

CAP 5/6 Livro – CÓDIGO LIMPO – FORMATAÇÃO – ERIC E JULIO CESAR

CAP 5 - OBJETIVO DA FORMATAÇÃO

A formatação serve como uma comunicação, que é a regra principal de um desenvolvedor profissional, ou seja, a legibilidade do código é algo extremamente importante, pois outras pessoas também irão ler seu código é por isso devem estar bem formatados para que sejam legíveis aos demais desenvolvedores.

O primeiro tipo de formatação é a Formatação Vertical, pois, é preciso saber qual será o tamanho do código fonte, as linhas verticais mostram os comprimentos mínimos e máximos em cada projeto. Por exemplo em Java, há vários projetos diferentes, e assim cada um tem um número aproximado de linhas.

O livro aponta como deve ser um código fonte, e o compara com um artigo de jornal, que começa informando os detalhes de alto nível sobre o assunto, e ao longo da leitura, vão aparecendo os detalhes menos importantes, é assim que fazemos o leitor se interessar pela leitura, mostrando já de começo os principais detalhes do assunto, para que ele se interessa a ler, caso contrário o leitor nem leria o jornal ou então o código.

ESPAÇAMENTO VERTICAL ENTRE CONCEITOS

Como a maioria dos códigos são lidos da esquerda para a direita e de cima para baixo, cada linha representa uma expressão ou estrutura, e cada grupo de linhas representa um pensamento, que devem ser separados por linhas em branco, essas linhas em branco são feitas para separar conceitos do código como por exemplo: separar variáveis de métodos.

SEM LINHAS EM BRANCO

```
1 import java.time.LocalDate;
2
3 public class Pessoaa {
4
5     public String nome;
6     public String sobrenome;
7     public LocalDate dataNascimento;
8     public String getNomeCompleto() {
9         return nome + sobrenome;
10    }
11    public int getIdade() {
12        return 1;
13    }
14 }
15
```

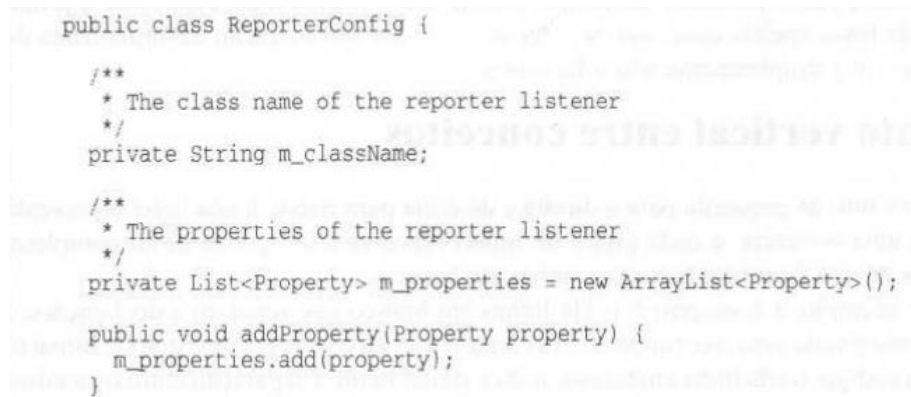
COM LINHAS EM BRANCO

```
1 import java.time.LocalDate;
2
3 public class Pessoaa {
4
5     public String nome;
6     public String sobrenome;
7     public LocalDate dataNascimento;
8
9     public String getNomeCompleto() {
10         return nome + sobrenome;
11     }
12
13     public int getIdade() {
14         return 1;
15     }
16 }
```

No código sem linhas em branco, fica confuso ler e entender o código, é preciso prestar mais atenção para conseguir obter o mesmo nível de entendimento do que o código com linhas em branco, que por estar separado, é muito mais fácil entender.

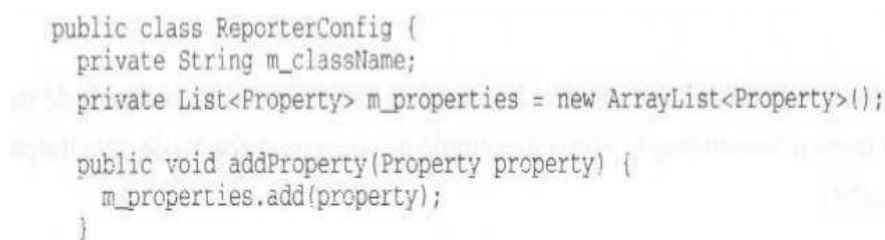
CONTINUIDADE VERTICAL

A continuidade vertical indica intimidade entre as linhas de código, códigos que estão intimamente relacionados devem aparecer verticalmente unidos. Por exemplo:



```
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);  
    }  
}
```

Nesta imagem, existem comentários entre as linhas de códigos, que são inúteis e também quebram a intimidade entre as duas linhas de código, além disso também fica meio confuso de entender o código.



```
public class ReporterConfig {  
    private String m_className;  
    private List<Property> m_properties = new ArrayList<Property>();  
  
    public void addProperty(Property property) {  
        m_properties.add(property);  
    }  
}
```

Já nesta segunda imagem fica mais fácil ler o código, e mesmo sem os comentários, é facilmente possível entender que no código está uma classe com duas variáveis e um método.

DISTÂNCIA VERTICAL

Como comentado acima sobre as linhas de códigos intimamente ligadas, também devem ficar juntas verticalmente, assim não é preciso o leitor ficar passando de função em função, tentando adivinhar como as funções se relacionam e operam, fazendo você melhorar seu código e também facilita a leitura e entendimento do mesmo.

Declarando variáveis. Devem ser declaradas o mais próximo possível de onde serão usadas. Em funções pequenas, as variáveis locais devem ficar no topo de cada função, como no exemplo abaixo.

```

private static void readPreferences() {
    InputStream is= null;
    try {
        is= new FileInputStream(getPreferencesFile());
        setPreferences(new Properties(getPreferences()));
        getPreferences().load(is);
    } catch (IOException e) {
        try {
            if (is != null)
                is.close();
        } catch (IOException e1) {
        }
    }
}

```

Já em variáveis de controle para loops, devem ser declaradas, dentro da estrutura de iteração, como no exemplo abaixo.

```

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

```

Instâncias de variáveis. Não existe exatamente um local específico para declará-las, por exemplo em C++ é comum ser usada a *regra da tesoura*, na qual é colocada todas as instâncias de variáveis no final. Já em Java, a convenção é coloca-las no início da classe. Nenhuma das duas é considerado certo ou errado, o importante é declará-las em um local bem conhecido, onde todos devem saber onde buscar as declarações. Um exemplo, é o estranho caso da classe a seguir, onde se você ler até metade do código verá duas instâncias de variáveis declaradas, elas estão muito cultas dentro da classe, e quem ler este código iria encontra-las por acaso.

```

theClass,
String name) {
    ...
}

public static Constructor<? extends TestCase>
getTestConstructor(Class<? extends TestCase> theClass)
throws NoSuchMethodException {
    ...
}

public static Test warning(final String message) {
    ...
}

private static String exceptionToString(Throwable t) {
    ...
}

private String fName;

private Vector<Test> fTests= new Vector<Test>(10);

public TestSuite() {
}

    public TestSuite(final Class<? extends TestCase> theClass)
{
    ...
}

    public TestSuite(Class<? extends TestCase> theClass, String
name) {
    ...
}
}

```

Funções dependentes. Uma função que chama outra, devem ficar verticalmente próximas, e a que chamar deve ficar acima da que foi chamada. Assim dá um fluxo natural ao programa, desse modo os leitores poderão confiar que declarações daquelas funções virão logo em seguida após seu uso. Como por exemplo na imagem abaixo.

```

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;
}

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();
}

```

É possível notar que as funções que estão acima, chamam a que está abaixo da que chamou, isso facilita encontrar as funções chamadas, e também facilita a legibilidade do código.

Afinidade conceitual. Determinados códigos que possuem uma afinidade conceitual, precisa ficar juntos, quanto maior afinidade entre os códigos, menor é a distância vertical entre eles, essa afinidade deve ser como uma dependência direta, como uma função chamando outra função usando uma variável. Um exemplo bom de afinidade conceitual, é de duas funções que possuem o mesmo tipo de nome e tenham variações da mesma tarefa, como no exemplo a seguir:



```
public class Assert {
    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);
    }
    ...
}
```

Formatação horizontal. A formatação horizontal diz qual seria o tamanho de uma linha de código, a média de cada projeto é de 20 a 60 caracteres nas linhas maiores, e em linhas menores cerca de 10 caracteres. Hoje em dia os monitores estão muito largos, então é possível, ultrapassar os 100 ou até 120 caracteres por linha, porém é desnecessário.

Espaçamento e continuidade horizontal. O espaçamento horizontal, é usado para associar coisas que estão intimamente relacionadas, e para desassociar outras fracamente relacionadas, considere o exemplo a seguir:

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

Os operadores estão entre espaços, e isso serve para destacá-los que são operadores. Por outro lado, não se deve separar o nome das funções e os parênteses, pois, a função e seus parâmetros estão intimamente relacionados, separá-los iria parecer que não estão juntos. Esse espaçamento deixa o código mais organizado e facilita a leitura do mesmo.

Endentação. A endentação é algo extremamente importante em um código, sem ela, os programas seriam ilegíveis para os humanos, pois o código fica totalmente bagunçado e desorganizado, segue um exemplo de um código sem endentação:

```
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(); } } }
```

Nesse código é quase impossível lê-lo sem estar totalmente focado para identificar o que é variável, método e construtor.

```
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();
        }
    }
}
```

Este é o mesmo código, porém com endentação, percebe-se que agora fica muito mais fácil entender e identificar quais são as ações dentro do código, e por isso a endentação é algo tão importante dentro de dele.

Escopos minúsculos. De vez em quando, o corpo de estruturas while ou for é minúscula, nesse caso é necessário verificar se a estrutura está endentada adequadamente e entre parênteses. É preciso também ter cuidado ao colocar o ponto-e-vírgula no final de um loop, se você não o tornar visível endentando-o em sua própria linha, fica difícil identifica-lo, como mostra o exemplo a seguir:

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

Regras de equipes. Cada programador possui uma regra de formatação preferida, porém quando ele trabalhar em equipe, essa regra é dela, ou seja, a equipe deve escolher uma regra e todos os desenvolvedores dessa equipe devem segui-la, assim todos constroem o código com o mesmo estilo, fazendo com que fique organizado e de fácil leitura, além disso mostra que a equipe está trabalhando em sintonia e organização.

Um exemplo de código bem organizado, seguindo regras de formatação simples é o código da imagem abaixo.

```
public class CodeAnalyzer implements JavaFileAnalysis {
    private int lineCount;
    private int maxLineWidth;
    private int widestLineNumber;
    private LineWidthHistogram lineWidthHistogram;
    private int totalChars;

    public CodeAnalyzer() {
        lineWidthHistogram = new LineWidthHistogram();
    }

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

    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);
    }

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

CAP 6 – OBJETOS E ESTRUTURAS DE DADOS

Há um motivo para declararmos nossas variáveis como privadas. Não queremos que alguém dependa delas. Desejamos ter a liberdade de alterar o tipo de implementação, seja por qualquer motivo. Por que, então, tantos programadores adicionam automaticamente métodos de acesso (getter, setters) em seus objetos, expondo suas variáveis privadas como se fossem públicas?

ABSTRAÇÃO DE DADOS

A abstração de dados é uma forma de **esconder** sua implementação, evitando aos usuários manipularem as essências dos dados. Vamos analisar as listagens 6.1 e 6.2. As duas apresentam dados de um ponto no plano cartesiano, mas a o caso concreto expõe sua implementação e o caso abstrato a esconde completamente.

Essa expõe a sua implementação.

Listagem 6-1 Caso concreto

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

Essa oculta a sua implementação.

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);  
}
```


ANTI-SIMETRIA A DATA/OBJETO

Os objetos utilizam-se de abstrações para esconder seus dados e expõem as funções que operam tais dados. As estruturas de dados expõem seus dados e não possuem funções significativas.

A seguir está a classe shape procedimental na listagem 6.5. A classe Geometry opera em três classes shape que são simples estruturas de dados sem nenhuma atividade. Todas as ações estão na classe Geometry.

Listagem 6-5

Classe shape procedimental

```
public class Square {
    public Point topLeft;
    public double side;
}

public class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}

public class Circle {
    public Point center;
    public double radius;
}

public class Geometry {
    public final double PI = 3.141592653589793;

    public double area(Object shape) throws NoSuch:
    {
        if (shape instanceof Square) {
            Square s = (Square)shape;
            return s.side * s.side;
        }
        else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle)shape;
            return r.height * r.width;
        }
        else if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
}
```

Isso é procedimental, mas nem sempre. Caso adicionássemos uma função `perimeter()` à `Geometry`. As classes `shape` não seriam afetadas, mas se adicionarmos uma nova classe `shape`, teríamos que alterar todas as funções em `Geometry`.

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 (OO), por outro lado, facilita a adição de novas classes sem precisar alterar as funções existentes.

A LEI DE DEMETER

A lei de Demeter diz que um método `f` de uma classe `C` só deve chamar os métodos de:

- `C`
- Um objeto criado por `f`
- Um objeto passado como parâmetro para `f`
- Um objeto dentro de uma instância da variável `C`

O método não pode chamar os métodos de objetos retornar por qualquer outra das funções permitidas. Em outro modo, falar apenas com conhecidos e não com estranhos. O código seguinte viola a Lei de Demeter, pois ele chama a função `getScratchDir()` no valor de `getOptions()` e, então, chama `getAbsolutePath()` no valor retornado de `getScratchDir()`.

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

CARRINHOS DE TREM

Este tipo de código é chamado assim por se parecer com vários carrinhos de trem acoplados. Elas são geralmente consideradas descuidadas e devem ser evitadas. Na maioria das vezes o melhor a se fazer é dividi-las assim:

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

O uso de funções de acesso confunde essas questões. Se o código tiver sido escrito como abaixo, então provavelmente não estaríamos se perguntando sobre o cumprimento ou não da Lei de Demeter.

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

Essa questão seria menos confusa se as estruturas de dados fossem todas públicas e nenhuma função, enquanto os objetos tivessem apenas variáveis privadas e funções públicas.

HÍBRIDOS

Essas são consideradas estruturas híbridas ruins, na qual ela é metade objeto e metade estrutura de dados. Elas possuem funções e também variáveis ou até mesmo métodos de acesso de alteração pública, tornando as variáveis privados em públicas, gerando assim outras funções externas a usarem as tais variáveis da forma com um programa procedimental usaria uma estrutura de dados.

ESTRUTURAS OCULTAS

Tem a função de ocultar suas estruturas internas, não deveríamos ser capazes de navegar por eles. Então como conseguiríamos o caminho absoluto de `scratchDir('diretório de rascunho')`

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

ou

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

A primeira opção poderia levar a abundância de métodos no objeto `ctxt`. A segunda presume que `getScratchDirectoryOption()` retorna uma estrutura de dados, e não um objeto.

A mistura adicionada de diferentes níveis de detalhes é um pouco confusa. Ponto, barras, extensão de arquivos e objetos `file` não deve ser misturado entre si e nem o código que os circunda. E se fizemos assim, na figura abaixo, isso permite o `ctxt` esconder suas estruturas internas e evitar que a função atual viole a Lei de Demeter ao navegar por objetos os quais ela não deveria enxergar.

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

OBJETOS DE TRANSFERÊNCIA DE DADOS

Como o próprio nome diz, é uma transferência de dados, ou DTO (sigla em inglês). Os DTOs são estruturas muito úteis, especialmente para se comunicar com os bancos de dados ou analisar sinteticamente mensagens provenientes de sockets e assim por diante.

O mais comum é o formulário “bean”. Os beans têm variáveis privadas manipuladas por métodos de escrita e leitura. Um exemplo de bean na figura abaixo:

Listagem 6-7 **address.java**

```
public class Address {
    private String street;
    private String streetExtra;
    private String city;
    private String state;
    private String zip;

    public Address(String street, String streetExtra,
                  String city, String state, String zip) {
        this.street = street;
        this.streetExtra = streetExtra;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public String getStreet() {
        return street;
    }

    public String getStreetExtra() {
        return streetExtra;
    }

    public String getCity() {
        return city;
    }

    public String getState() {
        return state;
    }

    public String getZip() {
        return zip;
    }
}
```

O ACTIVE RECORD

São formas especiais de DTOs, eles são estruturas de dados com variáveis públicas ou acessada por Beans, mas eles tipicamente possuem métodos de navegação, como o save e find. Esses Active Records são traduções diretas das tabelas de banco de dados ou de outras fontes de dados. Ter cautela ao tratar active records como objetos que poderá acarretar na criação de um híbrido na qual já foi comentando. Um conselho é tratar o active records como uma estrutura de dados e criar objetos separados que contenham regras de negócio e que ocultem seus dados internos.

RESUMINDO CAP 6

Os objetos expõem as ações e ocultam dados. Isso facilita a adição de novos tipos de objetos sem precisar modificar as ações existentes e dificulta a inclusão de novas atividades em objetos existentes.

As estruturas de dados expõem os dados e não possuem ações significativas. Isso facilita a adição de novas ações as estruturas de dados existentes e dificulta a inclusão de novas estruturas de dados em funções existentes.

Cabe ao programador decidir se ele deseja flexibilidade para adicionar novos tipos de dados e optar por objetos, ou caso desejar flexibilidade para adicionar novas ações, portanto, deveria optar por tipos de dados e procedimentos.