Código Limpo:

Habilidades Práticas do Agile Software

Capítulo 5: Formatação

Este material foi desenvolvido pelo grupo PET/ADS do IFSP São Carlos

Formatação: introdução

Quando as pessoas olham o código, desejamos que fiquem impressionadas com a polidez, a consistência e a atenção aos detalhes presentes.

Você deve tomar conta para que seu **código fique bem formatado**, escolhendo regras simples que governem seu código e as aplicando de forma consistente.

Mas qual o objetivo da formatação?

O objetivo da formatação

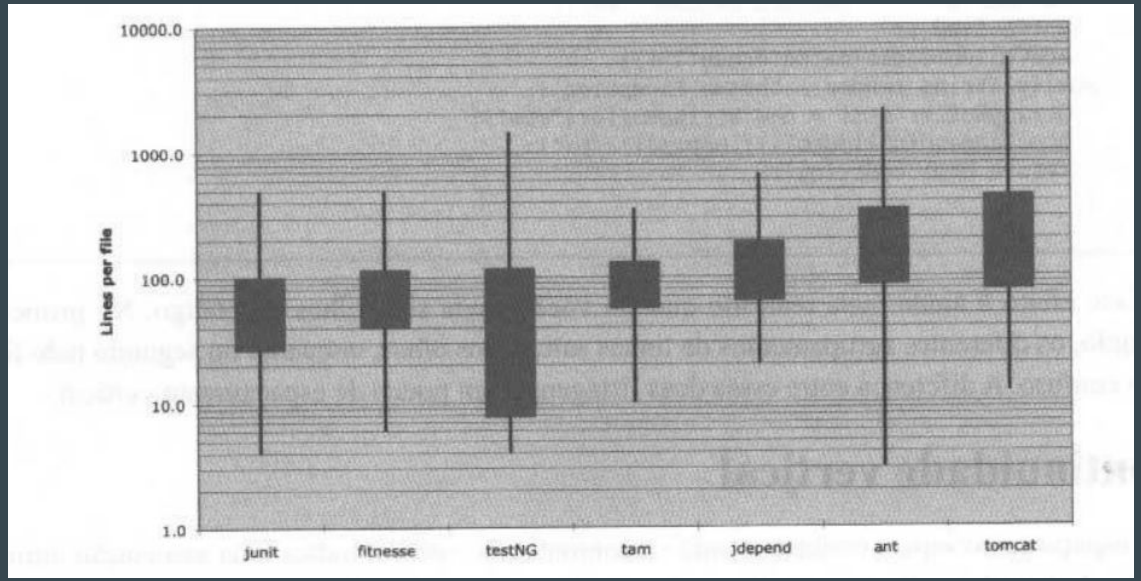
A formatação serve como uma **comunicação**, e essa é a primeira regra nos negócios de um desenvolvedor profissional.

A funcionalidade do seu código é muito importante, mas ela será alterada em um futuro próximo. Por isso, a **legibilidade** terá um grande **impacto** em versões futuras.

“Seu estilo e disciplina sobrevivem,
mesmo que seu código não”

Formatação vertical: exemplo

O gráfico a seguir ilustra o tamanho médio dos arquivos Java em sete projetos:



Formatação vertical: tamanho dos arquivos?

O seu código fonte deve ser de que tamanho?

O programa Fitnesse possui 50kloc, mas foi implementado a partir de arquivos que têm, em média, menos de 200 linhas. Nenhum arquivo tem mais do que 500 linhas.

Embora essa não seja uma regra, esse tamanho médio é um bom começo, pois arquivos pequenos são mais fáceis de entender e manter do que arquivos grandes.

A metáfora do Jornal: explicação

Um jornal é lido verticalmente. No topo você espera ver uma manchete que lhe diz do que se trata aquela matéria, e assim pensar se vale a pena a leitura ou não. O primeiro parágrafo é uma sinopse do resto da matéria, omitindo detalhes e falando de uma maneira mais geral. Ao prosseguir com a história os detalhes vão surgindo.

A metáfora do Jornal: conclusão

Desejamos que um código-fonte seja como um artigo de jornal:

- O nome deve ser **simples** mas **descritivo**
- O nome em si deve ser suficiente para nos dizer se estamos no arquivo certo ou não
- As partes **superiores** devem oferecer **conceitos** e **algoritmos** de **alto nível**
- Os **detalhes** devem ir surgindo conforme se move para **baixo**
- Os **arquivos** devem ser **numerosos,** porém **pequenos** e direto ao ponto

Espaçamento vertical entre conceitos: explicação

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

Esses pensamentos devem ser **separados por linhas em branco, que** indicam visualmente uma separação de conceitos.

Vejamos nos próximos slides como a separação de linhas faz diferença. É utilizado o mesmo código, porém no primeiro slide sem as linhas e no posterior com as linhas.

```
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("");
       html.append(childHtml()).append("");
       return html.toString();
   }
}
```

```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("");
       html.append(childHtml()).append("");
       return html.toString();
   }
}
```

Continuidade vertical: código antes

Nesse exemplo, os comentários tornam o código difícil de ler. Isso acontece porque a **continuidade vertical,** que indica uma associação íntima, foi quebrada:

```
public class ReporterConfig {
    /**
     * The class name of the repórter listener
     */
    private String tn_className;
    /**
     * The properties of the repórter listener
     */
    private List<Property> m_properties = new ArrayList<Property>() ;

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

Continuidade vertical: código depois

Com a remoção dos comentários desnecessários, as linhas de código que estão relacionadas aparecem verticalmente unidas. O código cabe em uma única visão:

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

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

Distância vertical: explicação

"Os conceitos intimamente relacionados devem ficar juntos verticalmente"

Obviamente, essa regra não funciona para conceitos em arquivos separados.

Não se devem separar em arquivos distintos conceitos intimamente relacionados, a menos que se tenha uma razão muito boa.

Distância vertical: declaração de variáveis

Variáveis devem ser declaradas o mais próximo possível de onde serão usadas:

```
private static void readPreferences() {
   InputStream is = null; //Em funções pequenas elas devem ficar no topo
   try {
       is = new FileInputStream(getPreferencesFile()) ;
       setPreferences(new Properties(getPreferences())) ;
       getPreferences().load(is) ;
   } catch (IOException e) {
       try {
           if (is != null) is.close();
       } catch (IOException e1) {
       }
   }
}
```

Distância vertical: declaração de variáveis

Variáveis de controle de loop devem ser declaradas dentro da estrutura de iteração:

```
public int countTestCases() {
   int count = 0;
   for (Test each : tests)
       count += each.countTestCases() ;
   return count;
}
```

Distância vertical: variáveis de instância

Deve - se declarar as **variáveis de instância** no **começo da classe**

Não há problema em deixar as variáveis de instância no topo, pois elas serão utilizadas em praticamente todos os métodos

```
public class Product {
    private String name; // Aqui ocorre a declaração
    private int quantity;

    public Product() {
        // Código do construtor
    }

    public void getProduct() {
        // Código do método
    }

}
```

Distância vertical: funções dependentes

Se uma função chama outra, então a função que chama deve ficar acima da função chamada.

A função de alto nível chama as funções abaixo dela, que por sua vez chamam aquelas abaixo delas.

Esse formato de organização facilita encontrar a função que foi chamada e também melhora a legibilidade do código.

Distância vertical: exemplo

```
public class WikiPageResponder implements SecureResponder {
   protected WikiPage page;
   protected PageData pageData;
   protected String pageTitle;
   protected Request request;
   protected PageCrawler crawler;

   public Response makeResponse(FitNesseContext context, Request request) throws Exception{
       String pageName = getPageNameOrDefault(request , "FrontPage");
       loadPage(pageName, context);
       if (page == null) return notFoundResponse(context, request);
       else return makePageResponse(context) ;
   }

   private String getPageNameOrDefault(Request request, String defaultPageName) {
       String pageName = request.getResource() ;
       if (StringUtil.isBlank(pageName)) pageName = defaultPageName ;
       return pageName;
   }
```

Distância vertical: exemplo

```
protected void loadPage(String resource, FitNesseContext context) throws Exception {
   WikiPagePath path = PathParser.parse(resource) ;
   crawler = context.root.getPageCrawler() ;
   crawler.setDeadEndStrategy( new VirtualEnabledPageCrawler()) ;
   page = crawler.getPage(context.root , path);
   if (page != null) pageData = page.getData();
}

private Response notFoundResponse(FitNesseContext context , Request request) throws Exception {
   return new NotFoundResponder().makeResponse(context , request);
}

private SimpleResponse makePageResponse(FitNesseContext context) throws Exception {
   pageTitle = PathParser.render( crawler.getFullPath(page));
   String html = makeHtml(context) ;
   SimpleResponse response = new SimpleResponse() ;
   response.setMaxAge( 0);
   response.setContent(html) ;
   return response;
}
```

Distância vertical: afinidade conceitual

Determinados bits de código querem ficar perto de outros bits. Eles possuem uma certa **afinidade**. Quanto **maior** a afinidade **menor** deve ser a **distância vertical**.

A afinidade pode surgir devido a uma função chamar a outra, ou uma função usar uma mesma variável. Ela ainda pode vir de funções que possuem propósitos parecidos.

Distância vertical: afinidade conceitual

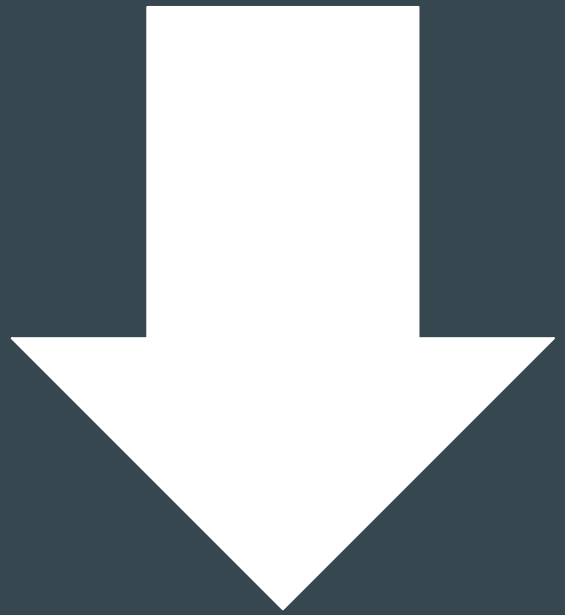
As funções a seguir estão próximas, pois compartilham nomes e efetuam operações muito parecidas:

```
static public void assertTrue(String message, boolean condition) {
    if (!condition) fail(message);
}

static public void assertTrue(boolean condition) {
    assertTrue(null, condition);
}

static public void assertFalse(String message, boolean condition) {
    assertTrue(message, !condition);
}

static public void assertFalse(boolean condition) {
    assertFalse(null, condition);
}
```

Ordenação vertical

As chamadas de dependência devem apontar para baixo. Sendo assim, a função chamada deve ficar embaixo da função que a chamou.

Isto cria um fluxo natural de **aumento de detalhamento** conforme se **desce** pelo código, assim como num artigo de jornal.



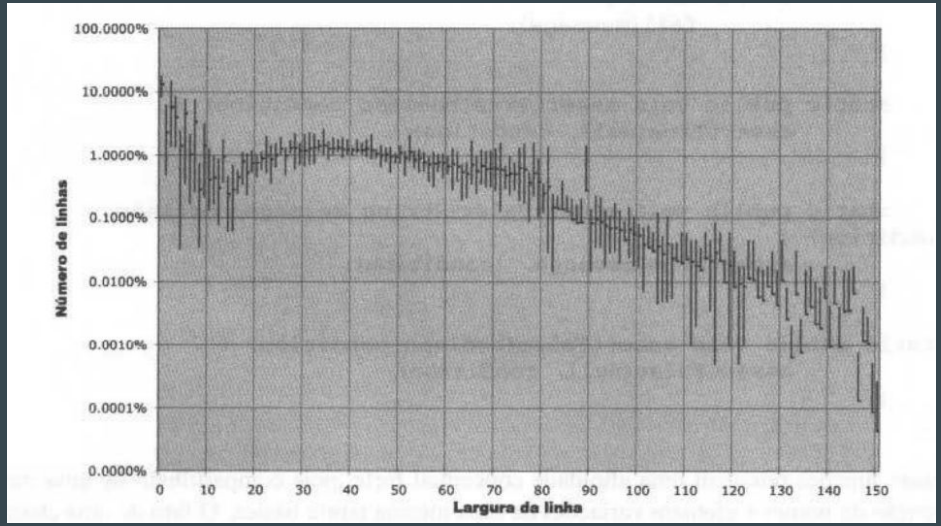
Formatação horizontal

Qual deve ser o tamanho de uma linha?

O recomendado é usar linhas pequenas, entre 20 e 60 caracteres.

Nos projetos do gráfico, raramente uma linha passa de 80 caracteres.

Tente escrever no máximo 120 caracteres por linha.



Espaçamento e continuidade horizontal

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

```
private void measureLine(String line) {
   lineCount++;
   int lineSize = line.length();
   totalChars += lineSize;
   lineWidthHistogram.addLine(lineSize, lineCount);
   recordWidestLine(lineSize);
}
```

No exemplo acima, os **operadores de atribuição** são colocados **entre espaços**. Isso cria uma separação clara entre o lado direito e o lado esquerdo.

Espaçamento e continuidade horizontal

Note que não há espaço entre o nome da função e o parênteses de abertura, pois esses estão intimamente relacionados.

```
private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
}
```

Dentro da função os parâmetros são separados com espaço para realçar a vírgula e mostrar que estão separados.

Espaçamento e continuidade horizontal

Outro uso do espaço em branco é ajudar a **destacar a prioridade entre operadores**:

```
class Quadratic {
   public static double root1(double a, double b, double c) {
       double determinant = determinant(a, b, c);
       return (-b + Math.sqrt(determinant)) / (2*a);
   }

   public static double root2(int a, int b, int c) {
       double determinant = determinant(a, b, c);
       return (-b - Math.sqrt(determinant)) / (2*a);
   }

   private static double determinant(double a, double b, double c) {
       return b*b - 4*a*c; // Note que os operadores de maior precedência estão juntos.
   }
}
```

Indentação: explicação e código antes

A indentação é usada para criar uma hierarquia visual para dividir estruturas, classes, funções, estruturas de repetição entre outros.

No exemplo a seguir é ilustrado como a ausência de indentação prejudica a leitura.

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

Indentação: código depois

No exemplo a seguir é apresentado o mesmo código, agora indentado:

```
public class FitNesseServer implements SockerServer{

   private FitNesseContext context;
   ...

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

Ignorando a indentação

Em funções, loops ou estruturas de decisão pequenas deixamos a indentação de lado. Mas mesmos nesses casos ela é necessária, vejo no exemplo a seguir

```
public class CommentWidget extends TextWidget {

   public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

   public CommentWidget(ParentWidget parent, String text) {super(parent, text);}

   public String render() throws Exception {return "";}
}
```

Ignorando a indentação

O término e o começo das funções ficam muito mais claros após a indentação:

```
public class CommentWidget extends TextWidget {

    public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

    public CommentWidget(ParentWidget parent, String text) {
        super(parent, text);
    }

    public String render() throws Exception {
        return "";
    }
}
```

Escopos minúsculos

Quando o corpo de uma estrutura while ou for é minúsculo, tudo bem deixar em uma linha.

Só preste atenção com os espaços em branco. E para facilitar a visualização, coloque o ponto e vírgula na sua própria linha como no exemplo abaixo.

```
while (dis.read(buf, 0, readBufferSize) != -1)
   ;
```

Regra de equipes

Uma equipe de desenvolvedores deve escolher um único estilo de formatação e todos os membros devem segui-lo.

O software deve possuir um estilo consistente.

Um bom sistema de software é composto por uma série de documentos de fácil leitura

Regras de formatação do Uncle Bob: exemplos ...

```java
// O nome da classe é claro e direto, assim como citado na metáfora do jornal
public class CodeAnalyzer implements JavaFileAnalysis {
   private int lineCount;
   private int maxLineWidth;
   private int widestLineNumber;
   private LineWidthHistogram lineWidthHistogram;
   private int totalChars;

   public CodeAnalyzer() { // Aqui a linha em branco é usada para separar atributos e
métodos
       lineWidthHistogram = new LineWidthHistogram();
   }

   public static List<File> findJavaFiles(File parentDirectory) {
       List<File> files = new ArrayList<File>();
       findJavaFiles(parentDirectory, files);
       return files;
   }
```

Regras de formatação do Uncle Bob: exemplos ...

```
//Essa função é declarada perto da de cima pois  contém afinidade conceitual
private static void findJavaFiles(File parentDirectory, List<File> files) {
    for (File file : parentDirectory.listFiles()) {
        if (file.getName().endsWith(".java"))
            files.add(file);
        else if (file.isDirectory())
            findJavaFiles(file, files);
    }
}

public void analyzeFile(File javaFile) throws Exception {
    BufferedReader br = new BufferedReader(new FileReader(javaFile));
    String line;
    while ((line = br.readLine()) != null)
        measureLine(line);
}
```

Regras de formatação do Uncle Bob: exemplos ...

```
//Essa função é chamada pela de cima, por isso são declaradas  próximas
private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length() ; // Variável sendo declarada perto de onde será usada
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize) ;
}

//Essa função é chamada pela de cima, por isso são declaradas  próximas
private void recordWidestLine(int lineSize) {
    if (lineSize > maxLineWidth) {
        maxLineWidth = lineSize;
        widestLineNumber = lineCount;
    }
}

public int getLineCount() {
    return lineCount;
}
```

Regras de formatação do Uncle Bob: exemplos ...

```
public int getMaxLineWidth() {
    return maxLineWidth;
}

public int getWidestLineNumber() {
    return widestLineNumber;
}

public LineWidthHistogram getLineWidthHistogram() {
    return lineWidthHistogram;
}

public double getMeanLineWidth() {
    return (double) totalChars / lineCount;
}
```

Regras de formatação do Uncle Bob: exemplos ...

```
public int getMedianLineWidth() {
    Integer[] sortedWidths = getSortedWidths() ;
    int cumulativeLineCount = 0; // Variável sendo declarada perto de onde será usada
    for (int width : sortedWidths) {
        cumulativeLineCount += lineCountForWidth(width) ;
        if (cumulativeLineCount > lineCount / 2)
            return width;
    }
    throw new Error("Cannot get here");
}
```

Regras de formatação do Uncle Bob: exemplos ...

```
private int lineCountForWidth(int width) {
    return lineWidthHistogram.getLinesforWidth(width).size();
}

private Integer[] getSortedWidths() {
        Set<Integer> widths = lineWidthHistogram.getWidths();
        Integer[] sortedWidths = (widths.toArray( new Integer[0]));
        Arrays.sort(sortedWidths);
        return sortedWidths;
    }
}
```

Sobre o autor

Robert C. “Uncle Bob” Martin é desenvolvedor e consultor de software desde 1990. Ele é o fundador e o presidente da Object Mentor, Inc., uma equipe de consultores experientes que orientam seus clientes no mundo todo em C++, Java, C#, Ruby, OO, Padrões de Projeto, UML, Metodologias Agile e eXtreme Programming.

Este conjunto de slides foi elaborado a partir da obra:

MARTIN, Robert. **Código Limpo:**
Habilidades Práticas do Agile Software. 1. ed.
Rio de Janeiro: Alta Books, 2009.

A equipe

Noah Rissatti, Autora

Aluna e ex-bolsista do PET/ADS
LinkedIn

Lucas Oliveira, Revisor

Professor de Computação, é tutor do PET/ADS desde janeiro de 2023.
LinkedIn