Desenvolvimento de Sistemas

Lean code: princípios e aplicabilidade

Os *softwares* estão dominando o mundo todo, e as empresas que lideram o setor estão cada vez mais entendendo a importância do desenvolvimento de *softwares*.

Os aplicativos de *software* modernos são uma orquestração complexa de infraestrutura, *pipelines* de construção e implantação, filtragem e processamento de dados, notificações, experiência do usuário, tratamento de erros e muito mais, sem mencionar a necessidade inerente de que todos os aspectos de qualquer integração sejam seguros. Cada um desses itens pode ter vários requisitos explícitos.

Assim, a maior preocupação deve ser “escrever” um código que seja legível, enxuto, de fácil interpretação e que agregue valor, e é aí que entram a metodologia *lean* e todos os seus recursos.

Metodologia *lean*

Qualquer um concordaria que uma empresa precisa eliminar o desperdício e a ineficiência para gerar lucro, mas muitos ainda não adotam totalmente tal princípio.

A metodologia *lean* (metodologia enxuta) oferece um método para as empresas **minimizarem o desperdício**, implementando processos contínuos de *feedback*, revisão e aprendizado para aumentar a eficiência, ou seja, para torná-las “*lean*”.

O objetivo é fornecer os serviços mais valiosos, econômicos e com os melhores preços, para que os clientes fiquem satisfeitos.

O que é metodologia *lean*?

A metodologia *lean* surgiu originalmente no Japão, no Toyota *production system*. Atualmente, já é aplicada em diversas empresas e em outros ambientes orientados ao conhecimento em todo o mundo. A metodologia é tanto uma filosofia quanto uma disciplina que, em essência, aumenta o acesso à informação para garantir a tomada responsável de decisões a serviço da criação de valor.

Fundamentos da metodologia *lean*

Embora as pesquisas sobre a metodologia enxuta tragam imediatamente a noção de “eliminação de desperdícios”, essa não é a definição completa. Fundamentalmente, o método enfatiza a ideia de “melhoria contínua”. Os pensadores da metodologia *lean* que a trouxeram do Japão para o Ocidente (especificamente James Womack e Daniel Jones) especificaram cinco princípios fundamentais:

- **1. Valor:**
  entender o que os clientes valorizam em um produto ou um serviço.
- **2. Fluxo de valor:**
  identificar o que é necessário para maximizar o valor e eliminar o desperdício em todo o processo, desde o *design* até a produção.
- **3. Fluxo:**
  garantir que todos os processos do produto fluam e se sincronizem perfeitamente.
- **4. Puxar:**
  saber que o fluxo é possibilitado pelo “puxar”, ou seja, pela ideia de que nada é feito antes de ser necessário, criando assim ciclos de entrega mais curtos.
- **5. Perfeição:**
  buscar incansavelmente a perfeição, engajando-se constantemente no processo de solução de problemas.

A ideia é refinar ao máximo os processos internos para dar ao consumidor o maior valor possível em um produto ou um serviço. Qualquer coisa que não contribua para o valor do produto ao cliente é considerada ineficiente.

Outra chave para a metodologia *lean* é a sua definição acerca de desperdício. Existem oito tipos:

Movimento

Consiste na movimentação desnecessária de pessoas ou processos (equipamentos e máquinas de fabricação, por exemplo), nos movimentos repetitivos que não agregam valor e se traduzem em desperdício de tempo e recursos.

Processamento excessivo

Trata-se da execução de processos ou etapas desnecessários para criar um produto valioso.

Extraprocessamento

Consiste nos produtos que exigem mais trabalho ou qualidade do que o necessário para entregar valor ao cliente.

Defeitos

São os processos de fabricação que criam produtos defeituosos, que se tornam materiais desperdiçados.

Transporte

É semelhante ao movimento, mas abrange distâncias maiores para incluir o transporte de ferramentas, estoque, pessoas ou produtos além do necessário.

Potencial humano

Refere-se às habilidades e aos talentos subutilizados devido à má gestão de funcionários e de estrutura de equipe que leva à falta de moral e produtividade.

Espera

Trata-se dos equipamentos ociosos e que aguardam materiais ou dos equipamentos que podem retardar os processos e a eficiência.

Estoque

Refere-se à ocupação de espaço com produtos e estoques excessivos, revelando superprodução e criando *backwork*.

É fácil ver como a melhoria contínua é sempre possível e inclui todos os níveis da empresa, desde gerenciamento de talentos, fabricação, tecnologia da informação (TI) até *marketing*.

Lean software development

Com base no modelo *lean* de manufatura, a comunidade de desenvolvimento de *software* tratou de adaptar seus princípios e suas práticas à rotina de programação de sistemas. A ideia segue a mesma: otimizar a eficiência nos processos e minimizar o desperdício.

O *lean software development* (ou desenvolvimento de *software* enxuto) se baseia em sete princípios, também derivados da metodologia clássica *lean*:

1. Elimine o desperdício.

Regularmente, os desenvolvedores discutem “gargalos” no processo de desenvolvimento e identificam desperdícios, montando um plano para eliminá-los. São exemplos de desperdício: código desnecessário, mais tarefas previstas do que a capacidade da equipe, processos burocráticos, questões de qualidade.

2. Construa com qualidade.

Incorpore técnicas como TDD (*test driven development*) e programação em pares para garantir a qualidade do *software*.

3. Compartilhe o aprendizado.

O aprendizado de um membro da equipe deve ser compartilhado com o restante do time, seja por meio de reuniões, seja por meio de revisão de código.

4. Adie o compromisso.

Realize os testes necessários antes de se comprometer com o cliente em alguma ação.

5. Entregue rapidamente.

Lance o produto (o sistema, no caso) o quanto antes, para que os *feedbacks* do cliente também cheguem o quanto antes e a melhoria possa ser incorporada.

6. Respeite as pessoas.

Para uma equipe colaborativa e produtiva, a metodologia *lean* incentiva as discussões produtivas (e saudáveis), a comunicação e o *feedback* constante.

7. Otimize o todo.

Avalie o processo como um todo, do início ao fim, para tornar o fluxo de tarefas o mais eficiente possível.

O método *lean* e as metodologias ágeis compartilham várias características, especialmente no que se refere à comunicação, podendo ser usados em conjunto. A principal diferença está no foco: o método *lean* é usado para construir **processos** melhores, enquanto as metodologias ágeis visam à construção de **produtos** melhores.

Alguns desafios na aplicação da metodologia *lean* de desenvolvimento de *software* são:

- Necessidade de treinamento da equipe para a rotina proposta, o que pode ser complicado em algumas equipes menos comunicativas ou menos colaborativas
- Definição de métricas de desenvolvimento para identificar gargalos e desperdícios
- Delimitação de escopo, a qual é necessária pela flexibilidade de inclusão de novas funcionalidades permitida pela metodologia (é necessário ter consciência de até onde o *software* deve crescer)
- Subotimização, ou boa otimização de um elemento contra má otimização de outro elemento presente no projeto, o que acaba afetando o projeto como um todo

É importante que a empresa confie na equipe de desenvolvimento e na capacidade de colaboração desta para que a aplicação da *lean* seja bem-sucedida. O benefício mais evidente é o corte de atividades que não afetem diretamente o produto final, reduzindo tempo e custos no projeto de *software*.

Lean code

O *lean code* (código enxuto), no seu sentido mais puro, é uma metodologia que visa a organizar as atividades humanas para entregar mais valor e eliminar desperdícios. As pessoas adotaram muitas abordagens diferentes para definir a metodologia *lean*, mas cada uma dessas disciplinas, embora possam ter nomes diferentes, presta alguma atenção nos princípios *lean*.

Assim, para organizar e ser disciplinado, primeiramente você precisa visualizar o que quer organizar e logo seguir uma disciplina para fazer a organização.

Clean code

É preciso agora definir "*clean*", e, para tanto, cabe voltar aos dias em que você costumava ouvir da sua mãe que, antes de qualquer coisa, você deveria limpar seu quarto para depois poder sair e jogar com seus amigos. Por incrível que pareça, antigamente as pessoas literalmente saiam de suas casas para jogar. Hoje em dia, podem jogar *on-line* a qualquer hora.

Então, quando você começa a limpar seu quarto, percebe que o primeiro passo para a limpeza ser eficaz é organizar toda a bagunça, colocando, de um lado, os videogames; de outro, as revistas; de outro, os brinquedos etc., atribuindo um lugar para cada grupo classificado. Agora você pode limpar mais facilmente cada coisa.

Finalmente, você percebe que é mais fácil criar um conjunto de regras sobre como as coisas devem ser tratadas em seu quarto (padronizar) para preservá-lo limpo (sustentar), permitindo que você saia rapidamente e jogue à vontade, sem precisar fazer alguma limpeza antes.

Analogamente, é preciso estar preocupado com a criação, o desenvolvimento e o suporte em códigos e estruturas de programação. O *clean code* (código limpo) é facilmente legível, de fácil manutenção e menos propenso a erros.

Por que é preciso se preocupar com o código limpo?

Existem inúmeras respostas em particular, mas a mais adequada é: deve haver uma preocupação com o código limpo porque ele eventualmente mostra aos programadores a capacidade de escrever um código de boa manutenção (manutenível), que é facilmente entendido por outros desenvolvedores que trabalharão com o código.

Na vida real, quando um programador trabalha em uma base de código, há uma grande chance de que, em algum momento, outros programadores também trabalhem na mesma base de código. Portanto, é muito importante para o programador escrever um código limpo, pois eventualmente será muito mais fácil para outros programadores lidarem com esse código e, mais precisamente, será muito mais fácil solucionar erros que possam aparecer em um futuro próximo.

Escrever um código limpo reflete a capacidade do programador de escrever um código que facilite a vida dele e a de outros programadores.

Características do código limpo

Agora que você já sabe o que é código limpo e por que precisa se preocupar com ele, é hora de discutir quais são as características de um código limpo. Como existem características para qualquer coisa na Terra, é possível definir que as do computador são velocidade, memória, precisão, confiabilidade, e assim por diante.

Existem também características de código limpo:

- **Simples**: o código deve ser o mais simples possível.
- **Manutenível**: o código deve ser sustentável a longo prazo, pois muitos desenvolvedores diferentes podem trabalhar nele.
- **Testável**: o código deve ser facilmente testável e menos propenso a erros.
- **Legível**: por último, mas não menos importante, o código deve ser facilmente legível.

Tais características diferenciam um *clean code* de um *bad code* (código ruim).

Aliás, você sabe o que é um código ruim?

Pode não haver nenhuma resposta específica, pois ela depende da perspectiva do programador. Entretanto, pode haver várias respostas para a mesma pergunta, pois os programadores podem ter diferentes perspectivas.

Para ser mais específico, qualquer código que leve a problemas a longo prazo pode ser chamado de “código inválido”. Os problemas podem ser:

- Código difícil de ler
- Código desnecessariamente complexo
- Código que não é facilmente testável
- Código difícil de modificar

Código ruim não é facilmente legível e é desnecessariamente complexo.

Como escrever um código limpo em Java?

Então, finalmente, chegou a hora de aprender a escrever um código limpo em Java. Java é uma linguagem de programação muito usada, que evoluiu e segue evoluindo muito, e várias empresas grandes de tecnologia utilizam a linguagem Java para suas operações.

Os pontos mencionados a seguir são de extrema importância para obter uma codificação limpa em Java. Sendo assim, aproveite a oportunidade e aprofunde os seus conhecimentos.

1. Estrutura

O primeiro e mais importante passo é a estrutura do código. Em Java, não existe a regra de seguir uma estrutura específica para o projeto, o que não significa que você não deva.

Na verdade, é uma boa prática seguir uma estrutura de projeto específica. Caso você ainda não saiba, em palavras simples, a estrutura do projeto é a maneira de organizar diferentes utilitários de projetos, como arquivos de origem, dados, configurações, testes etc. Deve-se sempre seguir um padrão para as estruturas de pastas de um projeto. Observe o exemplo:

Figura 1 – Estrutura de pastas
Fonte: Senac EAD (2022)

O Maven sugere essa estrutura de pastas, que deve ser criada e incluída na estrutura do projeto. Veja com mais detalhes:

1. **src/main/java** – Contém os arquivos de fonte do projeto. Você pode identificá-los como a subpasta principal neste caso.
2. **src/main/resources** – Contém os arquivos de recursos do projeto. Você pode identificá-los como a subpasta principal também neste caso.
3. **src/test/java** – Contém os arquivos de fonte de teste do projeto. Você pode identificá-los como a subpasta de teste neste caso.

4. **src/test/resources** – Contém os arquivos de recursos de teste do projeto. Você pode identificá-los como a subpasta de teste neste caso também.

Se não for o Maven, você pode escolher entre outras estruturas de projeto disponíveis para Java de acordo com as suas necessidades.

2. Nomeação adequada

Qualquer linguagem de programação existente tem algum conjunto de regras que você precisa seguir para criar e nomear identificadores. Portanto, o Java também tem um conjunto de regras que precisa ser seguido. Ao seguir as regras de nomenclatura, devem-se também nomear os identificadores que mostram algum significado (que se relacionam com a variável).

Quando se nomeia relacionado conteúdo de uma variável, o código se torna muito mais legível, o que eventualmente ajuda na manutenção do código a longo prazo. Em Java, é preciso nomear variáveis, classes, métodos etc., sempre dando a eles um nome com o qual possam se relacionar.

Por exemplo, se você deseja criar uma variável para contagem, nomeie-a **contador** em vez de nomeá-la **c** ou algo do tipo. Você deve sempre evitar nomear variáveis com um único caractere, como **a**, **b**, **c** etc.

Observe que, no exemplo, existem variáveis com letras, causando certa confusão de entendimento, mesmo que o código seja bem pequeno:

```
import java.util.Scanner;

public class SomaNumeros {

    public static void main(String[] args) {

        Scanner entrada = new Scanner(System.in);
        int a;
        int b;
        int c;
        System.out.println("Digite o primeiro número: ");
        a = entrada.nextInt();
        System.out.println("Digite o segundo número: ");
        b = entrada.nextInt();
        c = a + b;
        System.out.println("A soma dos numeros e: " + c);
        System.exit(0);
    }
}
```

Já neste outro exemplo, as variáveis foram renomeadas, ficando explícita a função de cada uma:

```
import java.util.Scanner;

public class SomaNumeros {

    public static void main(String[] args) {

        Scanner entrada = new Scanner(System.in);
        int num1;
        int num2;
        int soma;
        System.out.println("Digite o primeiro número: ");
        num1 = entrada.nextInt();
        System.out.println("Digite o segundo número: ");
        num2 = entrada.nextInt();
        soma = num1 + num2;
        System.out.println("A soma dos numeros e: " + soma);
        System.exit(0);
    }
}
```

Revise seus códigos desde o início do curso e verifique se as variáveis utilizadas estão no padrão correto.

3. Estrutura do código-fonte

A estrutura do código-fonte pode conter muitos tipos diferentes de elementos dentro dela. Pode-se seguir qualquer tipo de estrutura do código-fonte disponível ou ainda criar a própria estrutura, se necessário. Normalmente, os elementos colocados no código-fonte estão na seguinte ordem:

- Declaração do pacote
- Importação de declarações
- Todas as importações estáticas
- Todas as importações não estáticas
- Exatamente uma classe de nível superior
- Variáveis de classe
- Variáveis de instância
- Construtores
- Métodos

Veja um exemplo de uma estrutura do código-fonte típica que é bem formada:

```
package com.scaler.application;

import java.util.Date;

public class Employee {
    private String employeeName;
    private Date joiningDate;
    public Employee(String employeeName) {
        this.employeeName = employeeName;
        this.joiningDate = new Date();
    }

    public String getEmployeeName() {
        return this.employeeName;
    }

    public Date getJoiningDate() {
        return this.joiningDate;
    }
}
```

4. Espaços em branco e recuo

Os espaços em branco podem ajudar a tornar o código muito legível, e, por isso, é aconselhável adicioná-los sempre que possível (devem-se evitar espaços em branco onde eles não são necessários). Geralmente, a identação ajuda a tornar o código menos propenso a erros, pois se torna mais fácil de ser lido e compreendido.

Por exemplo, nesta função, os espaços em branco e o recuo não são atendidos:

```
public int sum(int num1,int num2,int num3){
int sum=num1+num2+num3;
return sum;}
```

Já nesta função, os espaços em branco e o recuo são devidamente cuidados:

```
public int sum (int num1, int num2, int num3) {
  int sum = num1 + num2 + num3;
  return sum;
}
```

Revise seus códigos desde o início do curso e verifique se todos têm espaços e recuo.

5. Parâmetros do método

Na maioria das vezes, são criados métodos que têm parâmetros, e às vezes são adicionados tantos parâmetros a um único método que uma confusão acaba sendo gerada.

O objetivo é simples: evite adicionar mais de três parâmetros, para não criar uma confusão desnecessária para outros desenvolvedores ou programadores que lidarão com o código no futuro. Você também pode "refatorar", se possível, como no exemplo a seguir.

Em programação, **refatoração** é uma técnica usada para reestruturar um código existente sem modificar o comportamento externo do código:

```
  // antes
  private void employeeDetails(String name, String age, String awards, Stri
ng ctc, String experience).
  // depois
  private void employeeDetails(String name, Details employeeDetails).
```

6. Codificação

De forma simples, *hardcoding* é a prática de adicionar os dados diretamente no código-fonte de um programa em vez de obter os dados de fontes externas. Lembre-se de que uma *hardcoding* pesada pode, além de levar a *bugs* na sua programação, tornar difícil manter seu programa.

Por exemplo, se você criar um programa que calcule os dias e o tempo restantes para a chegada do novo ano, você precisa da data e da hora atuais para tanto. Por isso, em vez de codificar a data e a hora atuais, você pode obter a data atual de várias classes em Java que fornecem a data e a hora atuais para que seu programa precise ser modificado para cada data.

7. Comentários do código

Muitos programadores ou desenvolvedores experientes recomendam adicionar comentários no código para que você ou qualquer outro desenvolvedor possa entender o funcionamento desse código no futuro.

Comentar também é uma das melhores práticas para tornar o código facilmente compreensível. Observou-se que, inicialmente, alguns desenvolvedores não seguem a prática de comentários e, depois de algum tempo, quando veem aquele código não comentado, acham muito difícil entendê-lo.

Em Java, dois tipos de comentários são permitidos: **comentários de documentação** e **comentários de implementação**. Veja-os em detalhes:

Documentação/comentários JavaDoc

O público-alvo são os usuários da *code base* (base de código). Por exemplo:

```
/**
  * Descrição geral da classe.
  *
  * @javadoc
  */
public class Exemplo { ...
```

Os detalhes adicionados aqui são geralmente livres de implementação e se concentram principalmente nos propósitos de especificação.

Comentários de implementação/bloqueio

O público-alvo são os desenvolvedores que trabalham na base de código. Por exemplo:

```
if (a == 2) {
  return TRUE;           /* caso especial */
} else {
  return numeroPrimo(a); /* funciona apenas para os ímpares */
}
```

Os detalhes adicionados aqui são geralmente específicos para implementação. É uma boa prática usar comentários JavaDoc para a maioria das classes, das interfaces, dos métodos públicos e protegidos etc.

Revise seus códigos desde o início do curso e verifique se você adicionou comentários explicando brevemente o que está sendo programado. Se não houver comentários, o que acha de começar a trabalhar com eles?

8. Registro

O registro em *log* (um tipo de arquivo, geralmente em **.txt**, que pode ser utilizado de diversas maneiras para acompanhar todas as ações executadas em *softwares* ou sistemas operacionais) é um dos fatores mais importantes na codificação limpa.

Um código bem registrado é fácil de depurar e pode economizar horas de trabalho de um desenvolvedor. Deve-se sempre tentar evitar o *log* excessivo, para evitar uma diminuição do desempenho.

Agora que você viu como pode escrever um código limpo em Java, segue um exemplo:

```
public class Employee {
    private String employeeName;

    public String getEmployeeName(){
        return this.employeeName
    }
}
```

Note que, no exemplo dado, criou-se uma classe pública chamada de **Employee**, com a primeira letra em maiúsculo, como deve ser. Então foram criados uma variável de *string* privada chamada de **employeeName** e um método público chamado de **getEmployeeName**, que retorna a *string* **employeeName**.

Esse código é, então, um código limpo?

Sim, devido às seguintes características:

- Variável, classe e método são nomeados de acordo com a convenção de nomenclatura. Como a classe tem a primeira letra em maiúsculo, variável e método são nomeados de acordo com a convenção de nomenclatura **camelCase** e também refletem um significado específico de acordo com seu nome.
- Garantiu-se que o método está fazendo apenas uma coisa: obter o nome do funcionário.
- Espaços em branco e recuo também são utilizados adequadamente.

Ferramentas para escrever um código limpo

Além das várias práticas e dos princípios disponíveis que podem ser usados para escrever um código limpo, existem certas ferramentas que também podem ajudar você a escrevê-lo se utilizadas de forma eficaz, tais como formatadores de código para formatar o código (pôr espaços em branco e recuo padrão) e algumas ferramentas de análise estática.

Formatadores de código

A formatação adequada é essencial ao escrever um código, e os editores de código populares para Java já têm formatação automática de código. Por causa dessa formatação automática, os programadores não precisam pensar em formatar o código etc. Se você não quiser a formatação de código padrão disponível no editor de código, pode personalizá-la facilmente de acordo com os requisitos que desejar.

No IDE (*integrated development environment*) do NetBeans 13, é possível utilizar o recurso de formatação de maneira rápida e simples, no menu de navegação superior:

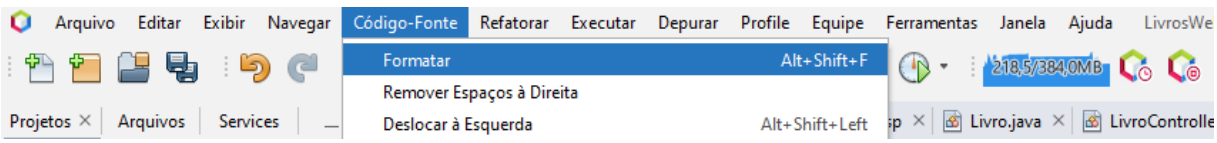

Figura 2 – Menu de navegação
Fonte: Senac EAD (2022)

Basta clicar em **Código-Fonte** e depois em **Formatar**, e assim toda a classe aberta será formatada.

Ferramentas de análise estática

Para Java, existem muitas ferramentas de análise estática de código disponíveis, como Checkstyle, SonarQube e SpotBugs, que oferecem um conjunto de regras que pode ser aplicado em projetos.

Encerramento

Você aprendeu neste material que o desenvolvedor não pode simplesmente ir elaborando e construindo os códigos e que deve sempre estar preocupado e ter cuidados com a formatação, a disposição e a legibilidade do código.

Quando se inicia um projeto, é imprescindível deixar tudo bem orquestrado para as próximas manutenções e/ou implementações que se fizerem necessárias.