```
function Exemplo02(nome: string): string;
begin
end;
```

Funções e procedimentos

Evite parâmetros de saídas

Parâmetros normalmente são interpretados como entrada

```
procedure Exemplo01(nome: string; out valor: string);
begin
end;
```

```
function Exemplo02(nome: string): string;
begin
end;
```

```
function Exemplo03(nome: string; out valor): string;
begin
end;
```

Funções e procedimentos

Separação comando consulta

As funções ou procedimentos devem fazer ou responder alguma coisa, mas não ambos, uma função ou altera o estado de um objeto ou retorna informações sobre ele. Efetuar duas tarefas costuma gerar confusão.

Por exemplo:

```
function Definir(atributo: string; valor: string): Boolean;
begin
end;

procedure Exemplo04();
begin
  if (Definir('nome', 'Gleryston Matos')) then
  begin
  end;
end;
```

Funções e procedimentos

Separação comando consulta

As funções ou procedimentos devem fazer ou responder alguma coisa, mas não ambos, uma função ou altera o estado de um objeto ou retorna informações sobre ele. Efetuar duas tarefas costuma gerar confusão.

Como ficaria melhor:

```
function AtributoExiste(atributo: string): Boolean;
begin
end;

procedure DefinirAtributo(atributo: string; valor: string);
begin
end;

procedure Exemplo05();
begin
  if (AtributoExiste('nome')) then
  begin
    DefinirAtributo('nome', 'Gleryston Matos');
  end;
end;
```

Funções e procedimentos

Prefira exceções a retorno de códigos de erro

```
if (deletePage(page) == E_OK) {
    if (registry.deleteReference(page.name) == E_OK) {
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
            logger.log("página excluída");
        } else {
            logger.log("configKey não foi excluída");
        }
    } else {
        logger.log("deleteReference não foi excluído do
registro");
    }
} else {
    logger.log("a exclusão falhou");
    return E_ERROR;
}
```

Funções e procedimentos

Prefira exceções a retorno de códigos de erro

```
try {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
    logger.log(e.getMessage());
}
```

Funções e procedimentos

Extraia os blocos try/catch

```
public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}
private void deletePageAndAllReferences(Page page) throws Exception
{
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

Funções e procedimentos

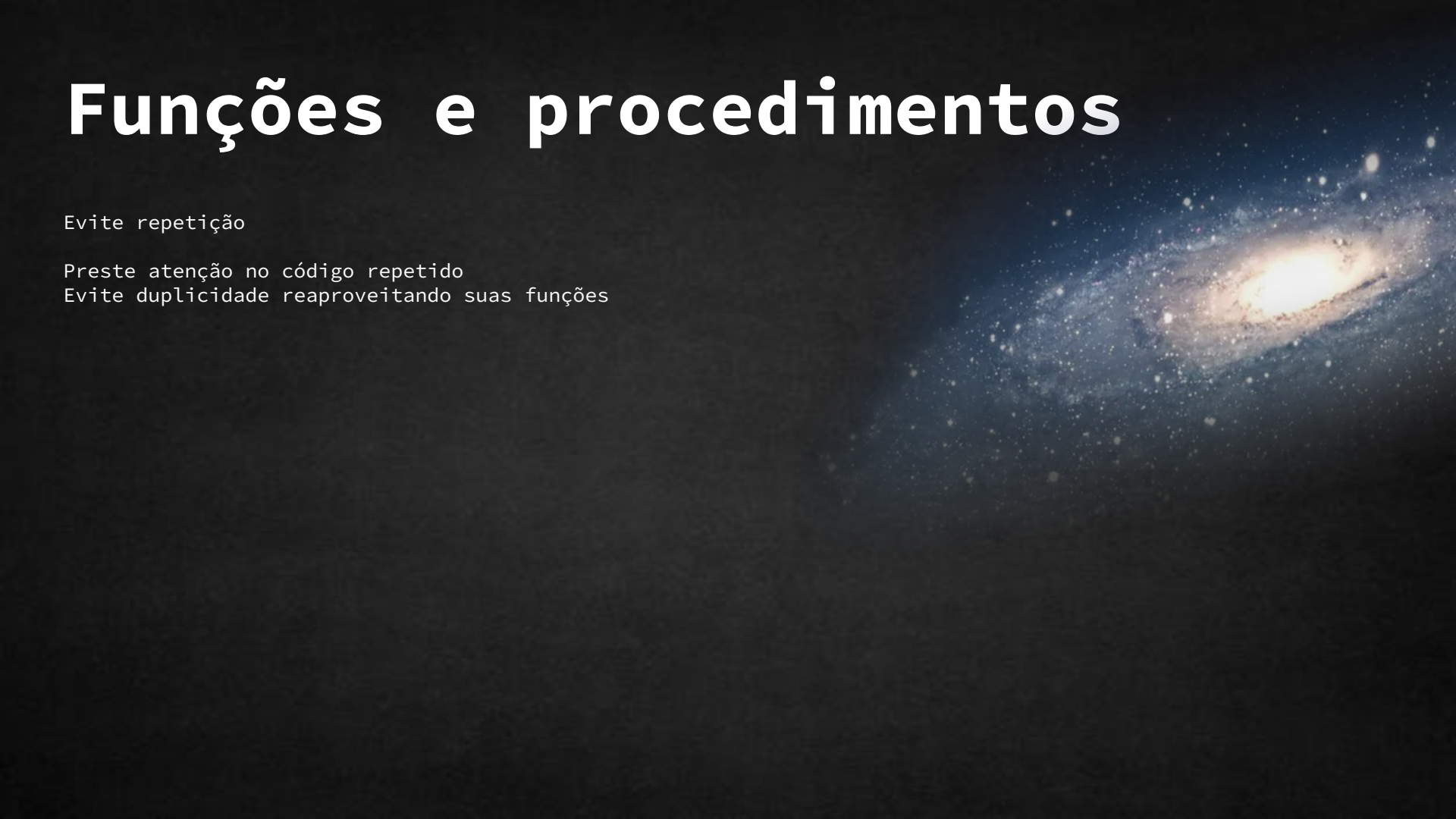
Extraia os blocos try/catch - Tratamento de erro é uma coisa só

```
public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}
private void deletePageAndAllReferences(Page page) throws Exception
{
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

Funções e procedimentos

Evite repetição

Preste atenção no código repetido
Evite duplicidade reaproveitando suas funções



Comentários

Quando é bom usar um comentario no código?

Comentários

Quando é bom usar um comentario no código?

Nunca!

Comentários

Quando é bom usar um comentario no código?

Nunca!

Brincadeiras a parte, comentários são algo bem delicado e que quase nunca é uma boa ideia usá-los no código.

Comentários

Quando é bom usar um comentario no código?

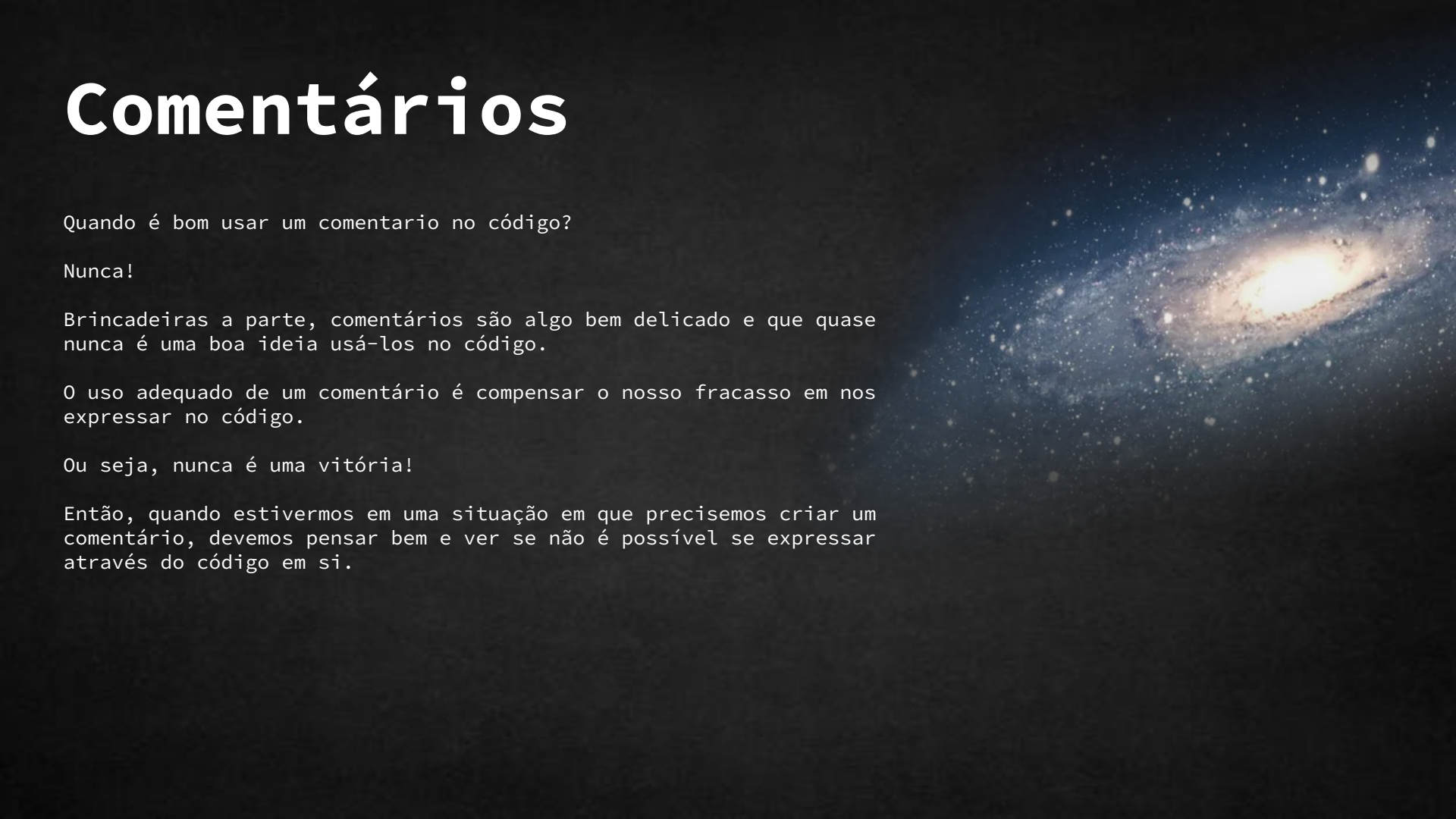
Nunca!

Brincadeiras a parte, comentários são algo bem delicado e que quase nunca é uma boa ideia usá-los no código.

O uso adequado de um comentário é compensar o nosso fracasso em nos expressar no código.

Ou seja, nunca é uma vitória!

Então, quando estivermos em uma situação em que precisemos criar um comentário, devemos pensar bem e ver se não é possível se expressar através do código em si.



Comentários

Porque não gosto de comentários?

Comentários mentem. Quanto mais antigo é um comentário e quanto mais longe ele estiver do código que ele descreve, é mais provável que ele esteja errado.

O motivo é simples, é muito difícil que programadores consigam mantê-los atualizados

Comentários

Códigos mudam e evoluem. Movem-se blocos de lá para cá, que se misturam, se reproduzem e se unem novamente, formando monstros gigantescos.

Por exemplo:

```
const
  HttpDataRegexp = '[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\' +
    's[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT';
  Request = '';
  Response = '';
  FileResponse = '';
  Locale = '';
  // Exemplo: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Comentários

Comentários compensam um código ruim?

A principal motivação para criarmos um comentário é um código ruim.

Nós criamos um módulo e sabemos que está confuso e desorganizado (temos ciência da bagunça)

Nesse momento nós dizemos “É melhor inserir um comentário”

Não seria melhor limpá-lo?

No lugar de usar o nosso tempo para criar explicações em comentários sobre o funcionamento do código, devemos usar esse tempo para limpar essa bagunça

Comentários

Explique-se no código

Existe uma noção errada de que não é possível explicar nosso objetivo através do código, porém esse é um pensamento errado, por exemplo, a pessoa que criou este comentário achou que não tinha como deixar claro seu objetivo através do código:

```
var
  funcionario: TFuncionario;
begin
  // Verifica se o funcionario tem direito a todos os beneficios
  if ((funcionario.Flags = TodasFlags) and (funcionario.Idade > 65)) then
  begin
  end;
end;
```

Comentários

Explique-se no código

Existe uma noção errada de que não é possível explicar nosso objetivo através do código, porém esse é um pensamento errado, por exemplo, a pessoa que criou este comentário achou que não tinha como deixar claro seu objetivo através do código:

```
var
  funcionario: TFuncionario;
begin
  if (funcionario.eElegivelParaTodosBeneficios()) then
  begin
  end;
end;
```

Comentários

Comentários bons

Uma boa utilização de um comentário pode ser para explicar a intenção por trás de uma decisão quando realmente ela não pode ser especificada no código

```
{ Observe que preciso deletar as baixas
  antes de Editar o contas a receber, pois
  caso contrário a LibX, não permitirá,
  pois há uma crítica: "Contas a Receber já
  possui baixas. Não pode ser editado." }
```

Comentários

Comentários bons

Podem nos alertar sobre consequências de mudar a implementação

```
function makeStandardHttpDateFormat: TSimpleDateFormat;
var
  df: TSimpleDateFormat;
begin
  // SimpleDateFormat não é thread-safe,
  // então precisamos criar cada instância independentemente.
  df := TSimpleDateFormat.Create('EEE, dd MMM yyyy HH:mm:ss z');
  df.SetTimeZone(TTimeZone.GetTimeZone('GMTZ'));
  Result := df;
end;
```

Comentários

Comentários Legais

Às vezes, os nossos padrões de programação corporativa nos forçam a escrever alguns comentários para atender questões legais

Por exemplo, frases sobre direitos autorais e informações necessárias que podem ser colocadas no início de um arquivo-fonte

Por exemplo:

```
// Direitos autorais (C) 2003,2004,2005 por Object Mentor, Inc. Todos
os direitos reservados.
// Distribuido sob os termos da versão 2 ou posterior da Licenca
Publica Geral da GNU.
```

Comentários

Comentários Legais

Às vezes, os nossos padrões de programação corporativa nos forçam a escrever alguns comentários para atender questões legais

Por exemplo, frases sobre direitos autorais e informações necessárias que podem ser colocadas no início de um arquivo-fonte

Por exemplo:

```
// Direitos autorais (C) 2003,2004,2005 por Object Mentor, Inc. Todos
os direitos reservados.
// Distribuido sob os termos da versão 2 ou posterior da Licenca
Publica Geral da GNU.
```

Porém temos de ter cuidado, esse tipo de comentário não deve ser um contrato ou n termos legais. Onde for possível devemos fazer referência a uma licença padrão ou outro documento externo em vez de colocar todos os termos e condições no mesmo comentário

Comentários

Comentários TODO

São tarefas que os programadores que acham que devem ser efetuadas, mas por alguma razão, não podem fazer no momento

É uma alternativa para organizar os pensamentos enquanto está desenvolvendo algo

```
// TODO: realizar a baixa do vencimento
```

Comentários

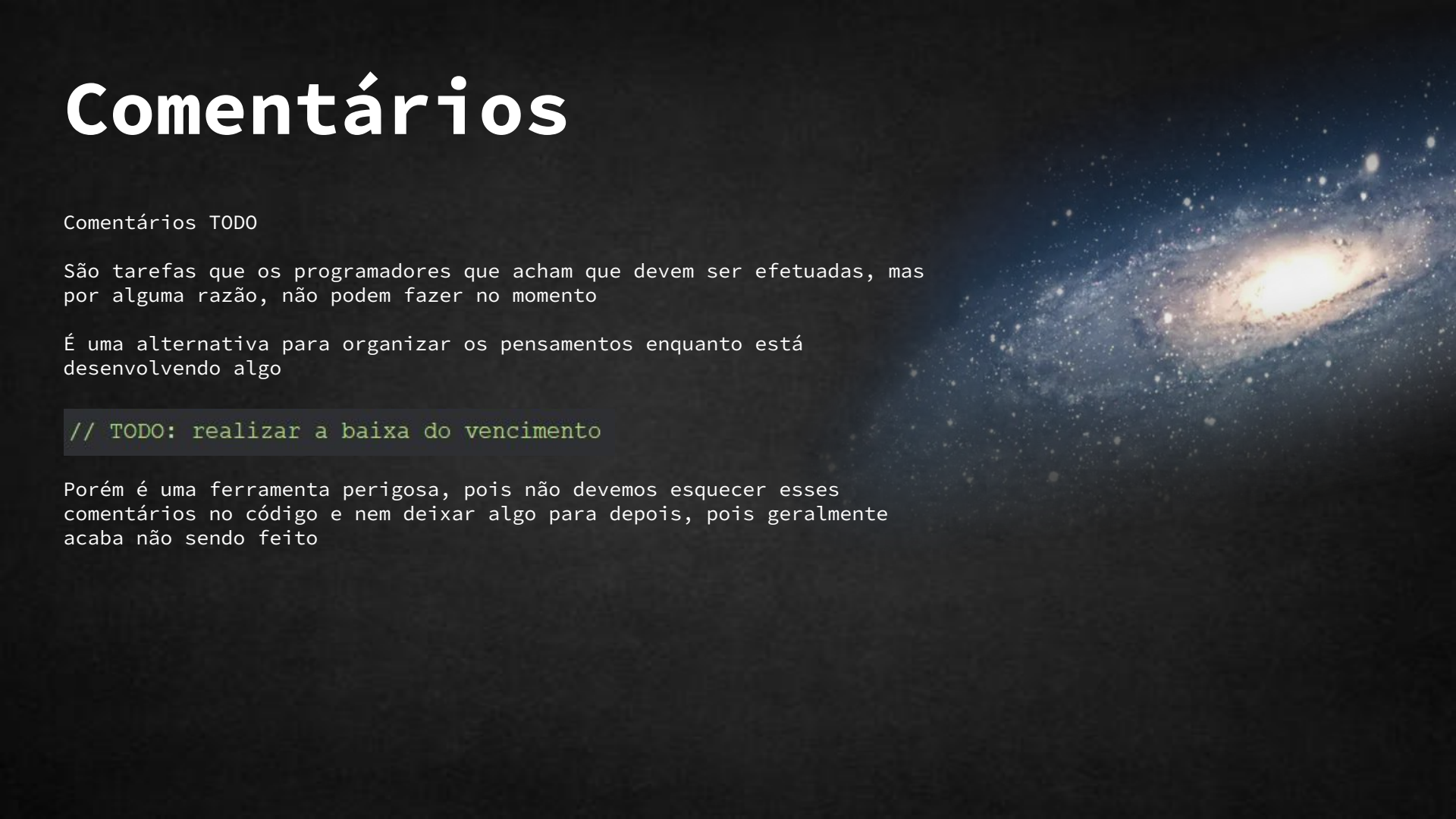
Comentários TODO

São tarefas que os programadores que acham que devem ser efetuadas, mas por alguma razão, não podem fazer no momento

É uma alternativa para organizar os pensamentos enquanto está desenvolvendo algo

```
// TODO: realizar a baixa do vencimento
```

Porém é uma ferramenta perigosa, pois não devemos esquecer esses comentários no código e nem deixar algo para depois, pois geralmente acaba não sendo feito



Comentários

Comentários ruins

A maioria dos comentários cai nessa categoria. Geralmente são desculpas para um código de baixa qualidade ou justificativas para a falta de decisões.

Comentários

Murmúrios

Murmúrios em comentários não tem muito significado para os programadores que estão lendo

Eles acabam nos obrigando a analisar outros módulos da aplicação em busca de um significado para aquela informação

```
public void loadProperties()
{
    try
    {
        String propertiesPath =
            propertiesLocation + "/" + PROPERTIES_FILE;
        FileInputStream propertiesStream =
            new FileInputStream(propertiesPath);
        loadedProperties.load(propertiesStream);
    }
    catch(IOException e)
    {
    // Nenhum arquivo de propriedades significa que todos os padrões
estão carregados
    }
}
```

Oque são esses padrões?

Quem carregou eles?

Onde eles estão?

Comentários

Comentários redundantes

São aqueles comentários que não são mais informativos que o próprio código

```
// Este procedimento serve para salvar
procedure Salvar;
begin
end;
```

Comentários

Marcadores de posição

Não devemos exagerar pois vão acabar sendo ignorados pelo leitor

```
// ***************** STATUS ***************** //
// Status
stErro = 1;
stAguardandoRoteiro = 2;
stAguardandoEnvio = 3;
// Eventos
evValidarDados = 1;
evConsultarTrecho = 2;
evSolicitarTrecho = 3;
```

Comentários

Comentários ao lado de chaves de fechamento

Podem até fazer sentido em funções muito longas (Lembrando que não devemos ter funções muito longas, se existe essa necessidade, pode indicar que temos que melhorar esse código) , mas no geral só serve para desorganizar o código

```
procedure NomeBuscavelRefatoracao;
{mais código aqui... }
begin
  realDaysPerIdealDay := 4;
  sum := 0;
  for j := 0 to NUMBER_OF_TASKS do
  begin
      realTaskDays := taskEstimate[j] * realDaysPerIdealDay;
      realTaskWeeks := realTaskDays / WORK_DAYS_PER_WEEK;
      sum := sum + realTaskWeeks
  end; // fim do for
end; // fim do procedre
```

Comentários

Código como comentário

Poucas práticas são tão condenáveis quanto explicar o código nos comentários.

Não façam isso!

```
procedure TfrmCdCnf.MasterPost;
begin
  if (dqrMain.State in [dsEdit,dsInsert]) then
  begin
       dqrMain.Post;
       //dqrMain.ApplyUpdates;
       MainServer.StartTransaction;
       try
           dqrMain.ApplyUpdates;
           MainServer.Commit;
       except
           on Exception do
           begin
               MainServer.Rollback;
               raise;
           end;
       end;
  end;
  ddsMain.RetrieveEditors;
end;
```

Comentários

Na maioria das vezes acaba sendo possível substituir um comentário por uma função ou variável

Formatação

Devemos tomar conta para que o nosso código fique bem formatado, escolher uma série de regras simples para governar o nosso código, e então, aplicá-las de forma consciente

Se estivermos trabalhando em equipe, então, todos devem concordar com uma única série de regras de formatação

Não devemos formatar nosso código como um bando de marinheiros bêbados

Formatação

O objetivo da formatação

Primeiramente e sendo bem claro, a formatação do código é importante

Ela serve como uma comunicação e essa é a primeira regra de negócio de um desenvolvedor profissional

É comum pensarmos que “Fazer funcionar” é a primeira regra, mas a esta altura creio que já podemos descartar esse conceito da nossa mente

As funcionalidades que criamos hoje tem grandes chances de serem modificadas na próxima release do projeto, e a legibilidade do nosso código tem um grande efeito em todas as mudanças que serão feitas

Então quais as questões sobre formatação que nos ajuda a nos comunicar melhor?

Formatação

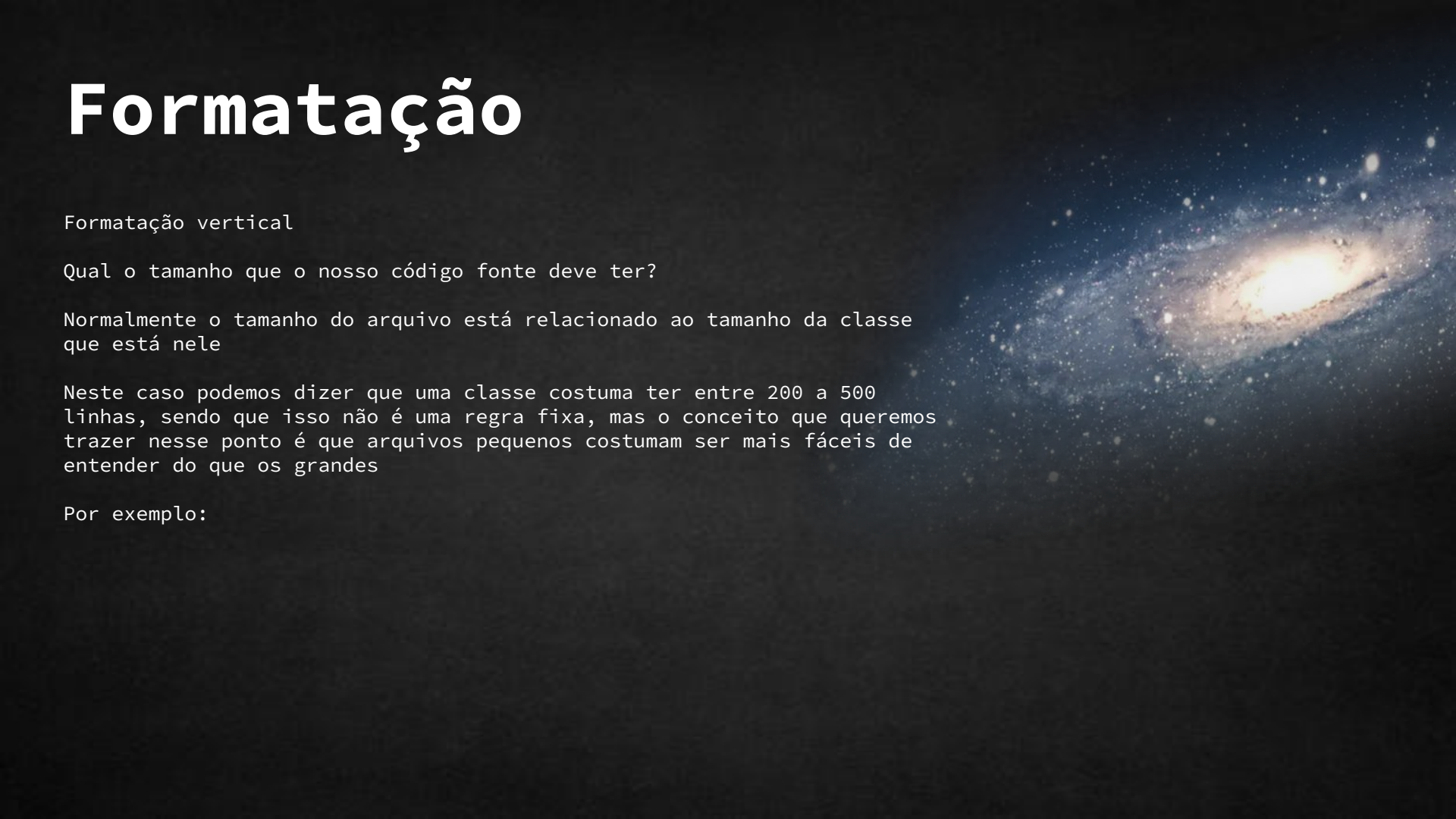
Formatação vertical

Qual o tamanho que o nosso código fonte deve ter?

Normalmente o tamanho do arquivo está relacionado ao tamanho da classe que está nele

Neste caso podemos dizer que uma classe costuma ter entre 200 a 500 linhas, sendo que isso não é uma regra fixa, mas o conceito que queremos trazer nesse ponto é que arquivos pequenos costumam ser mais fáceis de entender do que os grandes

Por exemplo:



Formatação

Formatação vertical

Qual o tamanho que o nosso código fonte deve ter?

Normalmente o tamanho do arquivo está relacionado ao tamanho da classe que está nele

Neste caso podemos dizer que uma classe costuma ter entre 200 a 500 linhas, sendo que isso não é uma regra fixa, mas o conceito que queremos trazer nesse ponto é que arquivos pequenos costumam ser mais fáceis de entender do que os grandes

Por exemplo:

Formatação

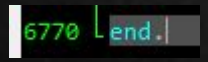
Formatação vertical

Qual o tamanho que o nosso código fonte deve ter?

Normalmente o tamanho do arquivo está relacionado ao tamanho da classe que está nele

Neste caso podemos dizer que uma classe costuma ter entre 200 a 500 linhas, sendo que isso não é uma regra fixa, mas o conceito que queremos trazer nesse ponto é que arquivos pequenos costumam ser mais fáceis de entender do que os grandes

Por exemplo:



Formatação

Formatação vertical

Qual o tamanho que o nosso código fonte deve ter?

Normalmente o tamanho do arquivo está relacionado ao tamanho da classe que está nele

Neste caso podemos dizer que uma classe costuma ter entre 200 a 500 linhas, sendo que isso não é uma regra fixa, mas o conceito que queremos trazer nesse ponto é que arquivos pequenos costumam ser mais fáceis de entender do que os grandes

Por exemplo:

Formatação

A metáfora do jornal

Podemos comparar um arquivo ou uma classe com um artigo de jornal bem redigido

Primeiro lemos o mesmo verticalmente, tendo no topo a manchete que nos informa de que se trata a história (Declaração da classe), o que já nos permite decidir se queremos ou não ler o artigo

O primeiro parágrafo apresenta uma sinopse da história toda, omitindo todos os detalhes falando mais de uma maneira geral (Declaração dos métodos)

Ao seguir com a leitura, verticalmente, vão surgindo mais detalhes, até que datas, nomes, citações e outras minúcias sejam apresentadas (Implementação dos métodos)

Não seja a próxima vítima

Marcos Aurélio Pereira
CEO
www.maisdados.com.br

Crimes cibernéticos

O Brasil ocupa lugar de destaque no cenário global de ataques cibernéticos. Apesar de o último ano ter feito 42,4 milhões de vítimas, muitos brasileiros ainda não protegem seus dados corretamente. Não podemos ignorar esse número, visto que a segurança das informações é considerada um dos pilares para o bom funcionamento dos negócios. Para se ter uma ideia, só no último ano, a perda financeira com essa prática no país foi de US$ 10,3 bilhões.

São cada vez mais raras as pessoas que não possuem pelo menos um endereço de e-mail, do qual realizam comunicações importantes. Seja na vida profissional ou pessoal, ignorar a utilização de mecanismos de armazenamento e segurança dos dados acarreta grandes riscos, pois, no mesmo passo em que o alcance das informações está mais fácil e rápido, também é verdade que a segurança na web se tornou ainda mais complexa.

Nos endereços eletrôni-

bancos, lojas e propagandas. O fato é que somos vigiados constantemente, e as pesquisas feitas na rede podem cair nas mãos de indivíduos mal-intencionados e em empresas como o Google. Esse mapeamento de dados é interessante para facilitar "compras online", mas é malicioso em situações em que o usuário fornece os números de CPF e identidade e senhas pessoais. Para não sofrer ataques cibernéticos, é necessário manter atualizados os sistemas operacionais e, mais do que isso, acessar sites confiáveis e que não resultem em futuras dores de cabeça.

A pirataria de softwares também está entre os motivos que levam a insegurança para o ambiente digital. A prática tem forte presença na indústria brasileira, e a prova disso são os recentes estudos que revelam também seus impactos na economia nacional. De acordo com um estudo realizado pela Business Software Alliance, também conhecida como The Software Alliance, hoje, quase metade dos softwares instalados no

mas usados em computado
res e em outras tecnologi
torna as informações ma
vulneráveis e favorece a in
dência de ataques cibernét
cos. Outro problema é a
zão recorrente para que
pessoas não adquirum so
twares originais é que
maioria é alugável e exige
vestimentos cíclicos. Ni
guém quer pagar por u
serviço e renová-lo period
camente. É nesse prazo
"não atualização" que está
brecha para a entrada de v
rus e outras ameaças.

Em tempos de "informa
ções em apenas um clique
todas as decisões a favor
segurança são importante
para resguardar e garantir
idoneidade e utilização ad
quada de dados. Não dev
nos subestimar a internet,
as pessoas que a usam
achar elevado o investimen
em softwares legalizados,
Brasil avança a cada dia
oferta de soluções que pre
nam crimes virtuais, mas ai
da estamos longe do ideal,
que, na medida em que
pessoas se acham imun
aos ataques, muitas máfi

Formatação

Espaçamento vertical entre conceitos

Quase todo código é lido da esquerda para a direita e de cima para baixo

Cada linha representa uma expressão ou uma estrutura, e cada grupo de linhas representa um pensamento completo

Esses pensamentos devem ficar separados por linhas em branco

Listagem 5-2
BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Formatação

Continuidade vertical

Se o espaçamento separa conceitos, então a continuidade vertical indica uma associação íntima

Linhas de código que estão intimamente relacionadas devem aparecer verticalmente unidas

Por exemplo a declaração das variáveis da seguinte classe:

Listagem 5.3

```
public class ReporterConfig {

  /**
   * The class name of the reporter listener
   */
  private String m_className;

  /**
   * The properties of the reporter listener
   */
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Formatação

Continuidade vertical

Se o espaçamento separa conceitos, então a continuidade vertical indica uma associação íntima

Linhas de código que estão intimamente relacionadas devem aparecer verticalmente unidas

Por exemplo a declaração das variáveis da seguinte classe:

Listagem 5.4

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

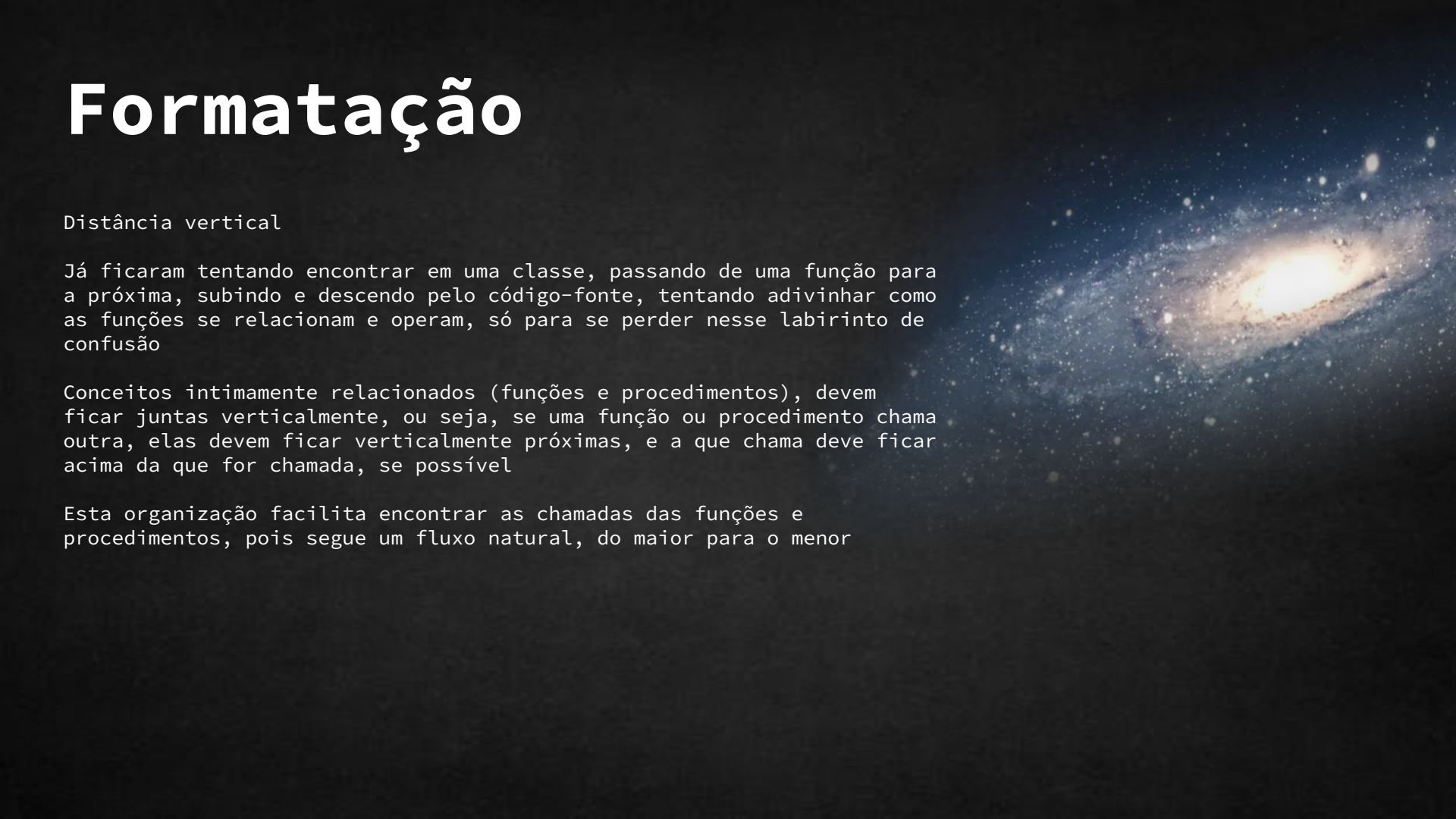
Formatação

Distância vertical

Já ficaram tentando encontrar em uma classe, passando de uma função para a próxima, subindo e descendo pelo código-fonte, tentando adivinhar como as funções se relacionam e operam, só para se perder nesse labirinto de confusão

Conceitos intimamente relacionados (funções e procedimentos), devem ficar juntas verticalmente, ou seja, se uma função ou procedimento chama outra, elas devem ficar verticalmente próximas, e a que chama deve ficar acima da que for chamada, se possível

Esta organização facilita encontrar as chamadas das funções e procedimentos, pois segue um fluxo natural, do maior para o menor

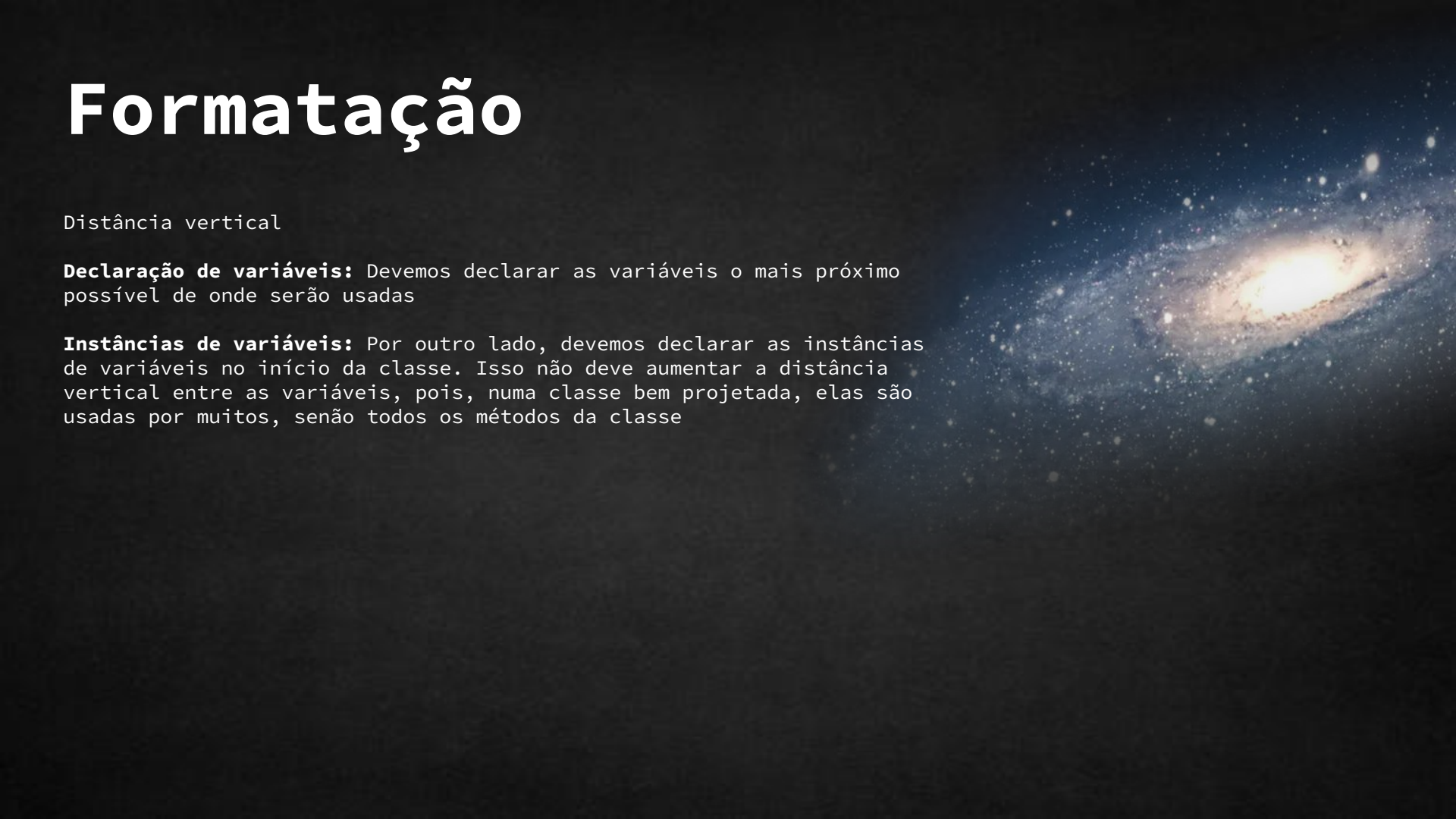


Formatação

Distância vertical

Declaração de variáveis: Devemos declarar as variáveis o mais próximo possível de onde serão usadas

Instâncias de variáveis: Por outro lado, devemos declarar as instâncias de variáveis no início da classe. Isso não deve aumentar a distância vertical entre as variáveis, pois, numa classe bem projetada, elas são usadas por muitos, senão todos os métodos da classe



Formatação

Espaçamento e continuidade horizontal

Usamos o espaço em branco horizontal para associar coisas que estão intimamente relacionadas e para desassociar outras fracamente relacionadas

```
procedure TfrmCdCnf.FormCreate(Sender:
TObject);
begin
  HideTabs(pcnMain);
  HideTabs(pcnDetail);
  CnfList := TCnfList.Create;
  CnfList.Grid := sgrCnf;
  pcnMain.ActivePage := tshGrid;
  pcnButtons.ActivePage := tshNovo;
end;
```

Formatação

Alinhamento horizontal

Devemos evitar esse tipo de alinhamento, pois enfatiza as coisas erradas
e afasta os olhos do propósito real

```
LIOHandleSSL :=                          TIdServerIOHandlerSSLOpenSSL.Create(AIdHTTPWebBrokerBridge);
LIOHandleSSL.SSLOptions.CertFile :=      FHorseProviderIOHandleSSL.CertFile;
LIOHandleSSL.SSLOptions.RootCertFile := FHorseProviderIOHandleSSL.FRootCertFile;
LIOHandleSSL.SSLOptions.KeyFile :=       FHorseProviderIOHandleSSL.KeyFile;
LIOHandleSSL.OnGetPassword :=            FHorseProviderIOHandleSSL.OnGetPassword;
AIdHTTPWebBrokerBridge.IOHandler :=      LIOHandleSSL;
```

Formatação

Indentação

Uma boa indentação do código ajuda a visualizar todo o escopo e identificar os pontos que queremos dar atenção

Torna mais fácil e rápida a identificação de situações e regras relevantes

```
public class FitNesseServer implements SocketServer { private
FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender =
new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```

Formatação

Indentação

Uma boa indentação do código ajuda a visualizar todo o escopo e identificar os pontos que queremos dar atenção

Torna mais fácil e rápida a identificação de situações e regras relevantes

```
public class FitNesseServer implements SocketServer {
    private FitNesseContext context;

    public FitNesseServer(FitNesseContext context) {
        this.context = context;
    }

    public void serve(Socket s) {
        serve(s, 10000);
    }

    public void serve(Socket s, long requestTimeout) {
        try {
            FitNesseExpediter sender = new FitNesseExpediter(s,
context);
            sender.setRequestParsingTimeLimit(requestTimeout);
            sender.start();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

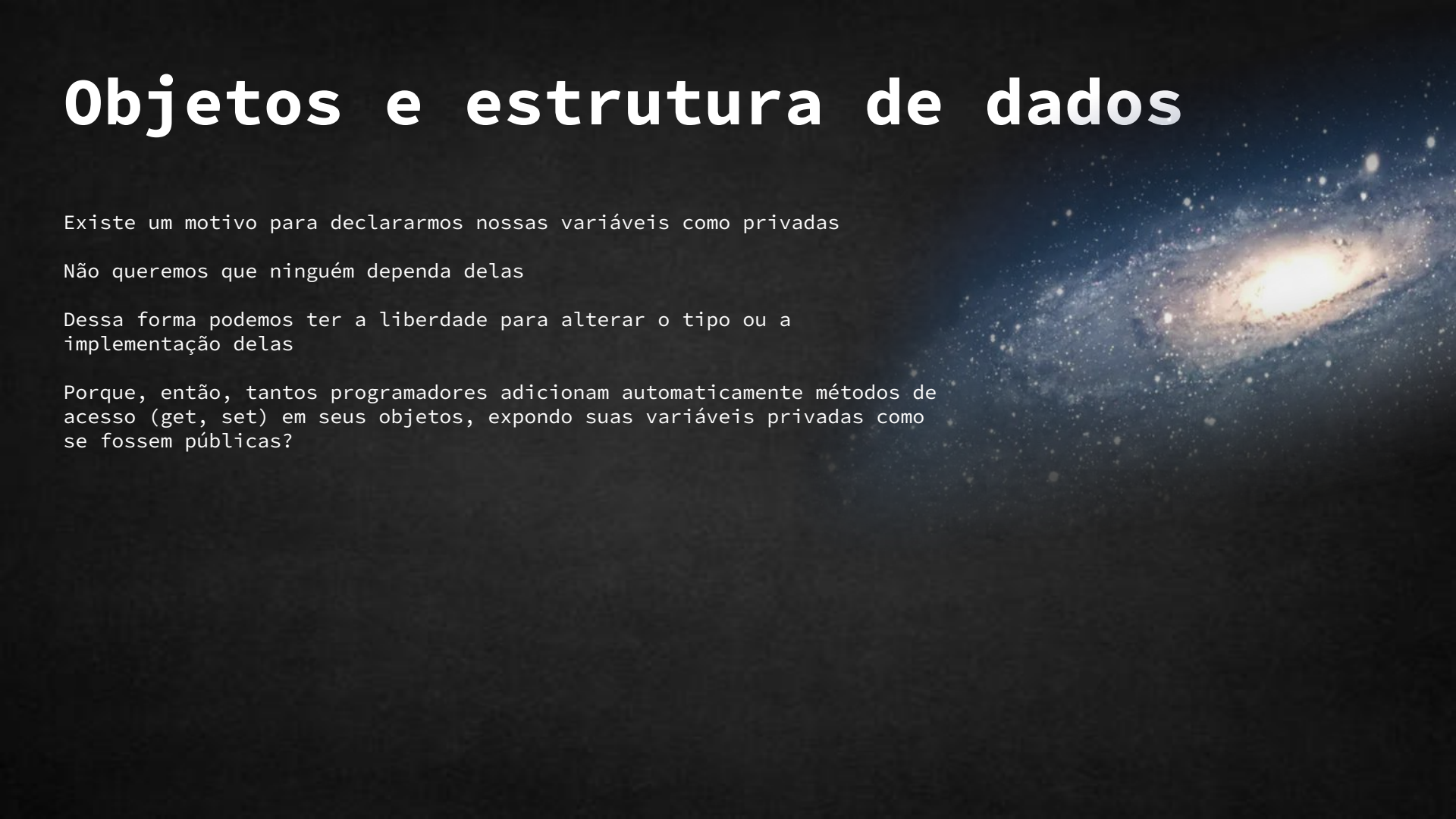
Formatação

Regras da equipe

Sempre devemos optar pelas regras da nossa equipe

Regras T&L

Objetos e estrutura de dados



Existe um motivo para declararmos nossas variáveis como privadas

Não queremos que ninguém dependa delas

Dessa forma podemos ter a liberdade para alterar o tipo ou a implementação delas

Porque, então, tantos programadores adicionam automaticamente métodos de acesso (get, set) em seus objetos, expondo suas variáveis privadas como se fossem públicas?

Objetos e estrutura de dados

Abstração de dados

Vamos analisar a diferença entre os seguintes códigos:

Listagem 6-1
Caso concreto

```
public class Point {
  public double x;
  public double y;
}
```

Listagem 6-2
Caso abstrato

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

Objetos e estrutura de dados

Abstração de dados

Ocultar a implementação não é só uma questão de colocar uma camada de funções entre as variáveis

Uma classe não passa suas variáveis simplesmente por meio de métodos de escrita e leitura

Em vez disso, ela expõe interfaces abstratas que permitem aos usuários manipular a essência dos dados sem precisar conhecer a implementação

Listagem 6-3
Veículo concreto

```
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}
```

Objetos e estrutura de dados

Antissimétrica data/objeto

Esses dois exemplos mostram a diferença entre objetos e estruturas de dados

Os objetos usam abstrações para esconder seus dados e expõem as funções que operam esses dados

As estruturas de dados expõem seus dados e não possuem funções significativas

Objetos e estrutura de dados

Antissimétrica data/objeto

Esses dois exemplos mostram a diferença entre objetos e estruturas de dados

Os objetos usam abstrações para esconder seus dados e expõem as funções que operam esses dados

As estruturas de dados expõem seus dados e não possuem funções significativas

Objetos e estrutura de dados

Antissimétrica data/objeto

O código procedimental (usado em estruturas de dados) facilita a adição de novas funções sem precisar alterar as estruturas de dados existentes

O código orientado a objeto facilita a adição de novas classes sem precisar alterar as funções existentes

Porém…

Objetos e estrutura de dados

Antissimétrica data/objeto

O código procedimental dificulta a adição de novas estruturas de dados, pois todas as funções teriam de ser alteradas

O código orientado a objeto dificulta a adição de novas funções, pois todas as classes teriam de ser alteradas

Tendo isso em mente…

Objetos e estrutura de dados

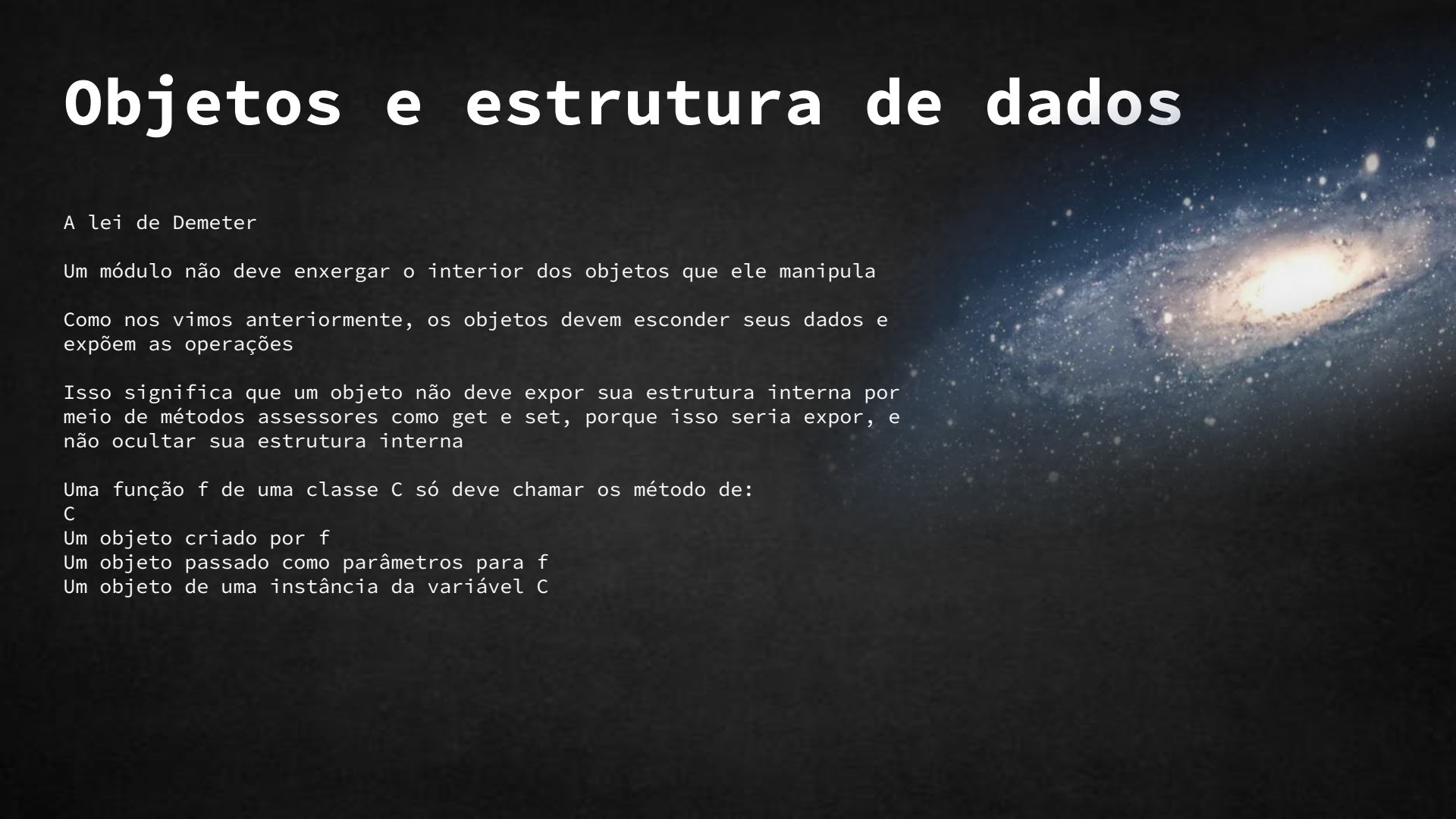
Antissimétrica data/objeto

Sabendo que cada uma dessas formas tem seus pontos positivos e negativos é fácil pensar em misturá-las e talvez usar o melhor de cada uma, criando estruturas híbridas

Essas estruturas possuem funções que fazem algo significativos e também variáveis ou funções de alteração e de acesso público

Esse tipo de estrutura deve ser evitada pois une as dificuldades além de pontos positivos onde ao resolver um acabamos criando outro na outra parte da estrutura

Objetos e estrutura de dados



A lei de Demeter

Um módulo não deve enxergar o interior dos objetos que ele manipula

Como nos vimos anteriormente, os objetos devem esconder seus dados e expõem as operações

Isso significa que um objeto não deve expor sua estrutura interna por meio de métodos assessores como get e set, porque isso seria expor, e não ocultar sua estrutura interna

Uma função f de uma classe C só deve chamar os método de:
C
Um objeto criado por f
Um objeto passado como parâmetros para f
Um objeto de uma instância da variável C

Objetos e estrutura de dados

Evite carrinhos de trem

```
final String outputDir = ctxt.getOptions().getScratchDir().
getAbsolutePath();
```

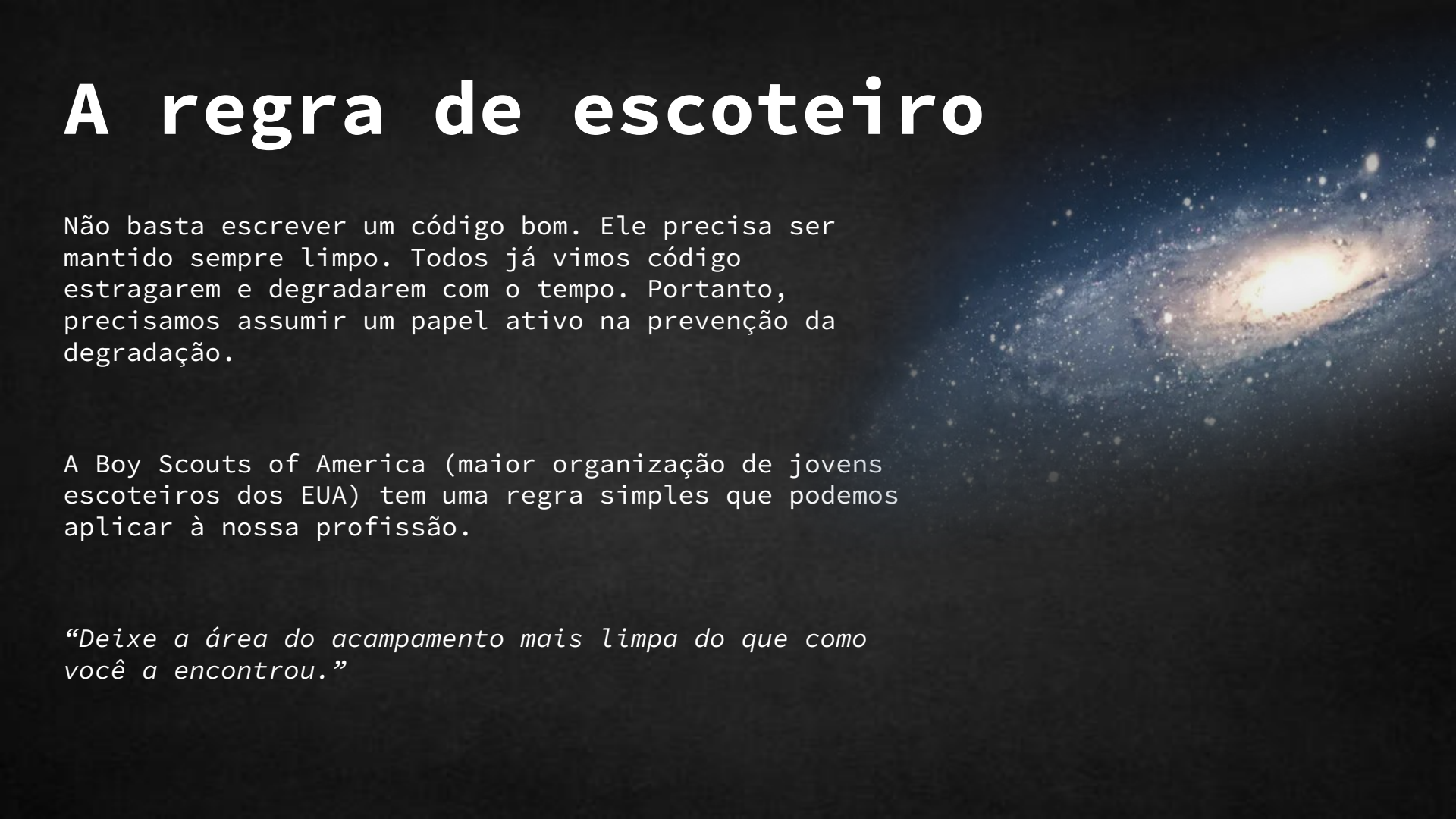
```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

A regra de escoteiro

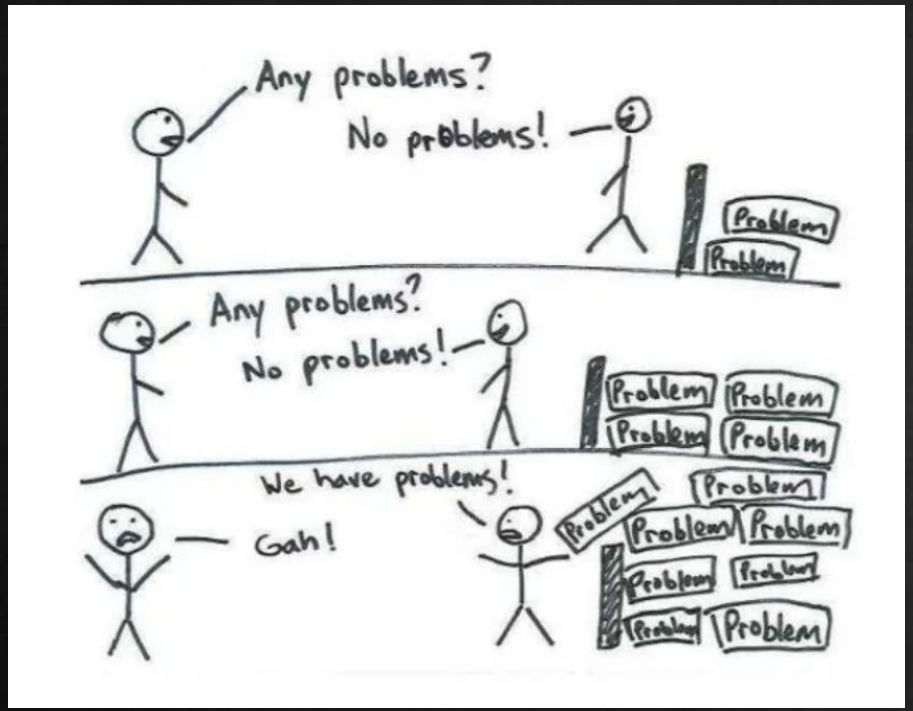
Não basta escrever um código bom. Ele precisa ser mantido sempre limpo. Todos já vimos código estragarem e degradarem com o tempo. Portanto, precisamos assumir um papel ativo na prevenção da degradação.

A Boy Scouts of America (maior organização de jovens escoteiros dos EUA) tem uma regra simples que podemos aplicar à nossa profissão.

“Deixe a área do acampamento mais limpa do que como você a encontrou.”



Não deixe acumular problemas



Código ruim cheira mal…

Torna o seu trabalho lento e desgastante com o passar do tempo

Pode arruinar seu projeto, carreira, empresa…

Fique atento!

Código Limpo

Bora praticar um pouco:
https://github.com/GlerystonMatos/GildedRose-Refactoring-Kata

Repositório com alguns dos exemplos:
https://github.com/GlerystonMatos/codigo-limpo

GitHub: https://github.com/GlerystonMatos
Medium: https://medium.com/@glerystonmatos
Linkedin: https://www.linkedin.com/in/glerystonmatos
Youtube: https://www.youtube.com/user/Glerystonmatos
Exemplos: https://github.com/GlerystonMatos/codigo-limpo

DUG-CE:
Grupo: https://t.me/DUGCE
Canal: https://www.youtube.com/c/DUGCE-CANAL

Perguntas?