Programação Orientada a Objetos II

Programação Orientada a Objetos II

Fabio Andrijauskas

Dados Internacionais de Catalogação na Publicação (CIP)

Andrijauskas, Fabio
A573p Programação orientada a objetos II / Fabio Andrijauskas. – Londrina : Editora e Distribuidora Educacional S.A., 2018.
240 p.

ISBN 978-85-522-1166-2

1. Orientação a objetos. 2. Java. 3. Manipulação de eventos. I. Andrijauskas, Fabio. II. Título.

CDD 001.64

Thamiris Mantovani CRB-8/9491

2018
Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: http://www.kroton.com.br/

Sumário

Palavras do autor

No início da computação, o software tinha uma importância menor do que o hardware. O valor de um sistema computacional era mensurado pelo custo de produção e instalação do computador; os custos do software eram diluídos no preço de uma arquitetura específica de hardware. Porém, com o avanço da tecnologia, os computadores começaram se tornar elementos mais comuns e o software recebeu o crédito necessário. Todavia, as formas de programação eram prematuras e as padronizações, necessárias para levar a paradigmas e características mais ligadas à regra de negócio. Assim surgiu a orientação a objetos, e com ela outros avanços em interfaces gráficas, abstração e outros elementos. Portanto, o estudo de métodos e técnicas mais especializadas como as que serão vistas nessa disciplina são essenciais para acompanhar o exigente mercado de trabalho relacionado ao desenvolvimento de software.

Em um primeiro momento, o objetivo está direcionado à criação de interfaces gráficas e acesso a banco de dados relacional seguindo as regras da orientação a objetos. A programação de interfaces leva a tarefas que necessitam da possibilidade de o computador efetuar mais de uma ação por software. Utilizando as interfaces gráficas e programação concorrente é comum encontrar certas porções de código que podem ser feitas de maneira padronizada, necessitando de ferramentas de controle de código e ainda de metodologias que possam contribuir para o desempenho da equipe. Para isso, abordaremos formas de tratar esses eventos e aplicar técnicas, ferramentas e metodologias de maneira efetiva. Além dessas capacidades, é importante vislumbrar novas técnicas para armazenamento de dados.

Para atingir essas metas de capacidade, serão abordadas na primeira unidade de ensino as formas de criação de interfaces gráficas, apresentando seus componentes para inserção de texto, número, imagens e outros. Com esses componentes estudados será possível verificar as formas de tratar os eventos relacionados aos componentes gráficos para gerar interfaces dinâmicas. A segunda unidade de ensino tem o objetivo de inserir os conceitos de programação concorrentes e tratamento de exceções, gerando

a capacidade de executar diversas tarefas ao mesmo tempo em um software e aumentar a disponibilidade do programa tratando os erros que podem ocorrer. Como a maneira de desenvolver softwares evolui rapidamente, na terceira unidade serão apresentadas diversas ferramentas para automatização de testes, controle de versão de código e para documentação e padronização de códigos. Todavia, ainda é necessário procurar maneiras de tratar o grande volume de dados que temos atualmente. Por isso, a última unidade também tratará das formas de acesso, inserção, alteração e remoção de dados em um banco de dados não relacional.

Todos esses tópicos são de grande valia para gerar conhecimento e expertises mais próximas do mercado de trabalho, lugar em que as técnicas e metodologias explicadas são capazes de gerar proximidade com equipes de desenvolvimento já estabelecidas, sendo a curva de aprendizados menor quando inserida em cenários comercias.

Programação orientada a eventos com interfaces gráficas e banco de dados relacional

Convite ao estudo

Os sistemas computacionais são elementos que estão presentes no dia a dia, sendo cada vez mais difícil encontrar qualquer processo ou atividade que não seja informatizada. Dada essa grande abrangência, é sempre necessário observar o avanço nas tecnologias que são empregadas nesses sistemas informatizados. Como exemplo, podemos pensar em sistemas que foram desenvolvidos 30 ou 40 anos atrás, conhecidos como sistemas legados. Tais sistemas legados possuem características como: interface gráfica limitada ou inexistente, sistemas de banco de dados antiquados e que apresentam dificuldades de manutenção. Dos elementos citados, um dos mais marcantes para o usuário é questão da interface gráfica. A falta de elementos visuais e a utilização de terminais de texto nesses sistemas é impactante, pois hoje temos telas de alta resolução e com diversos elementos em celulares, tablets e outros.

Após um longo processo de seleção, você foi contratado por uma empresa especializada na migração e na adequação de sistemas computacionais legados. Sua empresa conseguiu vencer uma licitação, e agora tem um contrato para migrar um sistema legado de uma universidade. Foram designadas a você três tarefas no desenvolvimento do sistema. Em primeiro lugar, será necessária a produção da interface com diversos componentes gráficos adequados aos dados que serão inseridos. A segunda etapa consiste em tratar os eventos que o usuário gera (como clicar em botões ou fazer seleção), e a terceira tarefa é fazer a gravação em um sistema de banco

de dados. Dessa forma, quais são os componentes gráficos que são relacionados aos dados da universidade? Quais são as formas corretas relacionadas à orientação a objetos que devem ser empregadas para tratar os eventos da interface gráfica ou utilizadas para gravar os dados no banco de dados?

Nessa unidade, as competências propostas consistem em conhecer e compreender a programação orientada a eventos, a modelagem de interfaces gráficas e o uso de banco de dados relacional. Nesse processo, a modelagem de interfaces gráficas requer criatividade para utilizar diversos componentes gráficos (por exemplo, botões, caixas de seleção e outros), juntamente com o raciocínio crítico para empregar a correta orientação a objetos no acesso aos bancos de dados relacionais. Ao conhecer e aplicar os conceitos, você poderá utilizar os recursos de programação orientados a eventos, com interface gráfica e persistência de dados em uma base relacional.

Bons estudos!

Seção 1.1

Desenvolvimento de interfaces gráficas na linguagem Java

Diálogo aberto

Imagine um sistema capaz de processar seus cadastros rapidamente e gerar relatórios, mas que apresente telas de difícil utilização e cujos tipos de dados não reflitam os componentes gráficos de inserção. Percebe-se que a interface gráfica de um sistema é um dos elementos mais impactantes para a aceitação de um sistema pelo cliente e validação do mesmo pelos usuários. Portanto, ter conhecimento das formas de inserir os diversos tipos de componentes é essencial para a produção de interfaces gráficas de maneira correta, de acordo com a necessidade de uma empresa que preza pela qualidade.

Depois de um processo de seleção contendo provas práticas e teóricas, você conseguiu a vaga de Programador Júnior, vaga essa de grande interesse. A empresa pela qual você foi contratado recebeu um pedido de uma universidade para o desenvolvimento de uma interface gráfica para um sistema de matrícula. Você está na equipe de desenvolvimento do *front-end*, e foi destacado para atender essa demanda. Esse sistema de matrícula já possui cerca de 30 anos e ainda não passou pelo processo de modernização. O líder da sua equipe pediu para que você faça o código que apresente os componentes visuais principais da tela de cadastro de aluno, e essa tela deve conter os seguintes dados do universitário:

- Nome (*label* e campo de texto).
- Endereço (*label* e campo de texto).
- Código de área e telefone (*label* e campo de texto com máscara de formatação).
- Tipo sanguíneo e fator Rh (*label* e caixa de seleção).
- Curso atual (*label* e caixa de seleção).
- Nome e telefone do contato de emergência (*labels*, caixa de texto e caixa de texto com máscara de formatação).

- Botões para inserir e cancelar.

Para efetuar essa entrega, será necessário fazer a definição e tamanhos da janela, bem como posição e tipos de componentes. Todos os tipos de dados devem ser relacionados aos componentes que serão utilizados. Além dessa tela, deve-se preparar um diagrama que contenha uma imagem da tela e a indicação de quais componentes gráficos do Java foi utilizado. Lembre-se, a interface gráfica de um sistema é a impressão que o usuário terá de todo o sistema. Os códigos devem possuir comentários e variáveis com nomes que descrevam qual é sua função, além de estar organizado. Com os elementos de qualidade de código, rapidamente você receberá novas tarefas e subirá na carreira.

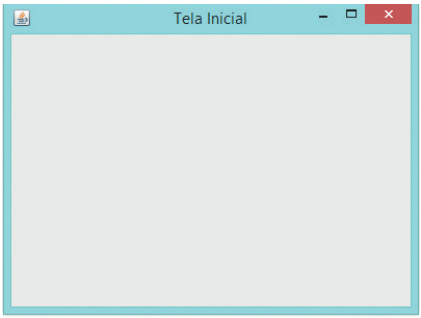
Não pode faltar

A criação de interfaces gráficas, também chamado de *Graphical User Interface* (GUI), é um processo que necessita de grande dedicação, pois por meio delas o usuário percebe e interage com o sistema. Dessa forma, em todo o processo de criação de uma interface é necessário seguir as orientações de boas práticas na utilização de componentes, inclusive quanto ao posicionamento e agrupamento (DEITEL; DEITEL, 2016). Portanto, nessa etapa de estudo você aprenderá sobre a criação inicial de uma interface gráfica utilizando as bibliotecas do Java.

A construção de interfaces gráficas é feita a partir da inserção de componentes em uma tela, como por exemplo, botões, caixas de seleção e a própria representação da tela, dentre outros. Todos esses componentes estão disponíveis dentro de bibliotecas específicas.

No caso do Java, utilizaremos o Swing (FURGERI, 2015). O Swing é uma biblioteca que vem incorporada no Java Development Kit, dispondo de diversos elementos para a produção dessas telas (HORSTMANN, 2016). O primeiro elemento a ser utilizado em uma interface gráfica é a representação de uma área que apresenta uma barra de título e um espaço reservado para se adicionar componentes. No caso do Java com Swing, a classe que faz essa representação é o `JFrame` (MANZANO; COSTA, 2014). A Figura 1.1 exibe um exemplo da interface criada por essa classe.

Figura 1.1 | Exemplo de frame ou tela utilizando o Swing do Java



Fonte: captura de tela do software Java.

Na Figura 1.1 é possível reparar uma barra no topo da tela com o texto "Tela Inicial", os botões para minimizar, maximizar e fechar e uma área cinza para adicionar os componentes. Todos esses elementos de tela são passíveis de alteração e todas essas modificações podem ser feitas por código em Java sem a necessidade de utilizar algum editor gráfico, todavia, existem editores para a criação de telas utilizando o Swing. No caso do *Integrated Development Environment* (IDE) Eclipse, existem *plugins* e extensões que propiciam esse editor (WINDER, 2009).

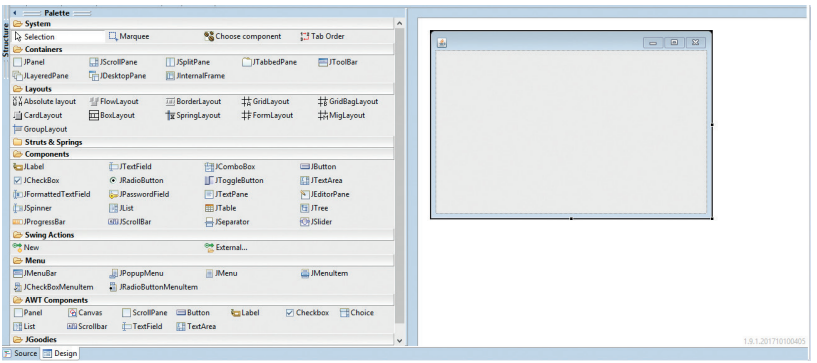
Reflita

Existem editores "*what you see is what you get*" (WYSIWYG) para a produção de interfaces gráficas. Esses editores apresentam uma interface gráfica na qual existe a possibilidade de selecionar um componente gráfico com o mouse e arrastar para a posição desejada na interface gráfica que está sendo montada. Esse tipo de editor propicia um dinamismo maior na produção de uma interface, porém é necessário atentar-se ao fato de que ao fazer a simples ação de colocar um botão em uma interface, inicia-se um processo automático de criação de código pelo editor. Com isso, como é possível mensurar os ganhos de tempo comparados à possibilidade de manutenção mais complexa no código?

Um dos editores gráficos acoplados ao Eclipse é o Windows Builder, que fornece diversas ferramentas e formas de criar a janela (WINDER, 2009). A Figura 1.2 apresenta uma visão desse editor, no

qual os itens do lado esquerdo, "Palette", são todos os componentes disponíveis para inserção na tela, e a direita é a representação da própria tela que receberá os componentes. Uma das grandes vantagens é a capacidade de, apenas com o cursor do mouse, fazer a inclusão de componentes na tela, o que proporciona aumento de velocidade no desenvolvimento.

Figura 1.2 | Editor para interfaces gráficas acoplado ao Eclipse



Fonte: captura de tela do software Eclipse.

Mas mesmo com essas ferramentas é necessário entender como os componentes são alocados na tela e como são feitos os processos de posicionamento, controle de eventos e outros, pois em caso de manutenção ou outros problemas você será capaz de fazer as correções sem depender de uma ferramenta específica. Além disso, muitos componentes precisam ser criados em tempo de execução, e isso só é possível com a programação.

O Quadro 1.1 apresenta o código necessário para produzir a tela da Figura 1.1.

Quadro 1.1 | Código para gerar a tela representada na Figura 1.1

```
1.  package view;
2.  import javax.swing.JFrame;

3.  public class InterfaceGrafica extends JFrame{
4.     public InterfaceGrafica(){
5.          setSize(400,500);
6.          setTitle("Tela Inicial");
7.          setVisible(true);
8.     }
9.     public static void main(String[] args) {
10.         InterfaceGrafica telaInicial = new Interfa
    ceGrafica();
11.    }
12. }
```

Fonte: elaborado pelo autor.

A linha 1 do Quadro 1.1 faz referência ao pacote (conjunto de classes que têm funções semelhantes). Sobre os *package* é sempre interessante manter as classes de interface no mesmo pacote para manter a alta coesão e baixo acoplamento. A coesão demonstra que as classes, pacotes e métodos têm funções delimitadas, relacionando apenas a um propósito, e o alto acoplamento refere-se ao funcionamento independente de outras classes para executar sua função (DEITEL; DEITEL, 2016). A linha 2 apresenta quais pacotes terão de ser incluídos no seu código para que se possa utilizar as classes do Swing; nesse caso foi necessário incluir somente a `JFrame`, classe à qual pertence a tela da Figura 1.1. A linha 3 apresenta a classe que foi criada para representar a tela, sendo que ela especializa (`extends`) a classe `JFrame`. A linha 4 é o construtor da classe InterfaceGrafica, no qual são feitas as configurações da tela que é criada nessa classe. O comando `setSize()`, na linha 5, define a largura e altura da janela, o comando `setTitle()` define o título da tela e, por fim, o comando `setVisible()` define que a tela é visível. As linhas de 9 a 11 são as funções iniciais (método *main*) do programa que utiliza a classe que representa a tela.

Para se criar uma interface gráfica são necessários cinco passos (HORSTMANN, 2016, DEITEL; DEITEL, 2016):

1. Criar uma relação de especialização com a classe que representa sua tela.
2. Declarar como atributos os elementos que serão adicionados à tela.
3. Definir a forma de alocação dos elementos gráficos na tela.
4. No construtor, instanciar, configurar e posicionar os itens na tela.
5. Tratar os eventos dos componentes para tratar as ações do usuário com a interface gráfica.

Os componentes que farão parte da interface gráfica podem ser configurados e posicionados na tela através dos atributos e métodos. Por exemplo, para inserir uma legenda e um campo para o usuário digitar seu nome, utilizam-se os componentes `JLabel` e `JTextField`, conforme código do Quadro 1.2. O primeiro passo para a inserção desses elementos consiste em importar as respectivas bibliotecas (linhas 2 a 4). Na linha 5, a classe `PrimeiraTela` herda os recursos da classe `JFrame` através da especialização feita com o comando `extends`. Nas linhas 6 e 7 são declarados dois componentes, sendo um do tipo `JLabel` e outro `JTextField`. Na linha 8 é feita a alteração do construtor da classe para que a tela seja desenhada. Nas linhas 9 e 10 os componentes criados anteriormente devem ser instanciados. As linhas 11 e 12 especificam o tamanho da janela e configuram um título para ela; na linha 13 é usado o método para deixar a janela visível e na linha 14 o layout é "zerado" (absoluto), para que os componentes possam ser dispostos conforme as próximas configurações. Os comandos das linhas 15 e 16 são justamente os que posicionam e dimensionam os elementos na tela. O método `setBounds()` especifica quatro parâmetros: posição horizontal (x), posição vertical (y), largura e altura. Por fim, nas linhas 17 e 18 os componentes são adicionados à tela criada, através do encadeamento dos métodos `getContentPane().add()`;

Quadro 1.2 | Interface gráfica em Java com `JLabel` e `JTextField`

```
1.  package view;
2.  import javax.swing.JFrame;
3.  import javax.swing.JLabel;
4.  import javax.swing.JTextField;

5.  public class PrimeiraTela extends JFrame {
6.          private JLabel lblNome;
7.          private JTextField txtNome;

8.  public PrimeiraTela() {
9.          lblNome = new JLabel("Nome");
10.         txtNome = new JTextField();

11.         setSize(400,200);
12.         setTitle("Tela Inicial");
13.         setVisible(true);
14.         setLayout(null);

15.         lblNome.setBounds(10,10,100,25);
16.         txtNome.setBounds(50,10,200,25);
17.         getContentPane().add(lblNome);
18.         getContentPane().add(txtNome);
19.    }
20.    public static void main(String[] args) {
21.         PrimeiraTela t1 = new PrimeiraTela();
22.    }
23. }
```

Fonte: elaborado pelo autor.

Segundo Deitel e Deitel (2016) os objetos do tipo `JLabel` devem ser usados para representar textos estáticos em interfaces. Já para receber entradas de texto sem formatação, Horstmann (2016) indica o uso de objetos do tipo `JTextField`. Para campos de entrada que

precisam de formatação específica, como, por exemplo, um CPF, o objeto `JFormattedTextField` pode ser usado para especificar máscaras. O Quadro 1.3 apresenta alguns componentes, disponíveis na biblioteca Swing, que são muito utilizados para a construção de interfaces gráficas. Tanto os componentes do Quadro 1.3 como diversos outros serão explorados ao longo de todo livro.

Quadro 1.3 | Componentes básicos da biblioteca Swing

Componente	Descrição
`JButton`	Objeto usado para criar botões.
`JCheckBox`	Objeto usado para oferecer uma opção para o usuário. Normalmente é representado por uma caixa de seleção, que quando está com "check" representa "sim" e quando está com "não" representa "não".
`JComboBox`	Objeto usado para oferecer mais de uma opção para o usuário em forma de lista *drop-down*. Para cada lista somente um item pode ser selecionado.
`JList`	Objeto usado para oferecer mais de uma opção para o usuário, mas, diferentemente do JComboBox, esse componente permite a seleção de mais que uma opção.
`JPanel`	Objeto usado para organizar diversos componentes.

Fonte: adaptado de Deitel e Deitel (2016, p. 378).

Reflita

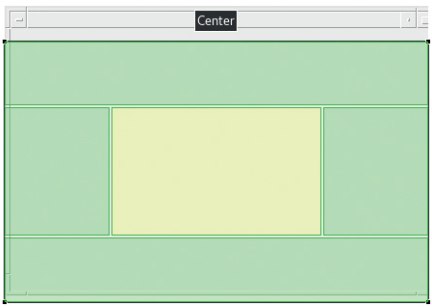
Para adicionar um *frame* foi preciso incluir a biblioteca `javax.swing.JFrame`, para adicionar um *label* foi necessário importar `javax.swing.JLabel`, e para adicionar uma caixa de texto foi necessário importar `javax.swing.JTextField`. Diante disso, para cada componente será necessário incluir uma biblioteca? E se houver dez, vinte ou mais componentes diferentes serão necessárias todas as linhas de inclusão?

Os componentes na interface podem ser posicionados de forma estática ou dinâmica. No segundo caso tem-se a alocação de componentes em posição relativa. Dessa forma, para cenários em que a tela muda de tamanho (devido a resolução do monitor ou interação com o usuário) as posições e tamanhos dos componentes são refeitas automaticamente.

Exemplificando

Para produzir interfaces que se adaptem a qualquer tamanho de janela ou resolução do sistema operacional é necessário utilizar posições relativas aos componentes da tela que está sendo produzida. O método `BorderLayout()`, pertencente à biblioteca AWT, fornece um mecanismo de layout responsivo, no qual uma tela pode ser dividida em cinco partes: *NORTH*, *SOUTH*, *EAST*, *WEST* e *CENTER* (HORSTMANN, 2016), como a Figura 1.3 apresenta.

Figura 1.3 | Divisão da classe `BorderLayout` do Java Swing



Fonte: captura de tela do software Java elaborada pelo autor.

O método pode ser utilizado configurando um novo layout e adicionando elementos em cada parte específica, por exemplo, as duas linhas a seguir criam um botão na parte inferior da tela:

```
setLayout(new BorderLayout());
add(new Button("Sim"), BorderLayout. SOUTH);
```

A melhor forma de aprender a programar é programando! Portanto, vamos agora implementar o código necessário para criar o layout da Figura 1.4.

Figura 1.4 | Interface gráfica feita pelo código do Quadro 1.4.

Fonte: captura de tela do software Java elaborada pelo autor.

Os componentes de texto fixo que indicam o conteúdo de cada campo e os elementos para entrada de texto estão descritos no Quadro 1.3. Todavia, o campo indicado com o texto "CPF" utiliza o componente `JFormattedTextField`, que é dedicado a receber entradas de texto onde é possível definir uma máscara que determina um formato de entrada, como exemplo `###.###.###-##`, que aceita uma sequência de números intercalados por um ponto, com um traço precedente aos dois últimos dígitos (FURGERI, 2015). Na frente do texto "Tipo de usuário" da Figura 1.3 está um componente que possui uma lista suspensa para a seleção de elementos pré-definidos (WINDER, 2009). E por fim, como último elemento da Figura 1.3, o componente mais abaixo com o texto "Enviar" pertence à classe `JButton`, que representa botões no Java Swing (MANZANO, 2014). No Quadro 1.4 está o código necessário para a implementação da interface gráfica da Figura 1.4.

As linhas 1 a 4 descrevem quais classes necessitam ser agregadas ao código para que seja possível construir a interface. A linha 5 define que a classe `PrimeiraTela` é uma especialização de `JFrame` (primeiro passo para a criação da interface gráfica). Os atributos da classe que estão entre as linhas 6 a 14 são os elementos que descrevem quais itens a tela possui (segundo passo para a criação), sendo eles os elementos que tornam a opção de utilizar herança ainda mais necessária, pois são novos elementos que serão incorporados a esse `JFrame`. O construtor dessa classe que

está entre as linhas 15 e 47 é o elemento central, pois ele faz todas as configurações necessárias para que a tela tenha a aparência desejada (passos 3 e 4 para a produção da interface gráfica). A linha 16 define o tamanho da tela com o método `setSize(int x, int y)` e a linha 17 define o título com o método `setTitle(String title)`. A linha 18 (`ctn = getContentPane();`) busca o atributo do `Container` do `JFrame` e a adiciona no atributo da classe `ctn` da `PrimeiraTela`. Isso é necessário para garantir a organização, pois o `Container` é a parte do `JFrame` que pode receber elementos gráficos, configurar esquemas de cores, layouts para inserção de componentes e outros. Portanto, todos os elementos serão adicionados no atributo `ctn`, que por sua vez será controlado e exibido pelo `JFrame`. Dentro do construtor, entre as linhas 19 e 29, inicia-se a quarta etapa para criar a interface, onde se deve instanciar, configurar e posicionar os itens na tela.

Na etapa 4 de criação de interfaces gráficas, o primeiro passo é instanciar os objetos, e junto a este processo configurar os componentes. O primeiro componente a ser instanciado é `JLabel`, o qual possui um construtor que pode receber o texto a ser apresentado na tela, como no exemplo: `lblNome = new JLabel("Nome")`. Esse processo é repetido nas linhas 21 e 26. Os outros componentes também são inicializados como `JTextField`, sendo que ele é um componente para receber entrada de texto, instanciado na linha 20 com um construtor sem parâmetros. Na linha 23 é inicializado o *JFormattedTextField* que é como o `JTextField`, porém com a possibilidade de informar uma máscara de entrada de texto, como: `new MaskFormatter` *("###.###.###-##")*. Na linha 27 se faz a inicialização do `JComboBox`, que recebe em seu construtor as opções que serão apresentadas no menu suspenso, e, por fim, na linha 28 se instancia o `JButton`, com um construtor que recebe o texto que deverá aparecer no botão.

Com todos os elementos já configurados, é necessário então posicioná-los na janela, e com isso é possível concluir a etapa 4 da criação de uma interface gráfica. A linha 30 faz com que não se utilize layouts, sendo possível definir a posição absoluta na janela. Entre a linha 30 a 43 são feitos o posicionamento e a definição de tamanho de cada componente com o método `setBounds()`, e são inseridos no `Container` que pela estrutura do `JFrame` é adicionado na tela. E, por fim, nas linhas 44 e 45 o método `setVisible` faz que a tela

seja visível, e o método `setDefaultCloseOperation` configura o `frame` para que quando fechada a janela, a aplicação finalize. O restante do código se refere à inicialização da classe.

Quadro 1.4 | Exemplo de utilização de componentes do Java Swing

```
1.  import java.awt.Container; //biblioteca para
    containers
2.  import java.text.ParseException;
3.  import javax.swing.*; //simplificando a
    inclusão de bibliotecas
4.  import javax.swing.text.MaskFormatter;

5.  public class PrimeiraTela extends JFrame {

6.  private JLabel lblNome;
7.  private JTextField txtNome;
8.  private JLabel lblCPF;
9.  private JFormattedTextField txtCPF;
10. private JLabel lblTipo;
11. private JComboBox cmbTipo;
12. private final String[] tiposUsuarios =
    {"Adminstrador","Geral"};
13. private JButton btnOK;
14. private Container ctn;

15.    public PrimeiraTela(){
16.           setSize(400,300);
17.           setTitle("Tela Inicial");
18.           ctn = getContentPane();
19.           lblNome = new JLabel("Nome");
20.           txtNome = new JTextField();
21.           lblCPF = new JLabel("CPF");
22.           try {
23.              txtCPF = new JFormattedTextField(new
                 MaskFormatter("###.###.###-##"));
```

```
            } catch (ParseException e) {
24.                 e.printStackTrace();
25.         }
26.         lblTipo = new JLabel("Tipo de usuário");
27          cmbTipo = new JComboBox(tiposUsuarios);
28.         btnOK = new JButton("Enviar");
29.         ctn.setLayout(null);
30.         lblNome.setBounds(0, 0, 100, 25);
31.         txtNome.setBounds(150,0,200,25);
32.         lblCPF.setBounds(0,50,100,25);
33.         txtCPF.setBounds(150,50,200,25);
34.         lblTipo.setBounds(0,100,200,25);
35.         cmbTipo.setBounds(150,100,200,25);
36.         btnOK.setBounds(150, 150, 100, 100);
37.         ctn.add(lblNome);
38.         ctn.add(txtNome);
39.         ctn.add(lblCPF);
40.         ctn.add(txtCPF);
41.         ctn.add(lblTipo);
42.         ctn.add(cmbTipo);
43.         ctn.add(btnOK);
44.         setVisible(true);
45.         setDefaultCloseOperation(JFrame.EXIT_
            ON_CLOSE);
46.     }
47.
48.     public static void main(String[] args) {
49.         PrimeiraTela t1 = new PrimeiraTela();
50.     }
51. }
```

Fonte: elaborado pelo autor.

Assimile

Segundo Deitel e Deitel (2016), para a máscara do objeto `JFormattedTextField` é possível utilizar os seguintes codificadores:

- **#** para representar a entrada de números.
- **U** para letras em caixa alta.
- **L** para letras em caixa baixa.
- **A** para qualquer número ou letra.
- **?** para qualquer caractere.
- ***** para qualquer elemento.
- **H** para entrada em hexadecimal.

Com esse exemplo de interface gráfica utilizando o Java Swing é possível adicionar outros elementos e criar a tela que você está precisando para concluir sua tarefa. Lembre-se de que uma interface gráfica apresenta muitas linhas de código, sendo assim é necessário manter a organização do código.

Pesquise mais

O Java possui uma vasta documentação e ferramentas para ajudar no desenvolvimento de interfaces gráficas, tais como:

- Documentação geral do Java. ORACLE Technology Network. **Java™ Platform, Standard Edition 7 API Specification.** Disponível em: <https://docs.oracle.com/javase/7/docs/api/overview-summary.html>. Acesso em: 16 mar. 2018.
- Vídeo sobre a criação de janela utilizando editor gráfico. MARTÍNEZ, Juan Jesús. **Java Swing componentes básicos.** Mar. 2015. Disponível em: <https://www.youtube.com/watch?v=yL_G0uCVSmI>. Acesso em: 16 mar. 2018.
- Tutorias da Oracle para Java Swing. ORACLE Technology Network. **The Java™ Tutorials.** Disponível em <https://docs.oracle.com/javase/tutorial/uiswing/>. Acesso em: 16 mar. 2018.

Sem medo de errar

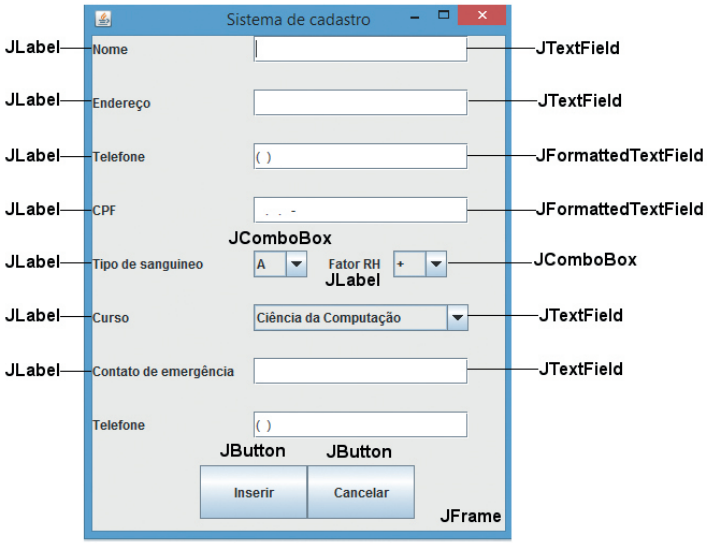
Sua tarefa na empresa é criar uma interface gráfica para um sistema de uma universidade. Esse é o primeiro passo para que um sistema legado seja atualizado e é sua primeira tarefa na empresa. Sendo assim, é muito importante que essa tarefa seja feita com precisão, para não afetar o projeto e não prejudicar sua evolução na carreira. Portanto, é necessário criar uma interface com os itens:

- Nome (*label* e campo de texto).
- Endereço (*label* e campo de texto).
- Código de área e telefone (*label* e campo de texto com máscara de formato).
- Tipo sanguíneo e fator Rh (*label* e caixa de seleção).
- Curso atual (*label* e caixa de seleção).
- Nome e telefone do contato de emergência (*labels*, caixa de texto e caixa de texto com máscara de formatação).
- Botões para inserir e cancelar.

Além da interface, um diagrama apontando quais componentes foram utilizados é de grande interesse para indicar aos próximos desenvolvedores quais são os padrões utilizados.

A Figura 1.5 apresenta o resultado e aponta quais componentes foram utilizados, essa informação será mais fácil para os próximos desenvolvedores terem um guia de como foi feito essa interface e quais componentes devem ser utilizados.

Figura 1.5 | Interface para o cadastro



Fonte: captura de tela do software Java elaborada pelo autor.

Agora, implemente o código necessário para produzir a tela da Figura 1.5. Acessando o link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/programacao-orientada-a-objetos-II/u1/s1/codigo.pdf> ou por meio do QR Code.

Avançando na prática

Sistema para cadastro de empresas em *Enterprise Resource Planning*

Descrição da situação-problema

Sistemas do tipo *Enterprise Resource Planning* (ERP) envolvem a construção e integração de vários subsistemas e são largamente utilizados por empresas dos mais diversos ramos. Você, como programador, ficou responsável por criar os campos do formulário que recebe dados de fornecedores.

Para isso, além de outros diversos campos, é necessário fazer um campo para cadastro de CEP, CNPJ e código de fornecedor. Os campos de CEP e CNPJ são numéricos, por exemplo:

- CEP: 12912-010
- CNPJ: 50.364.671/0001-00

Já o campo de código de fornecedor apresenta uma formação em que os dois primeiros elementos são letras de caixa alta, um separador representado por um traço, os próximos cinco elementos são números e o último elemento é uma letra em caixa baixa: AA-12345a. Para essa tarefa apenas mostre como seria a criação da instância desses três campos.

Resolução da situação-problema

Para cada campo é necessário utilizar uma instância da classe `JFormattedTextField` com um elemento de `MaskFormatter` diferente:

- CEP:`JFormattedTextFieldtxtCEP = new JFormatteddTextField(new MaskFormatter("######-##"));`
- CNPJ:`JFormattedTextFieldtxtCNPJ = new JFormattedTextField(new MaskFormatter("##.###.####/####-##"));`
- Código de fornecedor:`JFormattedTextFieldtxtCodFornecedor = new JFormattedTextField(new MaskFormatter("UU-#####L"));`

Na máscara do *JFormattedTextField* o **U** representa a entrada de letras que são tratadas para caixa alta, o **L**, a entrada de letras com caixa baixa, e o **#** representa a entrada de um número.

Faça valer a pena

1. A interface gráfica representa o primeiro contato do usuário com o sistema. Essa interface garante que o usuário utilize todos os aspectos que o sistema pode fornecer, e no caso do Java é possível empregar diversos componentes para fazer a construção das janelas de interface.

Dos componentes descritos a seguir, quais deles pertencem ao Java Swing?

a) `JButton, JFrame e JText`.

b) `JTextField, JSwing e JFrame`.

c) `JFrame, JButton e JTextField`.

d) `Container, JButton e JWindowsP`.

e) `JFrame, JButton, JTextField, JMask`.

2. Para a criação de uma interface utilizando o Java Swing podemos seguir cinco passos:

1. Especializar a classe que representa a janela.

2. Criar os componentes como atributos.

3. Utilizar ou não os layouts de posicionamento na tela.

4. Instanciar, configurar e posicionar os componentes na janela.

5. Tratar os eventos dos componentes para tratar as ações do usuário.

Caso o programador se esqueça de seguir a etapa de instanciar os componentes (etapa 4), quais serão os efeitos na interface?

a) Será gerado um erro `NullPointerException` durante a execução do código.

b) Durante a execução do código, o componente que não foi instanciado não aparecerá na tela.

c) Um erro de compilação será informado e o código não será executado.

d) O componente não poderá ser utilizado pelo usuário; os demais terão garantidas suas funcionalidades.

e) Será gerado um erro de `ArrayOutOfBounds`.

3. Existem diversos elementos visuais no Java Swing, e cada um deles apresenta funções bem definidas, que podem ser utilizados diversas vezes na mesma interface. Alguns elementos têm funções semelhantes e alguns aspectos peculiares de configuração e parâmetros.
Um campo de texto deve receber apenas letras, tanto em caixa baixa como em caixa alta, na quantidade máxima de 10 letras. Qual componente deve ser utilizado?

a) `JFormattedTextField`.

b) `JTextField`.

c) `JTextArea`.

d) `JComboBox`.

e) `JText`.

Seção 1.2

Programação em Java usando orientação a eventos

Diálogo aberto

Caro estudante, certamente você está habituado a usar computadores, smartphones e tablets, entre outros dispositivos eletrônicos. Para comprar um aplicativo na loja de aplicativos do seu smartphone, você tem que, no mínimo, clicar em um botão. Para realizar uma compra on-line, você precisa efetuar seu *login*. Para alterar a fonte do texto em um editor, você deve selecionar o texto e clicar na opção desejada. Todos os exemplos mencionados fazem parte dos tratamentos de eventos, assunto central da nossa seção.

Dando continuidade ao seu projeto de migração de um sistema legado de uma universidade, agora que uma das telas de cadastro foi criada, seu trabalho é implementar alguns tratamentos de eventos para essa interface gráfica. Com isso, é necessário tratar os seguintes eventos e executar as seguintes ações:

- Adicionar um campo de CPF, e quando o usuário terminar a inserção desse campo, o sistema deve verificar se o CPF está preenchido.
- Se o usuário preencher o campo "nome contato de emergência", é também obrigado a preencher o número de telefone do contato de emergência; isso é informando quando o campo de contato perde o foco do usuário.

Nessa seção o foco será o tratamento de eventos que uma interface gráfica pode gerar, e é necessário utilizar os mecanismos propostos pelo Java Swing para garantir que a orientação a objetos será correta. Você verá como abordar as formas de tratar esses eventos, quais eventos podem ser capturados e quais são as boas práticas nesse tipo de modelagem.

Vamos produzir um software com qualidade? Tenha foco e veja que é um processo importante e que apresenta alguns detalhes elementares de padronização para se elaborar classes com alta coesão.

Não pode faltar

Na seção anterior tratamos as formas de configurar uma interface gráfica e adicionar os componentes nessa janela. É importante ressaltar que as interfaces gráficas representam como o usuário percebe o sistema, e para se criar essa interface é necessário seguir alguns passos (HORSTMANN, 2016; DEITEL e DEITEL, 2016):

1. Criar uma relação de especialização com a classe que representa sua tela.
2. Declarar como atributos os elementos que serão adicionados à tela.
3. Definir a forma de alocação dos elementos gráficos na tela.
4. No construtor, instanciar, configurar e posicionar os itens na tela.
5. Tratar os eventos dos componentes para tratar as ações do usuário com a interface gráfica.

Interface gráfica x Interface

Seguindo esses passos é possível criar e posicionar os componentes, todavia, para que essa interface seja interativa, é necessário tratar os eventos decorrentes das ações do usuário na janela. No Java Swing existem diversos eventos que podem ser tratados através do uso de **interfaces** da orientação a objetos. Segundo Deitel e Deitel (2016) interfaces são coleções de métodos relacionados que informam aos objetos o que estes devem fazer, mas não como fazer. É importante perceber que esse novo elemento apresenta conceito e implementação distintos de uma interface gráfica. As interfaces são elementos que padronizam um conjunto de métodos; dessa forma é possível que uma classe apresente qualquer nível de complexidade, e essa interface propicia as formas de fazer o acesso padronizado e encapsulado.

Para criar uma interface, clique com o botão direito no projeto, selecione `New >> Interface`, escolha a pasta e o nome. O código no Quadro 1.5 apresenta um exemplo desse item da orientação a objetos. Repare que nesse tipo de código não temos nenhuma implementação de código, e sim um conjunto de métodos; essas descrições de implementação representam uma forma de

padronizar o método que será o responsável por tratar um evento na interface gráfica (FURGERI, 2015). Na linha 1 se define qual será o nome da interface, e nas linhas 2 e 3 são descritos quais métodos as classes que utilizarem essa interface deverão ter. Portanto, quando uma classe implementar a interface `AcessoElementos`, esta deverá, obrigatoriamente, implementar os métodos `getElemento` e `setElemento`.

Quadro 1.5 | Exemplo de interface da orientação a objetos do Java

```
1.  public interface AcessoElementos {
2.          public int getElemento (int index);
3.          public void setElemento (int index);
4.  }
```

Fonte: elaborado pelo autor.

Para uma classe implementar uma interface, esta deve usar o comando `implements`, conforme ilustra o Quadro 1.6. Veja que na linha 2, a classe `Aluno` implementa a interface `AcessoElementos`, portanto os métodos da interface tiveram que ser declarados. A notação `@Override` é usada para indicar que os métodos serão sobrescritos.

Quadro 1.6 | Uso de uma interface

```
1.  import javax.swing.*;
2.  public class Aluno extends JFrame implements
    AcessoElementos{
3.          @Override
4.          public int getElemento(int index) {
5.                  return 0;
6.          }

7.          @Override
8.          public void setElemento(int index) {
9.                  //códigos
10.         }
11. }
```

Fonte: elaborado pelo autor.

Assimile

No caso do Java temos a interface gráfica que é construída pelo Swing, no qual existem diversas formas de janelas, botões e outros elementos. Também existem as interfaces (classe interface), que são elementos da orientação a objetos feitos para propiciar a padronização de formas de acesso e também definir quais métodos podem ser utilizados para ações específicas de uma classe.

Para evitar confusões na definição dos termos, chamaremos de **interface** os elementos da orientação a objetos que descrevem um conjunto de métodos, e de **interface gráfica** os elementos como botões, janelas, campos e outros. Ao se construir uma interface gráfica usando Java Swing, acrescentamos as interfaces para definir qual método será utilizado no momento de chamada que um evento é iniciado.

Tratamento de eventos

A criação da interface gráfica representa o mecanismo de interação do usuário com o sistema. Cada ação do usuário gerará uma reação no sistema e do sistema. Esse conjunto de ações (interações) é o que se chama de **eventos**, e existem diversos tipos já predefinidos, por exemplo, clicar em um botão, pressionar uma tecla, maximizar uma janela, etc. O Quadro 1.7 traz alguns eventos suportados pela linguagem Java. É importante ter em mente que para cada evento é necessário implementar um método; ao longo do livro veremos diversos métodos.

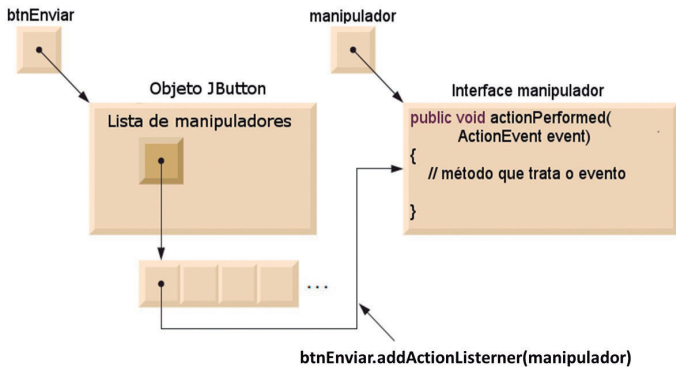
Quadro 1.7 | Eventos na linguagem Java

Evento	Descrição
`ActionListener`	Evento gerado pelo clique com o mouse em um botão.
`ItemListener`	Evento gerado quando um item em uma lista é selecionado.
`FocusListener`	Evento gerado quando um elemento ganha ou perde foco.
`WindowListener`	Evento gerado quando ocorre uma mudança na janela.

Fonte: elaborado pelo autor.

Para tratar a ação de "clique em um botão" em uma janela gráfica feita em Java Swing, utiliza-se a interface `ActionListener`. Na Figura 1.5 temos a representação de um objeto da classe `JButton`, instância que pode tratar diversos eventos. A instância "`btnEnviar`" registra uma interface do manipulador, nesse caso o `ActionListener`, para tratar um de seus possíveis eventos.

Figura 1.5 | Exemplo de como tratar o evento de clique em `JButton` do Java Swing



Fonte: adaptada de Deitel e Deitel (2016, p. 494).

Novamente, repare que nesses métodos não há implementação, apenas a descrição (assinatura). Para implementar o tratamento de eventos é necessário que a classe que está especializando o `JFrame` tenha alguma implementação dos métodos da *interface* que trate os eventos e associe essa interface aos componentes que enviarão os eventos (HORSTMANN, 2016).

Para demonstrar, utilizaremos a tela da Figura 1.6. Essa interface gráfica dispõe de um `JButton`, um `JLabel` e um `JTextField`. O objetivo é verificar se o botão foi clicado pelo usuário, e para isso é possível utilizar a interface `ActionListener` de três maneiras distintas (MANZANO, 2014):

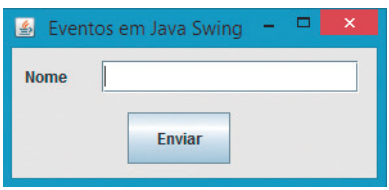
1. Implementar a interface `ActionListener` diretamente na classe que especializa o `JFrame`.
2. Criar uma classe que implementa a interface e trata todos os eventos dos componentes gráficos.
3. Fazer que cada componente crie uma classe anônima que implementa a interface `ActionListener`.

Os itens 1 e 2 utilizam apenas um método para tratar os eventos de diversos componentes; dessa forma torna-se necessário utilizar os parâmetros das interfaces que tratam os eventos e detectar qual o objeto fez a chamada. O item 3 cria uma interface específica para cada componente, sem a necessidade de avaliar qual componente fez a chamada.

Reflita

É possível tratar os eventos de um componente gráfico do Java Swing de três formas: (i) implementar a interface manipuladora na classe que cria a interface gráfica; (ii) criar uma classe separada para tratar os eventos; (iii) utilizar uma classe anônima que implementa a interface para tratar o evento. Todavia, em um sistema que trata 50 eventos, qual seria a opção mais indicada? Quais seriam os efeitos na classe que monta a interface gráfica, caso opte pelo terceiro modo?

Figura 1.6 | Exemplo de interface gráfica



Fonte: captura de tela do software Java elaborada pelo autor.

Vamos começar implementando o código necessário para gerar a interface gráfica. O Quadro 1.8 apresenta a parte que cria a interface da Figura 1.6. As linhas 1 e 2 descrevem as classes que serão importadas; a definição da classe que especializa a classe `JFrame` está na linha 3, e entre as linhas 4 a 7 são criados os atributos que representam os componentes da tela, nesse caso o `JLabel`, `JButton` e `JTextField`. As linhas 10 e 11 representam a configuração da tela em termos de tamanho (`setSize`) e título (`setTitle`), os elementos das linhas 12 e 13 são os pontos que tratam da posicionamento em local absoluto na tela `ctn.setLayout(null);` e `ctn = getContentPane();`. Nas linhas 14 a 22 os objetos são alocados e configurados. Veja que na linha 23 deixamos indicado onde será inserido parte do tratamento de evento ao clicar no botão. Na linha 24, a visibilidade da tela, e nas

linhas 26 a 28, a função inicial do programa. Na linha 29 também deixamos indicado que adicionaremos um código para tratamento do evento.

Quadro 1.8 | Porção de código que cria a interface da Figura 1.6

```
1.  import java.awt.Container;
2.  import javax.swing.*;

3.  public class PrimeiraTela extends JFrame{

4.          private JButton btnok;
5.          private JTextField txtNome;
6.          private JLabel lblNome;
7.          private Container ctn;
8.          public PrimeiraTela()
9.          {
10.                 setSize(300,140);
11.                 setTitle("Eventos em Java Swing");

12.                 ctn = getContentPane();
13.                 ctn.setLayout(null);

14.                 btnok = new JButton("Enviar");
15.                 lblNome = new JLabel("Nome");
16.                 txtNome = new JTextField();

17.                 lblNome.setBounds(10,10,100,25);
18.                 txtNome.setBounds(70,10,200,25);
19.                 btnok.setBounds(90,50,80,40);

20.                 ctn.add(lblNome);
21.                 ctn.add(txtNome);
22.                 ctn.add(btnok);

23.                 // nesse ponto vamos inserir
                    tratamento dos eventos
24.                 setVisible(true);
25.     }
```

```
26.     public static void main(String[] args) {
27.             PrimeiraTela tela = new PrimeiraTela();
28.     }
29.             // nesse ponto vamos inserir tratamento
                dos eventos
30. }
```

Fonte: elaborado pelo autor.

Com a interface gráfica criada, vamos tratar o evento de clique no botão "Enviar" utilizando a primeira opção, ou seja, implementando a interface `ActionListener` diretamente na classe que especializa o `JFrame`. Para isso é necessário alterar a declaração da classe e implementar uma interface, nesse caso utilizando uma `ActionListener` conforme o Quadro 1.9 (esse código deverá ser acrescentado ao Quadro 1.8, nas linhas indicadas). Nas linhas 1 e 2 foram importadas as referências para o tratamento de eventos que faremos. Na linha 3 é possível verificar a utilização do comando `implements` indicando a `ActionListener`, e com isso essa classe é forçada a implementar o método `actionPerformed()` feito na linha 30. Na linha 23, adicionamos o método `addActionListener()` no botão, e o parâmetro `this` define que a classe que tratará os eventos é a própria classe, nesse caso a `PrimeiraTela`. Essa abordagem é interessante quando se tem poucos eventos a serem tratados, pois se a janela apresentar mais botões e ambos definirem que a classe que desenha a interface gráfica cuidará de seus eventos, será necessário comparar a resposta do método `getActionCommand()` com o texto do botão em questão, como o `if` na linha 32.

Quadro 1.9 | Código que cria um tratamento de evento para a Figura 1.6

```
1.  import java.awt.event.ActionEvent;
2.  import java.awt.event.ActionListener;

3.  public class PrimeiraTela Tela extends
    JFrame implements ActionListener{
...
23.         btnok.addActionListener(this);
...
29.         @Override
30.         public void actionPerformed(ActionEvent e)
31.         {
```

```
        // caso seja necessário tratar eventos de
        mais de um botão
32.     if(e.getActionCommand().equals("Enviar"))
33.     {
34.             txtNome.setText("Botão clicado");
35.     }
36.   }
37. }
```

Fonte: elaborado pelo autor.

A segunda forma de tratar esses eventos consiste em criar outra classe que implemente o `ActionListener`, porém existe um problema na forma de tratar e diferenciar eventos que ainda não foi resolvido, sendo necessário criar diversas instâncias dessa classe de tratamento, e ainda fazer a comparação com o retorno do método `getActionCommand` da classe `ActionEvent`, o que torna essa opção de pouco utilidade.

Para ter uma solução mais precisa é interessante utilizar o conceito de classe anônima (terceira opção), que como sua própria classificação indica, não tem nome e cada elemento gráfico terá uma classe separada para fazer o seu próprio tratamento (DEITEL e DEITEL, 2016).

Veja no Quadro 1.10 a implementação de uma classe anônima dentro da classe `Tela`. Na linha 3 repare que não existe mais a implementação da interface, mesmo assim é necessário incluir as referências (linhas 1 e 2). Na linha 5 o `ActionListener` é adicionado ao botão pelo método `addActionListener()` usando uma classe anônima para fazer o tratamento do evento. Como a classe anônima e a classe `Tela` estão no mesmo arquivo, a classe anônima pode acessar os elementos da classe que a engloba (WINDER, 2009). Na linha 7 é feita a chamada de um método da classe que fará o tratamento da ação no `JButton`. O método `trataBotaoOk()` foi criado na linha 9 e na linha 10 foi especificada a ação do método, que consiste em acessar o método `setText()` do objeto `txtNome` adicionando um texto.

Quadro 1.10 | Tratamento de eventos utilizando classe anônima

```
1.  import java.awt.event.ActionEvent;
2.  import java.awt.event.ActionListener;

3.  public class Tela extends JFrame {
4.     public Tela()
       {
5.     btnok.addActionListener(new ActionListener() {
6.           public void actionPerformed(ActionEvent e) {
7.                trataBotaoOk();
8.           }
        });
       }
9.     public void trataBotaoOk()
       {
10.        txtNome.setText("Botão clicado");
       }
   }
```

Fonte: elaborado pelo autor.

O modelo de tratamento que o Quadro 1.10 apresenta é interessante, pois para cada elemento gráfico da tela é possível criar um método específico, garantindo assim a coesão de cada elemento da classe.

Em se tratando de eventos relacionados a uma caixa de texto é possível verificar se ocorreram alterações nas propriedades, remoções ou inserções. Essas alterações também podem ser vinculadas à interface `ActionListener`, bem como a outros eventos como, por exemplo, ao `FocusListener` e ao `DocumentListener`, entre outros. Observe o código no Quadro 1.11: na linha 3 o objeto `txtNome` é vinculado à interface `DocumentListener` em uma classe anônima, pelo método `addDocumentListener()`. Essa interface especifica a implementação de três métodos: `removeUpdate()`, `insertUpdate()` *e* `changedUpdate()`, e mesmo que você não vá implementar os três a chamada é obrigatória. O `removeUpdate()`, na linha 4, executa ações quando algo é apagado de uma caixa de texto. O método `insertUpdate()`,

na linha 5, executa ações quando algo é inserido na caixa de texto e o método `changedUpdate()`, quando algo é alterado. Os três métodos implementam a interface `DocumentEvent`, portanto também é necessário importar a referência `javax.swing.event.DocumentEvent`.

Quadro 1.11 | Tratar evento de mudança de texto em um *JTextField*

```
1.  import javax.swing.event.DocumentEvent;
2.  import javax.swing.event.DocumentListener;

3.  txtNome.getDocument().addDocumentListener(new
    DocumentListener() {
4.     public void removeUpdate(DocumentEvent e) {
           // ações quando texto for apagado
       }
5.     public void insertUpdate(DocumentEvent e) {
           // ações quando texto for inserido
       }
6.     public void changedUpdate(DocumentEvent e) {
           // ações quando texto for alterado
       }
    });
```

Fonte: elaborado pelo autor.

Para componentes gráficos que oferecem opções de escolha para o usuário, como o `JComboBox` por exemplo, pode ser interessante efetuar tratamento nos dados assim que o usuário seleciona uma das opções. O tratamento de eventos nesse objeto pode ser feito implementando a interface `ItemListener` (disponível em `java.awt.event.ItemListener`) e o método `addItemListener()`, conforme exemplifica o código do Quadro 1.12. Na linha 3 é feita a criação da classe anônima para a implementação da interface `ItemListener`. O uso dessa interface obriga a implementação do método `itemStateChanged()`, que apresenta como parâmetro a interface `ItemEvent` (disponível em `java.awt.event.ItemEvent`), conforme linha 4. Na linha 5, verificamos se o evento disparado (*e*) é igual a "selecionar um item"; se for, o método

`trataJmbTipos()` é invocado na linha 7. Na linha 11 é declarado o método para tratar o evento, na linha 13 é mostrada uma mensagem gráfica exibindo qual item foi selecionado pelo `JComboBox`.

Quadro 1.12 | Tratamento de seleção de um *JComboBox*

```
1.  import java.awt.event.ItemEvent;
2.  import java.awt.event.ItemListener;

3.  jmbTipos.addItemListener(new ItemListener() {

4.     public void itemStateChanged(ItemEvent e) {
5.             if (e.getStateChange() == ItemEvent.
            SELECTED)
6.             {
7.                 trataJmbTipos();
8.             }
9.     }
10. });

11. public void trataJmbTipos()
12. {
13.         JOptionPane.showMessageDialog(this,
                    "Item selecionado: "+ jmbTipos.
                    getSelectedItem());
14. }
```

Fonte: elaborado pelo autor.

Exemplificando

Existem muitos outros eventos para serem apresentados, como detectar se um objeto foi selecionado, se um componente perdeu o foco, etc. A forma de tratar esses eventos segue o mesmo conceito visto nos outros casos. Por exemplo, para verificar se um componente gráfico recebeu ou perdeu o foco, o Quadro 1.13 apresenta um código que trata esses casos. Na linha 1 cria-se uma classe anônima para programar a interface `FocusListener`; essa criação força a criação dos métodos `focusLost()` e `focusGained()` que, respectivamente, definem se o foco foi perdido ou ganho.

Quadro 1.13 | Evento de perda ou ganho de foco sob um componente

```
1.  txtNome.addFocusListener(new FocusListener() {
2.        public void focusLost(FocusEvent e) {

          }
3.        public void focusGained(FocusEvent e) {
          }
    });
```

Fonte: elaborado pelo autor.

O tratamento de eventos em Java Swing deve sempre seguir os elementos da orientação a objetos, e os métodos devem ter alta coesão, sendo que cada uma dessas ações deve se limitar a funções específicas. Nesse sentido, a implementação das interfaces para tratar os eventos deve apenas recebê-los e chamar outros métodos para que cada ação seja tratada de forma independente.

Exemplificando

A Oracle se dedica a gerar documentação, exemplos e tutoriais para que a linguagem Java ganhe cada vez mais abrangência, sendo assim é interessante buscar as formas que a desenvolvedora da linguagem indica.

- Documentação do Java Swing. ORACLE TECHNOLOGY NETWORK. **Package javax.swing.** 2018. Disponível em: <https://docs.oracle.

com/javase/7/docs/api/javax/swing/package-summary.html>. Acesso em: 16 mar. 2018.

- Vídeo sobre o tratamento de eventos em Java Swing. JAVA&CIA. **Universidade XTI – JAVA – 077 – GUI, Eventos e Listeners.** Ago. 2014. Disponível em: <https://www.youtube.com/watch?v=VaYsYI-kZbMc>. Acesso em: 30 mar. 2018.
- Tutoriais da Oracle para *JButton*. ORACLE TECHNOLOGY NETWORK. **The Java™ Tutorials.** 2017. Disponível em: <https://docs.oracle.com/javase/tutorial/uiswing/components/button.html>. Acesso em: 30 mar. 2018.
- Tutorias da Oracle para *JComboBox*. ORACLE TECHNOLOGY NETWORK. **The Java™ Tutorials.** 2017. Disponível em: <https://docs.oracle.com/javase/tutorial/uiswing/components/combobox.html>. Acesso em: 30 mar. 2018.

Sem medo de errar

Você está trabalhando em uma empresa especializada em atualização de sistemas legados. Nessa etapa, seu trabalho consiste em tratar os eventos que os usuários podem gerar da interface gráfica do programa que substituirá uma parte do sistema legado de uma universidade. Os sistemas legados são elementos comuns que precisam de cuidados extras quando se trata de migração. Nesses sistemas, muitos dos usuários têm grande experiência em sua utilização, e a alteração de uma interface gráfica pode gerar problemas de adaptação. Para executar essa tarefa você deve alterar na tela de cadastro os seguintes itens:

1. Quando o usuário entrar com os dados de CPF ou contatos de emergência, o sistema deve tratar esses eventos. No caso do CPF, o campo não pode estar vazio, e o sistema deve garantir que exista um telefone de emergência se existir um nome de emergência.
2. O botão de inserir deve buscar os dados dos campos e guardar em variáveis, e o botão cancelar deve limpar todos os campos.

Para executar essa tarefa é necessário:

1. Criar uma relação de especialização com a classe que representa sua tela, no caso com a classe `JFrame`:

```
public class InterfaceGrafica extends JFrame
```

2. Declarar como atributos os elementos que serão adicionados à tela, no caso os campos:

- Nome (*label* e campo de texto):
- `JLabel lblNome;`

  `JTextField txtNome;`
- Endereço (*label* e campo de texto):

  `JLabel lblEndereco;`

  `JTextField txtEndereco;`
- CPF:

  `JLabel lblCPF;`

  `JFormattedTextField txtCPF;`
- Código de área e telefone (*label* e campo de texto com máscara de formato):

  `JLabel lblTelefone;`

  `JFormattedTextField txtTelefone;`
- Tipo sanguíneo e fator Rh (*label* e caixa de seleção):

  `JLabel lblTipoSanguineo;`

  `JComboBox cmbTipoSanguineo;`

  `JLabel lblRH;`

  `JLabel lblRH;`
- Curso atual (*label* e caixa de seleção):

  `JLabel lblCurso;`

  `JComboBox cmbCurso;`
- Nome e telefone do contato de emergência (*labels*, caixa de texto e caixa de texto com máscara de formatação):

  `JLabel lblTelefoneEmergencia;`

  `JFormattedTextField txtTelefoneEmergencia;`
- Botões para inserir e cancelar:

  `JButton btnInserir;`

  `JButton btnCancelar;`

3. Definir a forma de alocação dos elementos gráficos na tela:

```
Container ctn = getContentPane();
ctn.setLayout(null);
```

4. No construtor, instanciar, configurar e posicionar os itens na tela, por exemplo, para o Nome: `lblNome = new JLabel("Nome");`

```
txtNome = new JTextField();
lblNome.setBounds(0, 0, 100, 25);
txtNome.setBounds(150,0,200,25);
ctn.add(lblNome);
ctn.add(txtNome);
setVisible(true);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

5. Tratar os eventos dos componentes para tratar as ações do usuário com a interface gráfica:

1. Para os cliques dos botões:

```
btnInserir.addActionListener(newActionListener()
   {
    public void actionPerformed(ActionEvent e) {
          trataEventosBotao();
    }
});
```

2. Para os campos de emergência:

```
txtNomeEmergencia.addFocusListener(new
FocusListener() {
    public void focusLost(FocusEvent e) {
          trataCampoContatoEmergencia();
    }
    public void focusGained(FocusEvent e) {
    }
});
```

Avançando na prática

Sistema de votação

Descrição da situação-problema

Uma empresa precisa fazer uma eleição para os representantes da CIPA, e você, como membro da empresa que presta o serviço de TI, foi alocado para essa tarefa. Para isso, é necessário produzir uma interface gráfica utilizando Java Swing que contenha:

- Conjunto de `RadioButtons` para a seleção de dois representantes.
- Campo de botão para confirmar o voto.
- Quando o voto for confirmado o sistema deve informar a operação.

Para a configuração da votação será feita uma lista em papel e o sistema poderá ser utilizado sem essa integração.

Resolução da situação-problema

O Quadro 1.14 apresenta o código para a produção da interface e tratamento dos eventos. Os comentários no código explicam os campos que não foram apresentados no texto.

Quadro 1.14 | Código para sistema de votação

```
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.*;

public class SistemaVotacao extends JFrame{

        private JButton btnVotar;
        private JRadioButton  rbCd1;
        private JRadioButton  rbCd2;
        private ButtonGroup btnGrupo;
        private Container ctn;
```

```
public SistemaVotacao()
{
        setSize(300,210);
        setTitle("Escolha um candidato!");

        rbCd1 = new JRadioButton("Candidato 1");
        rbCd2 = new JRadioButton("Candidato 2");

        //define o texto que o botão vai enviar quando
        selecionado
        rbCd1.setActionCommand("Candidato1");
        rbCd2.setActionCommand("Candidato2");

        btnVotar = new JButton("VOTAR");

        // cria um grupo de botões para garantir
        apenas a seleção de um deles e facilitar
        a busca pelo item selecionado
        btnGrupo = new ButtonGroup();

        btnGrupo.add(rbCd1);
        btnGrupo.add(rbCd2);

        ctn = getContentPane();
        ctn.setLayout(null);

        // marca esse campo como selecionado
        rbCd1.setSelected(true);

        rbCd1.setBounds(10,10,100,100);
        rbCd2.setBounds(150,10,100,100);
        btnVotar.setBounds(90,120,100,50);

        ctn.add(rbCd1);
        ctn.add(rbCd2);
        ctn.add(btnVotar);

        setVisible(true);
        setDefaultCloseOperation(JFrame.
        EXIT_ON_CLOSE);
        btnVotar.addActionListener(new
```

```
                ActionListener() {
                    public void
                    actionPerformed(ActionEvent e) {
                            trataVotar();
                    }
        });

    }

    public void trataVotar()
    {
            // Recupera o texto do botão que foi
            selecionado
               String candidatoSelecionado =
btnGrupo.getSelection().getActionCommand();
      // Imprime o texto do botão que foi selecionado
      JOptionPane.showMessageDialog(this,
               "Você votou em: "+
               candidatoSelecionado);
    }
    public static void main(String[] args) {
            SistemaVotacao stn=new SistemaVotacao();
    }
}
```

Fonte: elaborado pelo autor.

Faça valer a pena

1. Para tratar os eventos de componentes no Java Swing é necessário utilizar uma interface que força a criação de métodos no código. Esses métodos são chamados quando o evento acontece na interface gráfica. Assim, existem diversas formas de implementar essa interface e tratar esses eventos, como por exemplo, implementar a interface que trata o evento escolhido na classe que estende o `JFrame`.

No caso de implementar a interface `ActionListener` na classe que estende o `JFrame` e essa interface gráfica dispor de 30 botões, qual dos itens a seguir representa o tratamento necessário no método `ActionPerformed`?

a) É necessário utilizar o parâmetro `ActionEvent` do método `ActionPerformed` para diferenciar cada chamado de cada botão.

b) A interface `ActionPeformed` não deve ser utilizada para tratar cliques de botão.

c) O parâmetro `ActionEvent` não existe na interface `ActionPerformed`.

d) Não é possível detectar as diferentes chamadas, sendo assim a tela suporta apenas um botão.

e) O método do parâmetro `ActionEvent` a ser utilizado é o `getText`.

2. Das diversas formas de tratar um evento em interfaces gráficas utilizando Java Swing, é interessante destacar a técnica em que se adiciona uma interface diretamente para cada componente. Dessa forma, o tratamento dos eventos não necessita da detecção de qual componente fez o chamado, tratando assim a coesão dos métodos.
Para o tratamento em que cada elemento gráfico pode utilizar uma interface específica, qual tipo de classe é necessária?
a) Classe abstrata.
b) Classe anônima.
c) Classe interface.
d) Classe primitiva.
e) Classe estática.

3.Existem diversos eventos que podem ser tratados no Java Swing. Ganho ou perda de foco, seleção, clique e outros são exemplos do que se pode detectar durante a execução de um código. Para cada um desses eventos é necessário que cada um dos componentes tenha uma interface contendo os métodos, a qual avisará que evento ocorreu.
Para tratar o ganho de foco em um componente, qual interface e qual método devem ser utilizados?
a) `FocusListener` e `focusLost`.
b) `FocusChange` e `FocusChanged`.
c) `ActionListener` e `ActionEvent`.
d) `Focus` e `focusOn`.
e) `FocusListener` e `focusGained`.

Seção 1.3

Programação em Java usando banco de dados relacional

Diálogo aberto

Prezado aluno, com esta seção chegamos ao final da primeira unidade, e nessa caminhada já estudamos como criar e posicionar os componentes gráficos de uma interface gráfica. Vimos também como tratar os eventos de uma tela utilizando Java *Swing*, processo que envolve a utilização de interfaces implementadas em classes específicas para que seus métodos sejam chamados a cada ação do usuário com o sistema. Agora já temos ferramentas para produzir uma interface gráfica funcional, e nesse sentido são necessárias formas de utilizar os dados que a interface gráfica colhe dos usuários. Você já pensou no volume de informação que é gerado todos os dias? Onde e como esses dados são armazenados? Essas informações são armazenadas em sistema de banco de dados de diversas fabricantes e de diversas formas. A maneira mais direta de utilizar esses dados é empregar ferramentas de persistências de dados, tais como arquivos de texto *eXtensible Markup Language* (XML), ou sistemas de gerenciamento de banco de dados (SGBD).

Você está trabalhando em uma empresa que tem como projeto a atualização de um sistema legado universitário. A interface gráfica para a atualização do sistema legado da universidade foi aprovada, e a data para entrega da primeira versão totalmente funcional foi marcada. Agora é necessário implementar a parte de persistência de dados na interface gráfica que já foi produzida. Você, como membro da equipe desse projeto, foi destacado para fazer essa implementação. Para realizar essa tarefa é preciso:

- Recuperar os dados da interface gráfica utilizando o tratamento de eventos em variáveis locais.
- Criar uma classe que faça a conexão com o sistema de banco de dados.
- Produzir uma classe que execute os comandos no banco de

dados com a conexão criado no item anterior.

- Na classe que executa os comandos, criar métodos para receber os dados que serão inseridos e execute os comandos no sistema de banco de dados.

Aqui você deve lembrar sempre que cada classe deve fazer ações específicas para garantir a divisão de preocupações e aumentar a capacidade de manutenção dos códigos. Com isso, tornam-se necessárias formas de acesso a esses dados de maneira padronizada pela linguagem de programação.

Para essa tarefa utilizaremos o *Java Database Connectivity* (JDBC), acessando de forma padronizada e coerente à orientação a objetos. Nessa seção, trataremos das maneiras de se conectar a um sistema de banco de dados e fazer as operações de busca, inserção, remoção e alteração das informações. Com isso, vamos produzir códigos de alta coesão e qualidade.

Não pode faltar

Para que seja possível o desenvolvimento de aplicações que se conectem a um sistema de gerenciamento de banco de dados (SGBD), é preciso conectar-se diretamente ao banco utilizando uma *Application Programming Interface* (API) que o fabricante do SGDB provê. Todavia, a operação torna-se mais dinâmica quando a linguagem de programação fornece alguma maneira de conexão universal, sem a necessidade de mudar de API para troca de cada SGDB.

Introdução ao uso do MySQL em programas Java

Os SGBD são os elementos mais comuns para persistência de dados utilizados em aplicações comerciais, pois propiciam formas padronizadas para inserção, alteração, remoção e busca de dados. Portanto, é necessário verificar como as interfaces gráficas, quando acionadas pelo usuário, fazem o uso dos SGDBs para gravar seus dados.

Como existem diversos SGDBs, seria necessário utilizar bibliotecas específicas para cada sistema, o que causaria uma dependência ao tipo de persistência. Para utilizar os SGDBs em Java, especialmente em interfaces gráficas em Java Swing, é indicado utilizar o *Java Database Connectivity* (JDBC). O JDBC consiste em um conjunto de classes que são incorporadas ao *Java Development Kit* (JDK),

para possibilitar o acesso a diversos SGDBs de forma padronizada sem a necessidade de se utilizar formas específicas para cada sistema de banco de dados (FURGERI, 2015). O JDBC é compatível com diversos sistemas de banco de dados, tais como:

- MySQL: <https://www.mysql.com/>. Acesso em: 31 out. 2018.
- Oracle: <https://www.oracle.com/database/index.html>. Acesso em: 31 out. 2018.
- Microsoft SQL Server: <https://www.microsoft.com/en-us/sql-server/sql-server-2016>. Acesso em: 31 out. 2018.
- PostgreSQL: <https://www.postgresql.org/>. Acesso em: 31 out. 2018.

Dentre estas opções podemos destacar o MySQL, que consiste em uma solução *open source*, dispondo de versão comercial com suporte técnico e mais funcionalidades. Conta com elementos para controle de transação, *triggers,* suporte à conexão criptografada e muitos outros elementos de um sistema de banco de dados moderno. Além de suas características, apresenta diversas ferramentas que auxiliam no gerenciamento e desenvolvimento tais como o *MySQL Workbench*. Com essa ferramenta é possível criar, alterar e remover banco de dados, tabelas, *procedures* e outros elementos. Para obter o MySQL, no caso do Microsoft Windows, é necessário acessar https://www.mysql.com/downloads/ (ORACLE CORPORATION, 2018b), escolher a opção *MySQL Community Edition* e fazer o download de *MySQL Community Server* na opção *MySQL Installer for Windows*. O processo de instalação das últimas versões consiste em utilizar o assistente para tornar disponível o MySQL, sendo este um processo bem intuitivo.

Assimile

O termo "banco de dados" está relacionado ao arquivo ou arquivos que armazenam os dados. Em um sistema de gerenciamento de banco de dados, são os softwares que fazem a interface entre os sistemas que precisam guardar, alterar ou remover dados desses arquivos. Com isso, é possível garantir a integridade de dados quando ocorrem acessos concorrentes e outros eventos.

Para utilizar o JDBC com qualquer SGBD é necessário executar cinco passos, e então será possível executar comandos para buscar

dados ou enviar informações para o SGDB (DEITEL e DEITEL, 2016):

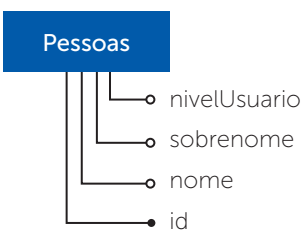
1. Estabelecer a conexão.
2. Criar um objeto da classe *statement* vinda da conexão para possibilitar a execução das consultas.
3. Executar as consultas.
4. Processar os resultados, sendo os dados enviados ou recebidos.
5. Fechar a conexão.

Pesquise mais

Antes de iniciar qualquer ação para utilizar o SGBD, é necessário criar um banco de dados; no caso do MySQL é possível criá-lo utilizando a linha de comando ou empregar alguma ferramenta como o MySQL *WorkBench*. Saiba mais no seguinte vídeo: CURSO EM VÍDEO. **Curso MySQL #3** - Criando o primeiro Banco de Dados. [S. l.], 29 fev. 2016. Disponível em: <https://www.youtube.com/watch?v=m9YPlX0fcJk>. Acesso em: 8 maio 2018.

A Figura 1.7 apresenta um exemplo de campos a serem armazenados para o cadastro de uma pessoa. Nosso objetivo é implementar esse diagrama na linguagem Java, que pertence ao paradigma da programação orientada a objetos, e para isso precisamos entender como funciona o JDBC.

Figura 1.7| Dados para cadastro de uma pessoa

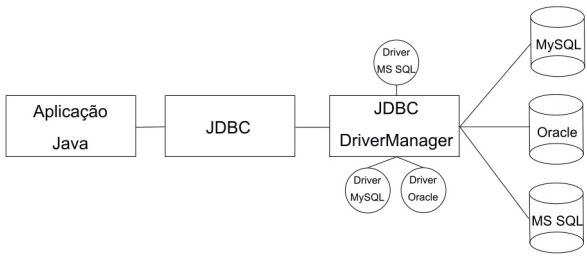


Fonte: elaborada pelo autor.

A Figura 1.8 apresenta uma visão de como o JDBC atua nas aplicações Java utilizando a API do JDBC, que garante uma interface única para acesso ao SGBD. O JDBC utiliza as classes que

controlam o *driver* que será utilizado para se conectar no banco de dados indicado. Para executar o primeiro passo, que consiste em estabelecer a conexão, é necessário garantir que o JDBC conheça o SGBD (HORSTMANN, 2016).

Figura 1.8 | Representação da ação do JDBC



Fonte: elaborada pelo autor

Lembrando: o JDBC é uma interface genérica para diversos sistemas de bancos de dados, e para que o JDBC possa fazer a conexão é necessário que ele conheça quais as peculiaridades do SGBC em que a conexão será estabelecida. Para isso, é preciso que o JDBC utilize um *driver* para o banco de dados que se deseja conectar, sendo preciso informar diretamente qual *driver* será utilizado (MANZANO,2014). A maneira de instalar depende do sistema operacional e do *driver*; nesse livro aprenderemos a usar o *driver* do MySQL.

A API JDBC é implementada pelos pacotes *java.sql* e *javax.sql*. Dentro desses pacotes estão as classes disponíveis para manipulação dos SGBD. Podemos destacar as classes:

- `java.sql.DriverManager`: essa classe é utilizada para criar a conexão com SGBD.
- `java.sql.Conection`: essa classe é utilizada para representar a conexão com o SGBD e fornecer acesso às consultas.
- `java.sql.Statement`: essa classe é utilizada para executar as consultas e comandos no SGDB.
- `java.sql.ResultSet`: essa classe é utilizada para recuperar os dados que foram buscados, por exemplo, um comando de *select*.

- `javax.sql.DataSource`: essa classe é utilizada para agrupar conexões com o SGBD.

Para se conectar a um banco de dados, esteja ele implementado em qualquer SGBD, é necessário criar uma *string* de conexão, ou URL JDBC (*Uniform Resource Locator* JDBC). Essa *string* informará o "caminho" do banco e apresenta a seguinte sintaxe:

jdbc:<*driver*>:<detalhes da conexão>

No item <*driver*> especifica-se qual SGBD será utilizado para conexão. Alguns exemplos são ilustrados no Quadro 1.15.

Quadro 1.15 | Exemplos de URLs JDBC

Banco de dados	URL JDBC
MySQL	jdbc:mysql://localhost:3306/nomeBancoDeDados
SQL Server	jdbc:sqlserver://localhost;databaseName=nomeBancoDeDados
Oracle	jdbc:oracle:thin@myserver:1521:nomeBancoDeDados

Fonte: elaborado pelo autor.

Já no item <detalhes da conexão> são especificados detalhes da conexão, como por exemplo o servidor, a porta de acesso, o nome do banco de dados, etc.

Conexão com banco de dados

O código no Quadro 1.16 apresenta como fazer a conexão ao MySQL utilizando o JDBC do Java. Entre as linhas 1 a 3 são incluídas algumas classes do JDBC que são necessárias para criar a conexão com o banco de dados. Na linha 6, a variável *URLDB* armazena a URL para o MySQL; cada SGBD apresenta formas diferentes de montar as informações de endereço e porta, nesse caso, especificamos o usuário local (*localhost*) com a porta padrão do MySQL 3306 e nome do banco de dados "*nomebd*".

Assimile

Os parâmetros da URL alteram conforme o SGBD usado, bem como com as configurações dos sistemas. No Quadro 1.16 apresentamos uma forma genérica "**jdbc:mysql://localhost:3306/nomebd**". Pode ser necessário inserir mais informações, como por exemplo habilitar explicitamente o certificado SSL e o *Time Zone*, ficando com a seguinte sintaxe:

"jdbc:mysql://localhost:3306/nomebd?useSSL=false&serverTimezone=UTC".

As linhas 7 e 8 representam o usuário e senha para a conexão, a linha 12 informa qual é o *driver* a ser carregado e a linha 13, o método `getConnection`(URLDB, usuario, senha), da classe `DriverManager`, faz a conexão utilizando todas as informações.

O comando *Class.forName* carrega uma classe específica, que fica à disposição para o ambiente que está sendo utilizado. No caso do JDBC, quando é feita a conexão com uma URL específica, ele busca a classe que acha mais indicada (WINDER, 2009). Para que o comando `Class.forName` tenha sucesso é necessário, em primeiro lugar, fazer o donwload do *driver* que se deseja usar. Por exemplo, para o MySQL você deverá acessar o endereço <https://www.mysql.com/products/connector/> (ORACLE CORPORATION, 2018a) e selecionar a opção **Connector/J**. No momento em que esse livro está sendo escrito, o conector está na versão 8.0.11, e essa informação é relevante pois pode impactar os parâmetros de conexão da URL. Esse arquivo poderá ser descompactado em qualquer local no computador. Com o arquivo já no computador, o próximo passo é fazer a devida configuração no Eclipse, e para isso você deve adicionar o caminho no sistema de arquivos na variável de ambiente CLASSPATH. No caso do Eclipse, é possível adicionar esse JAR como item do projeto: na aba *Package Explorer*, clique com o botão direto do mouse e selecione *Properties*, na opção *Java Build Path* selecione *Libraries* e depois clique em *Add External JARS*, e então adicione do local onde descompactou o conector no seu computador.

Quadro 1.16 | Conexão ao SGBD utilizando o JDBC

```
1.  import java.sql.Connection;
2.  import java.sql.DriverManager;
3.  import java.sql.SQLException;

4.  public class ConexaoBancoDeDados {

5.          private Connection conexao;
6.          private final String URLDB = "jdbc:mysql://
            localhost:3306/nomebd";
7.          private final String usuario = "user";
8.          private final String senha = "senha";

9.          public ConexaoBancoDeDados()
10.         {
11.                 try {
12.                         Class.forName("com.mysql.cj.
jdbc.Driver");
13.                         conexao  =  DriverManager.get-
Connection(URLDB, usuario, senha);
14.
15.                 } catch (Exception e) {
16.                         e.printStackTrace();
17.                 }
18.         }
            public static void main(String[] args) {
19.
20.                 ConexaoBancoDeDados conexao = new
21. ConexaoBancoDeDados();
            }
}
```

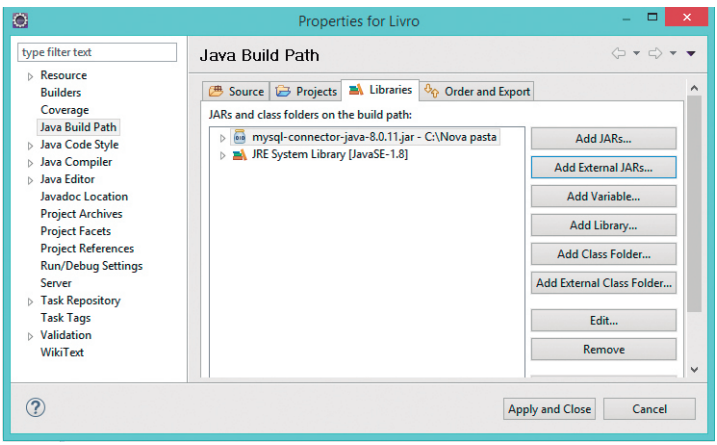
Fonte: elaborado pelo autor.

Reflita

Em um sistema que utiliza um SGBD é necessário ter acesso a informações para fazer a conexão, tais como IP, usuário, senha e nome do banco a ser conectado. Todos esses dados devem estar disponíveis no código fonte. Esses dados estão seguros no código fonte compilado? Se alguém inspecionar o código compilado poderá obter as informações?

A Figura 1.9 apresenta as propriedades do projeto no Eclipse, no qual é possível verificar que elas já foram adicionadas ao JAR para conexão ao MySQL. Com isso, ao executar o método *Class.for,* a classe "*com.mysql.cj.jdbc.Driver*" ficará disponível para o JDBC.

Figura 1.9 | Propriedades do projeto no Eclipse



Fonte: captura de tela do software Java elaborada pelo autor.

Ações no banco de dados: inserção, leitura, atualização e exclusão de registros

Com a classe do MySQL disponível será possível fazer a conexão com o banco de dados utilizando a classe `Connection`. Por exemplo, no código do Quadro 1.16 o objeto `conexao` da classe `Connection` garante o acesso a diversas ações no banco de dados.

As ações no banco de dados são construídas com a classe `Statement`. Essas ações são conhecidas como CRUD (*Create,*

Read, Update, Delete), pois representam as quatro operações possíveis em um banco de dados. Para seu uso é preciso adicionar a referência "`java.sql.Statement;`".

O código no Quadro 1.17 demonstra como inserir elementos no banco. Na linha 2 é criado um objeto do tipo `Statement` chamado `inserir`, e na linha 3 o comando de inserção é executado. Veja que os comandos são em linguagem SQL.

Quadro 1.17 | Execução de inserção em um SGBD

```
1.  import java.sql.Statement;

2.  Statement inserir = conexao.createStatement();
3.  inserir.execute(("INSERT INTO pessoa
    (nome,sobrenome,nivelusuario) VALUES
    ('NomePessoa','Sobrenome',1)");
```

Fonte: elaborado pelo autor.

O processo para as demais operações é análogo: cria-se um objeto do tipo `Statement` e faz-se a execução. Como exemplo, o código no Quadro 1.18 apresenta as demais opções. Na linha 1 é criado o objeto do tipo `Statement`. Na linha 2, usamos o comando SQL para criar uma instrução de atualização na coluna "nome" da tabela "pessoa" que apresenta o `id` 1. Na linha 3, o comando exclui o registro da tabela "pessoa" cujo `id` é 1. Na linha 4, foi criado um novo objeto chamado `res` do tipo `ResultSet`, para armazenar o resultado da seleção dos campos nome e sobrenome da tabela "pessoa". Para utilizar essa classe, é necessário incluir referência a `java.sql.ResultSet`. Veja que foi necessário usar o método `executeQuery` para fazer a seleção. Das linhas 5 a 8 criamos uma estrutura de repetição para percorrer o objeto que armazenou o resultado e imprimir os campos na tela. Para que seja possível buscar todos os dados é necessário executar o método "`next()`".

Quadro 1.18 | Execução de um *update* na tabela pessoa

```
1.  Statement comando = conexao.createStatement();
    //comando para atualizar
2.  comando.execute("UPDATE pessoa SET nome = 'Nome'
    WHERE id = 1");
    //comando para deletar
3.  comando.execute("DELETE FROM pessoa WHERE id =
    1");
    //comando para selecionar dados
4.  ResultSet res = comando.executeQuery("SELECT *
    FROM pessoa");

    //comando para exibir os dados lidos
5.  while(res.next())
6.  {
7.          System.out.println(res.getString("nome"));
8.  }
```

Fonte: elaborado pelo autor.

Exemplificando

O `ResultSet` pode retornar todos os tipos de dados que são definidos nas tabelas do banco de dados. Como exemplo, é possível recuperar `int` com o comando `getInt("id")`, e assim há a possibilidade de recuperação de um inteiro do banco de dados. Em alguns SGBD é viável armazenar arquivos em binário do tipo *blob*, que representam campos binários no banco de dados, e para recuperar, usa-se o método `getBlob` do `ResulSet`.

Portanto, todo o processo de manipulação de dados utilizando o JDBC consiste na criação de *strings* utilizando os comandos SQL. Para facilitar esse processo, é possível utilizar a classe `PreparedStatement`, e com ela define-se qual comando SQL será utilizado e marca-se com "?" os locais na *string* em que os valores serão inseridos no comando. Ainda na classe `PreparedStatement` pode-se utilizar os comandos `setString`, `setInt` e outros para preencher os campos do `insert`, o que facilita a criação de instruções de forma mais organizada do que com o `Statement`. Veja um exemplo dessa classe no Quadro 1.19: na linha 1 é criado

um objeto chamado "psInsert" do tipo *PreparedStatement*, que apresenta um comando SQL (insert). Nas linhas 2 a 4 são definidos os valores que serão substituídos no objeto psInsert nos locais com "?", ou seja, o compilador executará o seguinte insert: "INSERT INTO (nome, sobrenome,nivelusuario) VALUES ('Nome','Sobrenome',1)", e por fim, na linha 5, o comando será executado.

Quadro 1.19 | Executar comandos utilizando a classe Statement

```
1. PreparedStatement psInsert = con.prepareStatement("INSERT INTO (nome, sobrenome,nivelusuario) VALUES (?,?,?)");
2. psInsert.setString(1, "Nome");
3. psInsert.setString(2, "Sobrenome");
4. psInsert.setInt(3, 1);
5. psInsert.execute();
```

Fonte: elaborado pelo autor.

Portanto, com os comandos *Statement*, *PreparedStatement* e *ResultSet* é possível enviar e receber dados do banco de dados. É importante lembrar-se da coesão das classes no modelo orientado a objetos, portanto as classes que utilizam o banco de dados não devem fazer outras tarefas. . Isso garante a coesão do sistema e aumenta as possibilidades de sua manutenção.aumenta as possibilidades de sua manutenção.

Pesquise mais

A Oracle mantém uma documentação com diversos exemplos que podem ser utilizados para estudo e aperfeiçoamento. Acesse os endereços a amplie seus conhecimentos.

- Informações sobre o JDBC: ORACLE CORPORATION. Lesson: **JDBC Basics.** [S. l.], 2017. Disponível em: <https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>. Acesso em: 12 abr. 2018.
- Informações sobre o conector MySQL: ORACLE CORPORATION. **Chapter 4 Connector/J Installation.** [S. l.], 2018. Disponível em: <https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-installing.html>. Acesso em: 12 abr. 2018.

- CURSO EM VÍDEO. **Curso MySQL #3 -** Criando o primeiro Banco de Dados. [S. l.], 29 fev. 2016. Disponível em: <https://www.youtube.com/watch?v=m9YPlX0fcJk>. Acesso em: 8 maio 2018.

Sem medo de errar

Trabalhando em uma empresa que tem como projeto a atualização de um sistema legado universitário, chegou o momento de realizar a implementação necessária para fazer persistência dos dados referentes à primeira interface gráfica do sistema que substituirá o sistema legado da universidade.

Esse sistema deve ser tratado com precisão, pois um problema na migração pode causar insatisfação e críticas que são capazes de forçar a continuidade do sistema antigo. Esta última etapa consiste em fazer a persistências dos dados que a interface recolhe dos usuários, e para isso deve-se criar duas classes: uma que faça a conexão e outra que apresente os comandos SQLs. A primeira classe "`ConexaoBancoDeDados`" deve:

- Ter como atributo da classe *Connection* do JDBC para controlar a conexão com o SGBD:

  ```
  private Connection conJDBC;
  ```

- Implementar três atributos que disponham dos dados da conexão:

  ```
  private final String URLDB = "jdbc:mysql://localhost:3306/nomebd";
  ```

  ou dependendo da versão do driver utilizada:

  ```
  private final String URLDB = "jdbc:mysql://localhost:3306/nomebd?useSSL=false&serverTimezone=UTC";
  ```

  ```
  private final String usuario = "user";
  ```

  ```
  private final String senha = "senha";
  ```

- No construtor, carregar o *driver* e instanciar o objeto da classe *Connection*:

  ```
  Class.forName("com.mysql.cj.jdbc.Driver");
  ```

  ```
  conJDBC = DriverManager.getConnection(URLDB, user, senha);
  ```

- Criar um método para fornecer acesso à classe *Connection*:

```
public Connection getConexao()
{
        return conJDBC;
}
```

A segunda classe, "ComandoSQL", será responsável por utilizar a classe anterior e criar métodos para fazer a inserção dos dados:

- Crie um objeto da classe " ConexaoBancoDeDados":

```
private ConexaoBancoDeDados conexaoBanco;
```

No construtor da classe, crie uma instância da conexão:

```
public ComandoSQL() {
  conexaoBD = new ConexaoBancoDeDados();
}
```

- Após o construtor, crie um método que receba os dados a serem inseridos no banco utilizando um elemento do `Statement`:

```
public void insereDados(String nome,String endereco,String telefone,String CPF,String tipoSangue,String fatorRH,String nomeEmergencia,String telefoneEmergencia)
```

- Nesse método, utilizar todos os campos para serem inseridos por um `PreparedStatement`:

```
PreparedStatement psInsert = conexaoBanco.getConexao().prepareStatement("INSERT INTO aluno (nome,endereco,telefone,CPF,tipoSangue,fatorRH,nomeEmergencia,telegoneEmergencia) VALUES ?,?,?,?,?,?,?,?)");
```

- Na interface gráfica, crie uma instância da classe "ComandoSQL" chamada "acessoBD;":

```
private ComandoSQL acessoBD;
```

Instancie o objeto no construtor da interface gráfica:

```
public interfaceGrafica()
{
  acessoBD = new ComandoSQL();
```

```
        //... demais comandos para construção da tela
    }
```

- Utilize o método que foi criado para inserção no banco de dados em um evento de clique no botão:

```
acessoBD.insereDados(nome, endereco, telefone, CPF, tipoSangue, fatorRH, nomeEmergencia, telegoneEmergencia);
```

Agora que você completou o projeto da primeira interface funcional do novo sistema universitário, será possível migrar todos os outros módulos e garantir que esse novo sistema possa ser utilizado durante muitos anos. Porém, ainda é necessário pensar em outros módulos de processamento que necessitam de migração ou criação. Logo, mais conhecimentos serão demandados para garantir o correto funcionamento do sistema.

Avançando na prática

Sistema de cadastro de dados pessoais.

Descrição da situação-problema

Uma empresa está produzindo um cadastro básico de pessoas, contando com os dados de nome, endereço e telefone. Seu papel é organizar o *back end* dessa aplicação para a manipulação de dados e, para isso, é necessário produzir um conjunto de métodos que façam a inserção, alteração e remoção de dados

Resolução da situação-problema

É possível desenvolver a classe do Quadro 1.20.

Quadro 1.20 | Conexão para sistema de cadastro de dados

```
import java.sql.*;

public class CadastroPessoas {

        private Connection conexao;
        private final String URLDB = "jdbc:mysql://lo-
```

```
        calhost:3306/nomebd?useSSL=false&serverTimezo-
        ne=UTC";
        private final String usuario = "usuario";
        private final String senha = "senha";

        public CadastroPessoas() {
             try {
                   Class.forName("com.mysql.cj.jdbc.
                   Driver");

                  //criando a conexão com o BD
                   conexao   =   DriverManager.getConnec-
tion(URLDB, usuario, senha);
                  System.out.println("Conexão criada");

             }catch(Exception ex) {
              ex.printStackTrace();
             }
        }

        public boolean inserirPessoa(String nome,String
        endereco, String telefone)
        {
               try {
                     PreparedStatement psInsert =  cone-
xao.prepareStatement("INSET INTO pessoa (nome,endere-
co,telefone) VALUES (?,?,?)");
                    psInsert.setString(1,nome);
                    psInsert.setString(2,endereco);
                    psInsert.setString(3,telefone);
                    return psInsert.execute();

               } catch (Exception e) {
                    e.printStackTrace();
                    return false;
               }
        }

        public boolean alterarPessoa(String nome,String
endereco, String telefone,int id)
         {
             try {
```

```
        PreparedStatement psInsert = conexao.prepareS-
tatement("UPDATE pessoa SET nome=?,endereco=?,telefo-
ne=?) WHERE id=?");
        psInsert.setString(1,nome);
        psInsert.setString(2,endereco);
        psInsert.setString(3,telefone);
        psInsert.setInt(4,id);
        return psInsert.execute();

          } catch (Exception e) {
                e.printStackTrace();
                return false;
          }
   }
   public boolean removerPessoa(int id)
   {
          try {

               PreparedStatement psInsert =
    conexao.prepareStatement("DELETE    pessoa    WHERE
id=?");
               psInsert.setInt(1,id);
               return psInsert.execute();

           } catch (Exception e) {
                e.printStackTrace();
                return false;
           }
    }
}
```

Fonte: elaborado pelo autor.

Faça valer a pena

1. Para criar o acesso ao banco de dados é possível utilizar bibliotecas específicas, com isso se garante a forma correta de acesso para cada banco. Todavia, é possível utilizar o *Java Database Connectivity* (JDBC), que dispõe de um conjunto de elementos para garantir que a aplicação possa acessar diversos sistemas de banco de dados de maneira padronizada.

Quais dos elementos a seguir, integrantes do JDBC, garantem a conexão a diversos bancos de dados com a mesma interface?

a) *Driver* e *DriverManager*.

b) *Connection e Statement*.

c) *PreparedStatement e Statement*.

d) *URL e Password*.

e) *com.mysql.jdbc.Driver e jdbc:mysq.l*

2. Para conectar a um sistema de gerenciamento de banco de dados (SGBD) utilizando o JDBC, é necessário indicar qual *driver* utilizar. No momento em que o JDBC configura o objeto *Connection*, é preciso comunicar um conjunto de informações mínimas para garantir a utilização da conexão que está sendo criada.

Dos itens a seguir, quais deles representam as informações mínimas para conectar em uma instância de um banco de dados MySQL?

a) IP, usuário e senha.

b) DNS e usuário.

c) URL contendo jdbc:mysql://localhost:3306/, usuário e senha.

d) URL contendo jdbc:mysql://localhost:3306/bd, usuário e senha.

e) Usuário e senha.

3. Ao utilizar o JDBC é possível executar comandos SQL com diversas classes; duas delas são `Statement` e `PreparedStatement`. As duas classes dispõem da função básica de executar comandos SQL, todavia apresentam diferenças na maneira de informá-los.

Dadas as afirmações a seguir:

I. A classe `Statement` pode receber apenas comandos de inserção.

II. As classes `Statement` e `PreparedStatement` não devem ser utilizadas com o MySQL.

III. A classe `PreparedStatement` consegue separar os comandos SQL de seus valores.

IV. O `PreparedStatement` não pertence às classes padrão do JDBC.

V. A classe `PreparedStatement` fornece uma organização mais precisa do código do que a `Statement` na execução de inserção de dados.

Escolha a opção que apresente apenas as **afirmações corretas:**

a) Apenas as afirmações I e II estão corretas.

b) Apenas as afirmações III e V estão corretas.

c) Apenas as afirmações I, II e V estão corretas.

d) Apenas as afirmações I, II, III e V estão corretas.

e) Todas as afirmações estão corretas.

Referências

DEITEL, Paul; DEITEL, Harvey. **Java - como programar**. São Paulo: Pearson, 2016.

FURGERI, Sérgio. **Java 8** - Ensino Didático - Desenvolvimento e Implementação de Aplicações. São Paulo: Érica, 2015.

HORSTMANN, Cay S. **Core Java Volume I** - Fundamentals. New York: Prentice Hall, 2016.

MANZANO, José Augusto G., COSTA JR., Roberto da. **Programação de Computadores com Java**. São Paulo: Érica, 2014.

MICROSOFT CORPORATION. **SQL Server 2016**. [S. l.], 2018. Disponível em: <https://www.microsoft.com/en-us/sql-server/sql-server-2016>. Acesso em: 15 abr. 2018.

ORACLE CORPORATION. **MySQL Connectors**. [S. l.], 2018a. Disponível em: <https://www.mysql.com/products/connector/>. Acesso em: 4 maio 2018.

_____. **MySQL Downloads**. [S. l.], 2018b. Disponível em: <https://www.mysql.com/downloads/>. Acesso em: 7 maio 2018.

_____. **My SQL**. [S. l.], 2018. Disponível em: <https://www.mysql.com/>. Acesso em: 15 abr. 2018.

_____. **Oracle Database**. [S. d., s. l.]. Disponível em: <https://www.oracle.com/database/index.html>. Acesso em: 15 abr. 2018.

THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. **PostgreSQL**: the world's most advanced open source relational database. [S. l.], 2018. Disponível em: <https://www.postgresql.org/>. Acesso em: 15 abr. 2018.

WINDER, Russel; GRAHAM, Roberts. **Desenvolvendo Software em Java**. Rio de Janeiro: LTC, 2009.

Programação concorrente orientada a objetos

Convite ao estudo

Depois de muitos anos de pesquisa, desde o descobrimento do silício (matéria prima dos processadores), em 1971 uma empresa chamada Intel lançou o primeiro microprocessador chamado 4004 (HOPE, 2017). A partir desse marco, iniciou-se uma corrida entre os fabricantes para construírem processadores cada vez mais rápidos, mas essa evolução encontrou uma barreira, pois aumentar a velocidade dos processadores possui um limite físico que, para ser superado, é preciso fabricar processadores maiores e com maior dissipação de calor. Para superar essa limitação, os fabricantes passaram a construir processadores com mais de um núcleo (core), dessa forma, aumenta-se o poder de processamento, sem aumentar o tamanho.

Assim, computadores, smartphones, tablets e diversos outros dispositivos possuem mais de um núcleo de processamento. Essa arquitetura multinúcleo é muito interessante para o sistema operacional, todavia, para que uma aplicação tire proveito dos diversos núcleos, é necessário utilizar elementos de programação para que seja possível ter mais de uma linha de processamento. Além da relação de diversos núcleos de processamento, ainda é necessário que um software consiga executar mais de uma ação no mesmo processo, como fazer o download de uma aplicação e informar o tempo restante. Portanto, para explorar melhor o hardware, é necessário utilizar programação paralela, assunto central dessa unidade.

O objetivo dessa etapa de estudo é o desenvolvimento de aplicações Java™ utilizando programação concorrente

orientada a objetos. Para que isso seja possível, trataremos das formas de criação de linhas de processos paralelas à linha de execução principal do software de diversas maneiras. Além disso, veremos itens de sincronismo para tratar a concorrência no acesso à informação e também como tratar as possíveis exceções que um sistema paralelo pode gerar.

Trabalhando em uma empresa de software, depois de acumular experiência na construção de interfaces gráficas e demonstrar grande competência, você concorreu a uma vaga para trabalhar na equipe de *backend* (parte do software que recebe e processa as informações da interface com o usuário). Após uma extensa prova e uma ótima recomendação de seu antigo líder, você foi alocado nessa equipe. A primeira tarefa consiste em fazer o processamento das informações para o vestibular de uma universidade que está atualizando seus sistemas, com o objetivo de criar os relatórios que serão utilizados. Para o sucesso da tarefa será necessário fazer o tratamento de possíveis erros que o sistema possa gerar e armazenar todos esses dados em arquivos de texto. Dessa forma, quais classes e métodos na linguagem Java podem ser usados para se trabalhar com essas linhas de processamento independentes? Quais são as melhores práticas para esse paradigma de programação? Quais são as formas de tratamento de exceções para ambientes paralelos?

A primeira seção abordará as formas iniciais de criação de threads em Java, buscando elucidar os passos para a criação, inicialização e interrupção. A segunda seção apresentará as formas de se tratar as exceções em ambiente paralelo, para garantir a correta execução. Por fim, na terceira seção você verá os elementos que garantem o sincronismo de métodos paralelos, ou seja, asseguram que certas porções de código sejam acessadas por apenas uma thread de cada vez.

Esse conhecimento vai lhe proporcionar uma nova visão sobre o mundo da programação. Vamos lá?

Seção 2.1

Programação em Java usando threads

Diálogo aberto

Um sistema computacional é semelhante a uma empresa, em que diversas tarefas são executadas por diferentes pessoas e equipes, a fim de alcançar um determinado resultado. Assim são os softwares: diversos métodos são criados em diferentes classes para a execução de tarefas específicas. Além disso, já percebeu que diversos softwares parecem executar mais de uma ação por vez? Em um jogo, por exemplo, vários elementos podem interagir, mover e fazer ações diferentes ao mesmo tempo. Para que isso seja possível, é necessário que o processo seja dividido em diversas ações, então estudaremos os mecanismos de criação de threads.

Na empresa em que você trabalha, após um processo de promoção, você foi alocado na equipe que fará o *backend* de uma aplicação. Você continuará no processo de migração do sistema legado da universidade, porém agora fará parte da equipe de processamento das informações. A primeira tarefa consiste em processar as informações de inscrições para o vestibular. O sistema recebe um arquivo de texto contendo nome, curso pretendido e número de inscrição do candidato, com campos separados por ponto e vírgula, e as informações de candidatos diferentes são separadas por linha, como:

João; Ciência da Computação; 4567891

Pedro; Agronomia; 4567891

Cada lote possui cerca de 5000 linhas e há 250 lotes para o processamento, que consiste em ler cada um desses dados e gerar um relatório. Porém, a pessoa que fez a primeira versão percebeu que esse processo estava muito lento. Você foi designado pelo seu líder para solucionar esse problema, portanto, construa um código utilizando threads em Java. A thread deverá receber um diretório e nesta classe deixar os métodos prontos para a equipe de regras de negócio implementar o processamento. Com isso, será possível iniciar diversas threads para fazer o processamento de

vários diretórios. Como deve ser criado o processamento paralelo em Java? Quais recursos a linguagem disponibiliza para o controle desse mecanismo?

Nesta seção será apresentado o conceito de programação paralela usando threads, bem como a forma de inicialização e controle desse tipo de processamento e todo o conteúdo com foco em um código com alta coesão e baixo acoplamento.

Veja que agora você já pode utilizar mecanismos mais avançados de programação, gerando novos desafios e novas oportunidades. Use sua criatividade e visão para resolver esses problemas. Vamos produzir o código para extrair o máximo do hardware!

Não pode faltar

Para o desenvolvimento de um software, são necessários diversos conhecimentos sobre as maneiras de criação de interface gráfica, as diversas formas de conexão com o banco de dados e a utilização detalhada da orientação a objetos. É necessário usar corretamente a orientação a objetos para garantir a manutenção do código e a alta coesão e o baixo acoplamento dos métodos, classes e pacotes. Uma solução computacional tende a crescer e a se tornar mais complexa com o tempo, e todos os cuidados mencionados elevam as possibilidades de o sistema manter-se estável. Entretanto, para que novas demandas sejam atendidas, são necessárias mais técnicas e elementos de programação, a fim de propiciar o processamento e o acesso à informação de forma eficiente.

Com o avanço tecnológico, o hardware evoluiu para processadores que possuem diversos núcleos para a execução de instruções, levando novamente a necessidades mais específicas de programação. Esses cenários mais complexos precisam da execução de processos que tiram o máximo de rendimento do hardware, levando a um tempo menor de processamento ou ainda, à melhoria das formas de interação com o usuário.

Assimile

Os processadores dos computadores recebem atualizações constantes para aumentar a frequência e a memória *cache*. Todavia, um dos elementos que mais recebeu aumento foram os números de núcleos de processamento. Com isso, para usufruir deles, é necessário utilizar técnicas de computação paralela.

Pensando nesses cenários, podemos imaginar diversos casos em que é necessário usar mais de um núcleo ou executar duas tarefas diferentes ao mesmo tempo. Como exemplo, podemos pensar em um sistema que gere relatórios de um grande número de entradas ou ainda que demande um processamento estatístico com uma quantidade elevada de informações. A interface gráfica inicial de um sistema que gera esses relatórios pode ter apenas os campos com as informações que estarão no relatório, então, ao selecionar o item que inicia o processamento, o sistema faz a busca no sistema de gerenciamento de banco de dados e começa a processar a informação. É necessário informar o usuário sobre o progresso desse processamento. Porém, se o software está executando esse processo e precisa terminar o mais rápido possível, não deve parar exibir informações ao usuário.

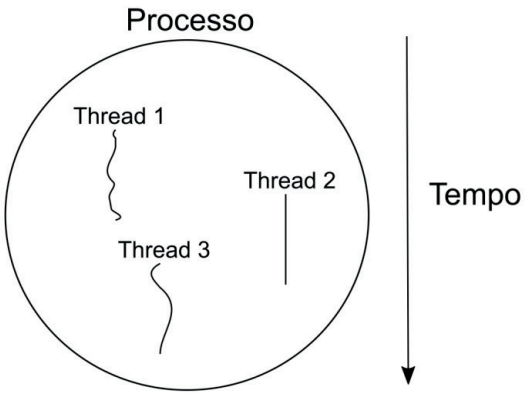
Observando as necessidades de aproveitar diversos núcleos de processamento, desenvolvendo softwares que usam o hardware de maneira mais eficiente, podemos pensar em duas abordagens. A primeira consiste em gerar diversos executáveis de um programa para processar as informações, porém, essa abordagem gera formas mais complexas e lentas de comunicação, como arquivos de texto que são lidos por diversas aplicações, comunicação pelas informações escritas no banco de dados ou comunicação pela rede. Uma forma mais eficaz de se tirar proveito de um processador com diversos núcleos ou ainda prover maneiras para que um software processe as informações e ainda informar o progresso para o usuário são as **threads**.

Cada programa em execução possui uma sequência de passos a serem realizados, ou seja, um fluxo de execução. Uma thread nada mais é do que um fluxo de execução ou, como muitas vezes é chamada, uma linha de execução dentro de um processo. Para

um software efetuar diversas tarefas ao mesmo tempo, é preciso utilizar os diversos núcleos do processador de forma paralela, ou seja, é preciso criar um programa que possua várias threads ou simplesmente *multithread* (DEITEL e DEITEL, 2016).

A Figura 2.1 representa um processo (unidade do sistema operacional que possui os dados de um software em execução) em que há diversas linhas de execução que podem processar funções diferentes para aproveitar hardware com diversos núcleos ou ainda utilizar uma linha de execução para processar informações e outra para informar o usuário sobre o progresso dessa tarefa.

Figura 2.1 | Representação de um processo com threads em um sistema operacional



Fonte: elaborada pelo autor.

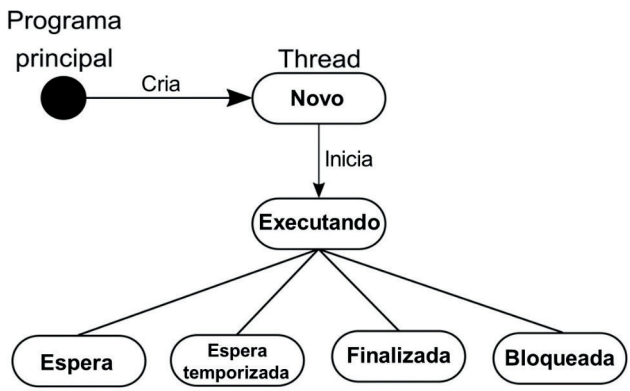
Reflita

Quando se pensa em computadores modernos, o nível de frequência dos processadores tem diminuído e o número de núcleos de processamento tem aumentado. Quais são os problemas e dificuldades dessa abordagem para a programação? Os cenários ficam mais ou menos complexos na implementação?

Utilização de threads na linguagem Java

Na linguagem Java, para que seja possível gerar essas linhas de execução, é necessário utilizar uma *Application Programming Interface* (API) que interaja com o sistema operacional, determine a criação das threads e faça as configurações necessárias (FURGERI, 2015). Além da criação, é preciso observar que as threads possuem estados de execução relativos ao tempo de processamento concedido pelo sistema operacional. A Figura 2.2 apresenta os passos de criação e os possíveis estados que uma thread pode assumir. O programa principal cria a thread e essa nova linha de execução fica em estado "Novo". Quando o sistema operacional escalona essa thread para execução, ela entra no estado "Executando", etapa em que as tarefas definidas para thread são efetuadas. A partir desse ponto é possível que ela passe para quatro outros estados: "Esperando", "Espera temporizada", "Finalizada" ou "Bloqueada" (DEITEL e DEITEL, 2016).

Figura 2.2 | Criação e estados de uma thread



Fonte: adaptada de Deitel e Deitel (2016, p. 960).

A thread entra estado de "Espera" quando, após executar por um certo tempo, o sistema operacional interrompe o processamento para permitir que as outras threads ou processos possam realizar suas tarefas. O estado "Espera temporizada" é acionado quando a própria thread pede para interromper a execução por certo tempo, e essa interrupção é importante para garantir que uma thread não

tente ocupar muito tempo de processamento (HORSTMANN, 2016). Certos momentos, uma thread pode requisitar alguma operação que necessita de ações relacionadas à entrada e saída do sistema. Com isso, é feita uma chamada ao sistema operacional para disponibilizar esse recurso (TANENBAUM e BOS, 2016) que, enquanto não está pronto e disponibilizado pelo sistema operacional, deixa a thread no estado "Bloqueada", somente voltando a executar quando o recurso está liberado. Por fim, o estado "Finalizada" se refere a quando o processo que controla a thread determina seu fim e pede para o sistema operacional terminar a execução.

Para criar threads na linguagem Java, é possível utilizar diversas classes que fazem a abstração das linhas de execução de dentro de um processo. O Quadro 2.1 apresenta as duas mais comuns.

Quadro 2.1 | Classe que podem ser utilizadas para criar threads

Classe	Descrição
java.lang.Thread	Classe inicial que fornece a utilização básica de threads.
java.util.Timer	Classe que fornece a utilização de thread com elementos de repetição e controle encapsulados.

Fonte: elaborado pelo autor.

Para se criar uma thread, são necessários 3 passos:

1. Criar uma classe que implementa a interface `Runnable` e o método `run()`.
2. Criar uma instância tipo Thread e indicar a classe que implementa a interface `Runnable`.
3. Iniciar a thread com o método `start()`.

Para visualizar essas implementações a melhor forma é um exemplo de código, vamos ver como criar essas threads? No Quadro 2.2 a classe `Processador` implementa a interface `Runnable`, que determina a criação do método `run()`, que será uma nova linha de execução dentro desse processo. A linha 2 declara um objeto da classe `Thread` chamado `th` que faz a abstração para se criar as threads. Na linha 4 esse objeto é instanciado, informando que a própria classe `Processador` possui o método `run()` (`th = new Thread(this);`) e, por fim, a thread é iniciada (`th.start();`). A

linha 6 é o método que será executado quando a thread for iniciada e, finalmente, a linha 7 cria uma instância da classe `Processador` no método `main`, dando início ao software.

Quadro 2.2 | Exemplo de utilização das threads em Java

```
     package U2S1;
1.   public class Processador implements Runnable{
2.         private Thread th;
           public Processador()
3.         {
                th = new Thread(this);
4.              th.start();
5.         }
           public void run() {
6.              for (int i = 0; i < 1000; i++) {
                          System.out.println("Processando
dados "+i);
                }
           }
                public static void main(String[] args)
7.              Processador p = new Processador();
           }
     }
```

Fonte: elaborado pelo autor.

O Quadro 2.3 apresenta o resultado da execução do código da thread que acabamos de verificar. Ainda no código, repare que em nenhum momento é chamado o método `run()`, ele apenas é indicado na criação da thread `(th = new Thread(this);)`. Com isso, foi criada uma linha de execução separada da linha principal.

Quadro 2.3 | Resultado da execução do processo da thread

```
Processando dados 0
Processando dados 1
Processando dados 2
Processando dados 3
Processando dados 4
Processando dados 5
...
Processando dados 999
```

Fonte: elaborado pelo autor.

Exemplificando

A utilização de threads em sistemas modernos é comum, e alguns cenários são mais propícios para sua utilização, como:

- Jogos utilizam threads para atribuir tarefas de comunicação de redes, controle de mouse, de teclado ou de joystick ou renderização.
- Sistemas *Enterprise Resources Planning* (ERP) utilizam para a geração múltiplos relatórios ou emissão de notas fiscais em paralelo.
- Sistemas embarcados utilizam para leitura de sensores ou para monitores de atividade (*watchdogs*).

Aplicação usando thread na linguagem Java

Um cenário para aplicação de threads são os *watchdogs*, softwares que têm o objetivo de monitorar recursos e aplicações. Como exemplo, podemos citar o monitoramento de uma conexão com o sistema de gerenciamento de banco de dados ou ainda a execução de um software de controle de câmeras de segurança. Os *watchdogs* ficam monitorando as aplicações e, em caso de problemas, podem gerar um alerta ou forçar a reinicialização do software ou do computador. Vamos ver no Quadro 2.4 um código completo que utiliza thread para monitorar um recurso.

As linhas 1 e 2 definem os arquivos a serem incluídos e a qual pacote a classe pertence. A configuração da thread é feita das linhas 3 a 8, onde criam-se os elementos básicos da thread, que consistem

em implementar a interface `Runnable` (linha 3), declarar o objeto da classe `Thread` (linha 4), declarar e atribuir estado da variável booleana para monitoramento (linhas 5 e 7), instanciar o objeto do tipo `Thread` e indicar qual classe possui o método `run()` (linha 8). O processo de controle de uma thread consiste em formas de iniciar e interromper essa linha de execução. O método da linha 9 **`public void`** `iniciar()` inicia a thread chamando o método `th.start();` e o método da linha 11 **`public void`** `parar()` força a interrupção da thread. O processo de interrupção necessita de dois passos: `interrupt()` e `join()`. Na linha 12 a variável booleana de controle é definida como `false` e na linha 13 é chamado o método `th.interrupt();`, que declara que essa thread deve ser parada, mas isso deve ser feito pela própria implementação no método que a thread executa (repare que o método `run()` na linha 20 verifica se a thread for interrompida e, nesse caso, o método `run()` para de executar). O próximo passo consiste em chamar o método `th.join(2000);` na linha 15, que espera 2000 milissegundos até que a thread finalize sua execução. Repare que ele utiliza o sistema de tratamento de exceção nas linhas 14, 16 e 17. Utilizar o `join` é importante para garantir que a thread termine seu processamento e possa reportar sua finalização ao processo principal.

O método `run()` é aquele que a thread utiliza para executar suas tarefas e inicia exibindo uma mensagem, em seguida inicia-se o processo de monitoramento através da estrutura **`while`**. Quando a thread é interrompida pelo método `parar()`, a execução do `run()` entra na linha 22, definindo a interrupção do processamento. Em alguns casos, a thread pode não mostrar essa mensagem, pois é possível que esteja em `sleep`. Repare que na linha 26 é utilizado o `Thread.`*`sleep`*`(1000);`, forçando a thread a entrar em estado de espera temporizada, evitando usar muitos recursos e permitindo que outros processos possam ser utilizados. Nesse método `run()` devem ser implementadas as regras e o código que serão executados pela thread. Repare que o `sleep` está utilizando o tratamento de exceção (**`try`** e **`catch`**), para que, caso esse método seja interrompido, a thread também seja.

No método `main`, nas linhas 30 e 31 é inicializado o monitoramento e como exemplo foi utilizado a classe `Scanner` (linha 32) para ler os dados do teclado para perguntar ao usuário (linha 36). Na linha 37 há a reposta do usuário comparada, ignorando se o texto é maiúsculo

ou minúsculo, e verificando se deve continuar o monitoramento. Caso seja interrompido, é chamado o método `monitor.parar();`.

Quadro 2.4 | Utilização de thread em um *watchdog*

```
1.  package U2S1;

2.  import java.util.Scanner;

3.  public class Monitor implements Runnable{

4.      private Thread th;
5.      private boolean monitorando;

6.      public Monitor()
        {
7.          monitorando = true;
8.          th = new Thread(this);
        }
9.      public void inciar()
        {
10.         th.start();
        }
11.     public void parar()
        {
12.                     monitorando = false
13.         th.interrupt();
14.         try {
15.             th.join(2000);
16.         } catch (InterruptedException e) {
17.             e.printStackTrace();
            }
        }
18.     public void run() {
19.         System.out.println("Iniciando
    monitoramento.");
20.         while(monitorando == true)
21.         {
```

```
22.             // verifica se sistema alvo ainda está em execução
23.             System.out.println("Monitorando.");
24.             if(th.isInterrupted() == true)
                {
25.                 System.out.println("Parando
26. monitoramento.");
27.                 return;
28.             }
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    monitorando = false;
29.           }
30.        }
31.
32.      }
33.      public static void main(String[] args) {
34.          Monitor monitor = new Monitor();
35.          monitor.inciar();
36.          Scanner sc = new Scanner(System.in);
37.          boolean monitorar = true;
             do {
38.              System.out.println("Continuar
monitoramento S/N?");
                 String resp = sc.next();
40.              if(resp.equalsIgnoreCase("N") ==
true)
41.              {
                     monitorar = false;
                     monitor.parar();
                 }
             }while(monitorar == true);
             sc.close();
      }
}
```

Fonte: elaborado pelo autor.

O resultado da execução está no Quadro 2.5. Repare que, após o início do monitoramento, as respostas para "*Continuar monitoramento S/N?*" ficam fora de ordem em relação às mensagens "Monitorando". Isso ocorre porque a thread que faz o monitoramento possui uma linha e execução separada da parte que faz a leitura do teclado.

Quadro 2.5 | Resultado da execução do *watchdogs*

```
Iniciando monitoramento.
Monitorando.
Continuar monitoramento S/N?
Monitorando.
Monitorando.
S
Continuar monitoramento S/N?
Monitorando.
Monitorando.
Monitorando.
N
```

Fonte: elaborado pelo autor.

O uso de threads é essencial para seja possível produzir códigos de qualidade que utilizam os recursos do processador de forma otimizada. A implementação desses recursos cria diversas possibilidades para interações, desempenho e monitoramento. Todavia, como é uma execução paralela, deve-se planejar e manter a organização do código para evitar problemas de alto uso de recursos e problemas de sincronismo.

Pesquise mais

A programação paralela é um tópico que apresenta certa complexidade, pois é necessário pensar que diversas tarefas serão executadas ao mesmo tempo e, com isso, é interessante a consulta de diversas fontes:

- Documentação e exemplos da utilização da API para threads do Java, disponível em: <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>. Acesso em: 17 jul. 2018.

- DRAKE, D. G. Introduction to Java threads. JavaWorld, 1 abr. 1996. Disponível em: <https://www.javaworld.com/article/2077138/java-concurrency/introduction-to-java-threads.html>. Acesso em: 17 jul. 2018.
- CARLOS HENRIQUE JAVA. **Aula de Java 063 – Thread, execuções simultâneas**. [S.l.], 9 dez. 2014. Disponível em: <https://www.youtube.com/watch?v=pyldfRC6pKE>. Acesso em: 17 jul. 2018.

Sem medo de errar

Dando continuidade à implementação do sistema universitário, sua tarefa na equipe de desenvolvimento de *backend* consiste em produzir um código capaz de processar diversos arquivos utilizando threads.

O Quadro 2.6 apresenta o código que implementa o processamento de diversos arquivos. Lembrando que toda execução inicia-se pelo método `main()`, nas linhas 39 e 41 são criados dois objetos da classe `ProcessadorArquivos`, passando como parâmetros o caminho de duas pastas na raiz do diretório C:\\. A classe instanciada possui em sua estrutura o uso de threads e, para sua manipulação, foram criados três métodos: o `iniciar()` na linha 13, o `parar()` na linha 15 e o `run()` na linha 21, com as instruções para execução da thread. Vejamos com mais detalhes essa classe.

A linha 1 define a classe que implementa a interface `Runnable`, que força a implementação do método `run()` e a linha 2 declara a classe que representa thread no código. Entre as linhas 3 a 5 são declarados os atributos que definem o caminho no sistema de arquivos onde estarão os textos a serem processados, a quantidade total de arquivos no diretório que será informado e o arquivo atual a ser processado.

A linha 6 define o construtor que tem de receber o caminho dos arquivos e a linha 7 define que a thread utiliza o método `run()` da própria classe `(th = new Thread(this);)`.

Os métodos das linhas 9 a 12 são utilizados para resgatar o arquivo atual e a quantidade. As linhas 13 e 15 possuem os métodos para iniciar e parar a thread. A linha 21 define o método `run()`, tendo como primeiro comando (linha 22) listar os arquivos do diretório passado na instanciação da classe `ProcessadorArquivos (File`

`diretorio = new File(caminhosArquivos)`. Na linha 23 criou-se um vetor para guardar a lista de arquivos daquele diretório `(File[] listaDeArquivos = diretorio.listFiles();)`. Na linha 24 é armazenada a quantidade de arquivos no atributo `qtdFiles`. A estrutura de repetição que inicia na linha 25 percorre todos os arquivos do diretório e, para cada um, chama o método `processaArquivos()`, passando como parâmetro o diretório e o arquivo. Na linha 30, é impressa na tela uma mensagem de controle.

O método `processaArquivo()` na linha 31 recebe como parâmetro arquivo por arquivo. Dentro desse método poderiam ser criados diversos tratamentos pela equipe de controle. No nosso exemplo, estamos apenas imprimindo na tela a primeira linha de cada arquivo com o comando `in.nextLine()`.

É importante você ter em mente que a utilização de duas threads no método `main()` fará com que os arquivos sejam processados paralelamente por diferentes núcleos, tendo um aumento significativo de performance.

Quadro 2.6 | Processador de arquivos utilizando thread

```
    package U2S1;
    import java.io.File;
    import java.io.FileNotFoundException;
    import java.io.FileReader;
    import java.util.Scanner;

1.  public class ProcessadorArquivos implements
    Runnable{
2.      private Thread th;
3.      private String caminhosArquivos;
4.      private int qtdFiles;
5.      private int arquivoAtual;

6.      public ProcessadorArquivos(String
    pCaminhosArquivos)
7.      {
8.          th = new Thread(this);
            caminhosArquivos = pCaminhosArquivos;
```

```
9.        }
          public int getAtual()
10.       {
               return arquivoAtual;
11.       }
          public int getTotal()
12.       {
               return qtdFiles;
13.       }
          public void inciar()
14.       {
               th.start();
15.       }
          public void parar()
16.       {
17.            th.interrupt();
18.            try {
19.                 th.join(1000);
20.            } catch (InterruptedException e) {
                    e.printStackTrace();
               }
          }
21.
          public void run() {
22.
23.            File diretorio = new
File(caminhosArquivos);
24.            File[] listaDeArquivos = diretorio.
listFiles();
25.            qtdFiles = listaDeArquivos.length;
26.
27.            for (int i = 0; i < listaDeArquivos.
28.  length; i++) {
29.                 arquivoAtual = i;
                    if (listaDeArquivos[i].isFile()) {
30.                      String nomeArquivos =
listaDeArquivos[i].getName();
```

```
        processaArquivo(caminhosArquivos+nomeArquivos);
                    }
31.                 System.out.println("Processando
arquivo " + getAtual() + " de " + getTotal());
32.             }
33.         }
34.         // implementar aqui a regra de negocio
35.         public void processaArquivo(String
nomeArquivo)
36.          {
37.               Scanner in;
                  try {
                       in = new Scanner(new
38. FileReader(nomeArquivo));
                       System.out.println(in.nextLine());
39.               } catch (FileNotFoundException e) {
40.                    e.printStackTrace();
                  }
41.         }
42.         public static void main(String[] args) {


                  ProcessadorArquivos pa1 = new
ProcessadorArquivos("c:\\dados1\\");
                  pa1.inciar();


                  ProcessadorArquivos pa2 = new
ProcessadorArquivos("c:\\dados2\\");
                  pa2.inciar();
         }
}
```

Fonte: elaborado pelo autor.

Avançando na prática

Sistema de monitoramento climático

Descrição da situação-problema

Uma empresa possui diversos sensores espalhados pela cidade seguindo a ideia de *Internet of Things* (Internet das Coisas - IoT). Os sensores são inteligentes e conseguem enviar informações para um banco de dados. Dentre os que monitoram a temperatura, a velocidade do vento e a umidade merecem destaque, pois se algum deles enviar valores que correspondam a características de chuvas fortes, alertas devem ser enviados à defesa civil. Para monitorar todos esses elementos, é necessário um software que processe as informações de maneira rápida. Você trabalha na equipe de desenvolvimento desse produto e precisa criar um sistema com threads para fazer essa leitura. Para isso, produza um código que receba como parâmetro qual sensor fez a leitura, sendo identificado pelo seu tipo (1 - temperatura, 2 - velocidade do vento e 3 - umidade) e que possua linhas de execução independentes. Crie uma classe que utilize threads seguindo os passos abaixo:

1. Criar uma classe que implementa a interface `Runnable` e o método `run()`.
2. Criar uma instância tipo Thread e indicar a classe que implementa a interface `Runnable`.
3. Iniciar a thread com o método `start()`.
4. O construtor da classe deve receber o parâmetro que representa cada sensor.
5. A thread deve executar um método para fazer a leitura dos dados.

Resolução da situação-problema

O Quadro 2.7 possui uma classe que implementa um thread e que, em seu construtor, recebe por parâmetro qual tipo de sensor fará a leitura. Com isso, é possível criar diversas instâncias para monitorar o ambiente.

Quadro 2.7 | Implementação com threads para leitura de sensores ambientais

```
public class MonitorClima implements Runnable{
      private Thread th;
      private boolean monitorando;
      private int tipoSensor;
      public MonitorClima(int pTipoSensor)
      {
            tipoSensor = pTipoSensor;
            th = new Thread(this);
            monitorando = true;
            th.start();
      }
      public void run() {
            System.out.println("Iniciando monitoramento
climatico.");
            while(monitorando == true)
            {
            System.out.println("Lendo sensor" +
lerDadosSensor(tipoSensor));
            if(th.isInterrupted() == true)
            {
                  System.out.println("Parando
monitoramento.");
                  return;
            }
            try {
                  Thread.sleep(1000);
            } catch (InterruptedException e) {
                  monitorando = false;
            }
         }
      }
      private float lerDadosSensor(int tipo)
      {
         // código para ler dados do sensor
         return 0;
      }
      public static void main(String[] args) {
         MonitorClima monitorTemperatura = new
MonitorClima(1);
         MonitorClima monitorVelocidadeVento = new
MonitorClima(2);
         MonitorClima monitorUmidade = new
MonitorClima(3);
      }
}
```

Fonte: elaborado pelo autor.

Faça valer a pena

1. O desenvolvimento de aplicações paralelas requer um planejamento preciso, pois diversas tarefas ocorrerão ao mesmo tempo. Dessa forma, para seguir a correta implementação, é necessário executar três passos para se criar uma thread.

Quais são os 3 passos para se criar uma thread utilizando Java?

a) Implementar a interface `Runnable`, indicar essa implementação na instância da classe `Thread` e iniciar a thread com o método `start()`.

b) Implementar a interface `Clonnable`, indicar essa implementação na instância da classe `Thread` e iniciar a thread com o método `join()`.

c) Instanciar a classe `Thread`, pois o método de execução não depende de interfaces.

d) Implementar a interface `Runnable` e indicar essa implementação na instância da classe `Thread`, pois a thread tem o início automático.

e) Instanciar a classe `Runnable`, pois o método de execução não depende de interfaces.

2. As threads são iniciadas por um processo principal que já possui a sua própria linha de execução. Quando a thread termina a execução dos comandos do seu corpo, é possível estabelecer uma espera (sleep) para que esta seja finalizada.

Na *application programming interface* (API) do Java, qual comando garante que só terá retorno quando a thread terminar sua tarefa?

a) `sleep`.

b) `cancel`.

c) `start`.

d) `join`.

e) `kill`.

3. Uma thread consiste em uma linha de execução separada do programa principal. Com ela é possível criar uma ação que execute uma tarefa de forma recorrente, como o monitoramento de algum recurso. Para executar ação recorrente, é recomendado utilizar o comando `sleep` no final do método que a thread executa.

Assinale a alternativa que representa a correta descrição do comando `sleep`.

a) O comando `sleep` faz com que a thread pare de executar e apenas volte com o comando `wait`, com isso é possível controlar a thread.

b) O comando `sleep` faz com que a thread entre em estado de espera temporizada e permita que outros processos utilizem a memória.

c) O comando `sleep` faz com que a thread entre em estado de espera temporizada e permita que outros processos utilizem a CPU.

d) Sem o `sleep`, o método `interrupt` não funciona.

e) O comando `sleep` envia a thread para o estado executando.

Seção 2.2

Definição e tratamento de exceções para sistemas com threads

Diálogo aberto

Pense em um software que apresentou um erro e não foi possível nem ao menos explicar o problema. Você já deve ter reparado que sistemas computacionais de diversos tamanhos possuem uma característica em comum: defeitos que podem deflagrar uma falha que o usuário pode perceber, impedindo-o de executar uma tarefa. A diferença entre grandes e pequenos sistemas está relacionada à quantidade de linhas, classes, métodos e todos os elementos que compõem um grande sistema. Em pequenos sistemas, a menor quantidade de elementos propicia chances maiores de detecção desses defeitos e de produção de testes que possam descobrir tais problemas. Todavia, em sistemas de grande porte, devido à sua grande quantidade de componentes, a criação de testes que possam cobrir todo o sistema fica mais difícil e, ainda, quando uma falha é detectada pelo usuário, é mais complexo encontrar a linha de código. Independentemente do tamanho do sistema, se a própria linguagem de programação tiver recursos para indicar onde essa falha ocorreu, com mensagens que informem a razão, o processo de detecção e correção dessa falha se torna mais fácil, aumentando, assim, a capacidade de aplicar a manutenção. O sistema de tratamento de exceção em uma linguagem de programação pode ser comparado ao de monitoramento de movimento por câmeras: quando ocorre movimento em áreas ou horários fora do programado, o sistema gera um alerta.

Você trabalha em uma empresa que está com um projeto para atualizar o sistema de vestibular de uma universidade, agora está alocado na equipe que faz o *backend* das aplicações. O módulo que você desenvolveu para o processamento de dados de vestibular teve ótima avaliações, e a parte de relatório gerou uma informação útil para o planejamento da universidade. Todavia, depois de 30 dias de funcionamento, o sistema parou de funcionar e não

gerou nenhum tipo de log ou aviso para detectar o problema. O seu líder está preocupado com essa situação e quer garantir que, caso a falha ocorra novamente, seja possível descobrir a falha. Sua tarefa consiste em criar, para todos os métodos, o tratamento de exceção. Com isso, caso o sistema apresente algum problema, será possível verificar qual parte do software gerou um erro e qual erro exatamente foi gerado.

Nessa seção veremos as diversas formas de tratamento de exceção que a linguagem Java possui. Veremos como usar os blocos `try`, `catch` e `finally`, além de formas de personalizar o sistema de tratamento de exceção na sua aplicação. Vamos tratar os erros de uma aplicação e descobrir a origem deles de maneira muito mais rápida.

Não pode faltar

Os recursos para tratamento de exceções são muito importantes para produzir sistemas de qualidade e são mecanismos que a própria linguagem propicia para detectar o local em que um problema ocorreu. As formas de tratar esses erros geram blocos em que, caso ocorra alguma falha, a própria linguagem de programação informa quais linhas geraram o problema e qual problema foi gerado (DEITEL; DEITEL, 2016).

O processo de detecção de falhas em sistemas que possuem apenas uma linha de execução já é trabalhoso e se torna ainda mais complexo em códigos que possuem diversas linhas de execução (threads). Com isso, utilizar as diversas formas de tratamento de exceção que o Java possui é importante para detectar o problema. Imagine que o sistema em que você está trabalhando será apresentado em 15 minutos e, por um motivo desconhecido, ele para de funcionar. Com o sistema de tratamento de exceção é possível descobrir em qual linha o problema foi gerado e qual a mensagem de erro relacionada ao problema (FURGERI, 2015).

Assimile

O uso dos blocos **try** e **catch** é muito útil para detectar e tratar algum erro no código. Com eles o sistema não será encerrado caso ocorra alguma falha e pode tentar se recuperar (MANZANO, 2014).

Em passos anteriores já nos deparamos com esse tratamento de exceção, por exemplo, no Quadro 2.8, que apresenta um código simples que lê números do teclado. A linha 1 importa a classe `Scanner`, que faz a leitura de diversos fluxos e que, no caso, estamos utilizando para ler o teclado (`System.in`) na linha 8, e na linha 9 é utilizado o comando `int a = in.nextInt();` para ler números inteiros. Repare que o bloco **try** está entre as linha 7 e 12 e, em sequência, temos o bloco **catch**. O **try** determina que o código enclausurado pelas chaves será redirecionado para o bloco **catch**, caso alguma linha gere alguma exceção. O comando `e.printStackTrace();` imprime na saída padrão do sistema a sequência que gerou a exceção (no caso, o console, mas essas mensagens devem ser direcionadas para um arquivo de texto central ou um banco de dados).

Quadro 2.8 | Código com tratamento de exceção para leitura de número de teclado

```
1.  import java.util.Scanner;
2.
3.  public class LeitorTeclado {
4.
5.       public void lerDados()
6.       {
7.            try {
8.                 Scanner in = new Scanner(System.
    in);
9.                 int numero = in.nextInt();
10.                System.out.println(numero);
11.                in.close();
12.           }catch(Exception e)
13            {
14.                e.printStackTrace();
```

```
15.             }
16.         }
17.
18.         public static void main(String[] args) {
19.             LeitorTeclado l = new LeitorTeclado();
20.             l.lerDados();
21.         }
22. }
```

Fonte: elaborado pelo autor.

O código do Quadro 2.8 lê números do terminal e os imprime na tela. Todavia, caso o usuário insira um dado que não seja um número, o sistema gerará uma exceção. O Quadro 2.9 apresenta o resultado de uma execução de quando o usuário insere um texto, no caso, a palavra `teste`, e o sistema não está preparado para esse tipo de entrada. O erro gerado está especificado na linha 2 `java.util.InputMismatchException` que, traduzindo, significa Entrada incompatível. Esse erro é gerado pois o sistema espera um número inteiro e o usuário inseriu um texto. A linha 8, do Quadro 2.9, mostra o local que originou o erro, ou seja, na linha 20 do método `main` da classe `LeitorTeclado` (Quadro 2.8). A linha 7 (Quadro 2.8) indica o que causou o erro, ou seja, a linha 9, do método `lerDados` da classe `LeitorTeclado` (Quadro 2.9). Se você voltar ao código do Quadro 2.8, a linha 9 é exatamente onde a entrada do teclado é lida e atribuída à variável inteira `numero`. As demais linhas do Quadro 2.9 indicam os locais na classe `Scanner` que geram o erro. Nesse caso, o erro foi realmente gerado por uma entrada errada do usuário. Todavia, como é possível ver o código-fonte do Java, existe a chance de analisar o erro e verificar se o SDK está com defeitos (na instalação do JDK é possível instalar o código-fonte).

Quadro 2.9 | Sequência de linhas mostrando a origem do erro

```
1. Teste
2. java.util.InputMismatchException
3.       at java.util.Scanner.throwFor(Scanner.
   java:864)
4.       at java.util.Scanner.next(Scanner.java:1485)
```

```
5.      at java.util.Scanner.nextInt(Scanner.
java:2117)
6.      at java.util.Scanner.nextInt(Scanner.
java:2076)
7.      at LeitorTeclado.lerDados(LeitorTeclado.
java:9)
8.      at LeitorTeclado.main(LeitorTeclado.java:20)
```

Fonte: elaborado pelo autor.

Reflita

O uso do tratamento de exceção é o mecanismo indicado para avaliar se bloco de código foi executado corretamente. Porém, como isso afeta a robustez, coesão e acoplamento de classes e métodos sob o foco de um sistema complexo?

No Quadro 2.10 são descritos alguns tipos de exceção comuns que podem ocorrer em diversos cenários.

Quadro 2.10 | Exceções comuns em códigos Java

Exceção	Descrição
`java.lang.ArrayIndexOutOfBoundsException`	Quando se tenta acessar uma posição de vetor ou matriz que não existe.
`java.lang.ArithmeticException`	Gerado na divisão de um número **`int`** por zero.
`java.lang.IllegalArgumentException`	Enviado quando se passa argumentos errados para um método.
`java.io.FileNotFoundException`	Quando se tenta fazer a leitura ou escrita em um arquivos que não existe.

Fonte: elaborado pelo autor.

Esse tipo de tratamento deve ser usado na maioria dos pontos de uma aplicação, porém, devemos nos atentar à execução das linhas. Quando ocorrer uma exceção, utilizaremos como exemplo uma implementação do método `escreveArquivo()` no Quadro

2.11. Se uma falha for lançada na linha 11, as linhas 12 e 13 não serão executadas e a sequência irá para o bloco **catch**. Com isso, o recurso nunca será "fechado", pois a instrução para fechar está na linha 13 e esta só será executada em uma sequência sem erros. Para evitar esse problema, é possível utilizar o bloco **finally**, que sempre executará, não importando se o método gerou erro ou se executou um **return**. O **finally** não executará somente se a aplicação for terminada (HORSTMANN, 2016). Assim, é possível fazer o tratamento final de qualquer evento, como nas linhas de 16 a 23. Veja que, mesmo dentro do **finally**, é necessário utilizar um **try** e um **catch** por imposição do método. Porém, isso garante que sempre se tentará executar o procedimento para finalizar ou tratar algo sem importar se houveram erros a serem tratados (WINDER, 2009).

Quadro 2.11 | Método **escreveArquivo()** com **try**, **catch** e **finally**

```
1.  import java.io.*;
2.
3.  public class ControleArquivos
4.  {
5.
6.      public void escreveArquivo(String
    caminhoArquivo)
7.      {
8.          FileWriter fw = null;
9.          File f = new File(caminhoArquivo);
10.         try {
11.             fw = new FileWriter(f);
12.             fw.write(10);
13.             fw.close();
14.         }catch (IOException e) {
15.             e.printStackTrace();
16.         }finally {
17.            if(fw != null)
18.                try {
19.                    fw.close();
20.                } catch (IOException e) {
21.                        e.printStackTrace();
```

```
22.                         }
23.                     }
24.                 }
25.         public static void main(String[] args) {
26.                 ControleArquivos l = new
ControleArquivos();
27                  l.escreveArquivo("c:\\arquivoTeste.txt");
28          }
29  }
```

Fonte: elaborado pelo autor.

Com essa maneira de tratar os erros, acabamos por determinar uma regra:

1. No bloco **try** se executa as tarefas pertinentes à abertura de recursos necessários (arquivos, conexões de rede e outros).
2. No **catch** se avalia o que ocorreu e se cria logs para informar qual é o problema;
3. No **finally** se fecha os recursos utilizados, em alguns casos sendo necessários outros blocos para garantir que todos os recursos sejam utilizados.

Com base nessas regras, os próprios desenvolvedores do Java criaram um mecanismo de tratamento para condicioná-las em um próprio comando da linguagem, a partir do Java 9.

Exemplificando

Veja essa nova versão do **try** na linha 4 do Quadro 2.12. Dentro do bloco é possível declarar qual recurso deverá ser fechado, e a própria *Java Virtual Machine* (JVM) o fecha. Para que a classe possa ser fechada, deve-se implementar a interface Closeable. Esse tipo de recurso ajuda a retirar preocupações do programador e deixa o processo menos propenso a erros. Todavia, isso gera mais trabalho para a JVM, causando mais processamento.

Quadro 2.12 | Novo tratamento de exceção para recursos

```
1.  public void escreveArquivo()
2.  {
3.           f = new File(caminhoArquivo);
4.           try(FileWriter fw = new
5.  FileWriter(f);)
6.            {
7.                  fw.write(10);
8.            }catch (IOException e) {
9.                  e.printStackTrace();
10.           }
    }
```

Fonte: elaborado pelo autor.

Existem diversas classes que implementam a interface Closeable e que podem aproveitar essa nova abordagem de tratamento de exceção, tais como:

- ServerSocket: classe que controla os sockets da camada *transmission control protocol* (TCP).
- DatagramSocket: classe que controla o envio de pacote pelo protocolo *user datagram protocol* (UDP).
- Connection: controla as conexões com os sistemas de gerenciamento de banco de dados.

Pesquise mais

No desenvolvimento de sistemas, é importante que todos os erros sejam enviados para um arquivo de texto ou um banco de dados, a fim de criar relatórios e, para isso, existem ferramentas que ajudam, como o Log4J. Conheça mais sobre ela acessando o link: <https://logging.apache.org/log4j/2.x/>. Acesso em: 17 jul. 2018.

Tratamento personalizado de exceções

Existem outras formas de fazer o tratamento de exceção relacionado à centralização e ao controle do local onde os tratamentos de erros são implementados.

Alguns métodos, quando utilizados, obrigam a implementação

do tratamento de erros. Isso ocorre porque esses métodos utilizam a cláusula **throws**, recurso que faz com que o método "lance" uma exceção ou mais, que serão especificadas pelo programador, caso ocorra algum problema.

No exemplo do Quadro 2.13 é apresentada uma classe que faz a leitura de um arquivo. Nesse tipo de situação, podem ocorrer erros ao ler o arquivo, por exemplo, ele pode não ser encontrado. Para evitar que o programa pare sua execução, na linha 12, o método `escreveArquivo()` foi escrito de modo a lançar uma exceção do tipo `IOExeption`, ou seja, ele invocará a classe = usada para esse tipo específico de erro. A classe só será invocada caso na linha 23 ocorra um erro, que será capturado na linha 24, especificamente pela classe usada no `throws`. Implementar dessa forma permite centralizar o tratamento das exceções, além de personalizar cada ação.

Quadro 2.13 | Utilização do **throws**

```
1.  import java.io.File;
2.  import java.io.FileWriter;
3.  import java.io.IOException;

4.  public class EscritorArquivo {

5.      private String caminhoArquivo;
6.      private FileWriter fw;
7.      private File f;

8.      public EscritorArquivo(String
    pCaminhoArquivo)
9.      {
10.         caminhoArquivo = pCaminhoArquivo;
11.     }

12.     public void escreveArquivo() throws
    IOException
13.     {
14.         f = new File(caminhoArquivo);
15.         fw = new FileWriter(f);
```

```
16.             fw.write(10);

17.             if(fw != null)
18.                 fw.close();
19.         }

20.         public static void main(String[] args) {
21.             EscritorArquivo s = new
EscritorArquivo("dados1/arquivo.txt");
22.             try {
23.                 s.escreveArquivo();
24.             } catch (IOException e) {
25.                 e.printStackTrace();
26.             }
27.         }
28. }
```

Fonte: elaborado pelo autor.

Outro ponto interessante sobre o uso da cláusula `throws` é a possibilidade de o programador criar suas próprias exceções. Isso é feito por meio do comando `throw new Exception()`. Veja que, no código do Quadro 2.14, linha 9, o método `lerArquivo()` lança uma exceção da classe `Exception`. Caso ocorra um erro na leitura do arquivo, será exibida a mensagem Arquivo não encontrado na linha 22. Isso acontece porque o erro será tratado na linha 14, pelo comando **`throw new Exception`**`("Arquivo nao encontrado");`.

Quadro 2.14 | Controle de erros por envio de exceção

```
1.  import java.io.File;
2.  import java.util.Scanner;

3.  public class LeitorArquivo {

4.      private String caminhoArquivo;

5.      public LeitorArquivo(String pCaminhoArquivo)
```

```
6.        {
7.             caminhoArquivo = pCaminhoArquivo;
8.        }
9.        public void lerArquivo() throws
10.  java.lang.Exception
11.       {
12.            File f = new File(caminhoArquivo);
13.            if(f.exists() == false)
14.            {
15.                 throw new Exception("Arquivo nao
16.  encontrado");
17.            }
18.            Scanner sc = new Scanner(f);
               String dados = sc.next();
19.            System.out.println(dados);
          }
20.       public static void main(String[] args) {


21.            LeitorArquivo le = new
22.  LeitorArquivo("dados1/arquivo.txt");
23.            try {
24.                 le.lerArquivo();
25.            } catch (Exception e) {
26.                 e.printStackTrace();
               }
          }
     }
```

Em sistemas paralelos temos exceções que devem ser tratadas devido à exceção de diversas threads. Veja no Quadro 2.15 algumas exceções que poderão ocorrer caso uma thread seja interrompida ou receba parâmetros incorretos.

Quadro 2.15 | Exceções geradas por um sistema paralelo

Exceções geradas por threads	Descrição
`IllegalArgumentException`	Quando é feito um `Thread.sleep` (valor), o valor deve estar entre 0 e 999999.
`InterruptedException`	Quando uma thread é interrompida. Esse processo pode ocorrer quando se tenta parar a thread.
`SecurityException`	É possível criar grupos de threads. Essa exceção é gerada quando a thread não pode ser inserida em um certo grupo.

Portanto, o sistema de tratamento de erros é essencial para garantir o funcionamento caso algum evento ocorra de forma incorreta, além de assegurar que o sistema terá maiores chances de manutenção, evitando problemas maiores em caso de falhas ou problemas.

Pesquise mais

Existem diversas fontes para avançar nos conhecimentos sobre tratamento de exceção em Java:

- PANDEY, A. **Try-With-Resource Enhancements in Java 9**. DZone, 10 abr. 2017. Disponível em: <https://dzone.com/articles/try-with-resources-enhancement-in-java-9>. Acesso em: 17 jul. 2018.
- ORACLE. **The try-with-resources statement**. [S.l.], [s.d.]. Disponível em: <https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>. Acesso em: 17 jul. 2018.
- GHOSH, S. **Method for optimizing java bytecodes in the presence of try-catch blocks**. 29 jan. 1999. Disponível em: <https://patents.google.com/patent/US6412109B1/en>. Acesso em: 17 jul. 2018.
- APACHE LOGGING SERVICES. **Apache Log4J™**. [S.l.], [s.d.]. Disponível em: <https://logging.apache.org/log4j/2.x/>. Acesso em: 17 jul. 2018.
- LOIANE GRONER. **Curso de Java 47**: Exceptions: try, catch. [S.l.], 2 mar. 2016. Disponível em: <https://youtu.be/ld2C4GcAtsg>. Acesso em: 17 jul. 2018.

Sem medo de errar

Você trabalha em uma empresa especializada em migração de sistemas legados, que está trabalhando em um sistema de uma universidade. A primeira etapa do projeto era a criação de uma nova interface gráfica que foi muito bem aceita e que o levou para a equipe de *backend*. Como uma nova tarefa nessa equipe, você desenvolveu o sistema para processamento dos dados de vestibular, mas depois de 30 dias o sistema parou de funcionar e o seu líder está muito preocupado. Para solucionar a questão, você deve aplicar o tratamento de exceção para que, quando houver uma falha, seja possível descobrir em qual parte o problema ocorreu.

Para resolver esse problema, deve-se utilizar o sistema de tratamento de exceção nos seguintes casos:

- Todos os métodos ou construtores devem ter os blocos **try** e **catch** com a função que envia para a saída padrão a sequência de erros gerados:

```
try {
      //conteúdo do método ou construtor
} catch (Exception e) {
      e.printStackTrace();
}
```

- Caso seja utilizado algum recurso como arquivos, conexões de rede ou outros, o método **finally** deve fechar esse recurso:

```
Scanner in = null;
try{
      try{
            //conteúdo do método ou construtor
      } catch (FileNotFoundException e) {
      e.printStackTrace();
      }
}catch (Exception e) {
      e.printStackTrace();
```

```
}finally {
    if(in != null)
        in.close();
}
```

- Ou ainda, pode-se utilizar a nova função do **try** que fecha esse recurso:

```
try {
    try( Scanner in = new Scanner(new
FileReader(nomeArquivo));) {
            System.out.println(in.next());
    } catch (FileNotFoundException e) {
            e.printStackTrace();
           }
}catch (Exception e) {
    e.printStackTrace();
}
```

Com as diversas formas de tratamento de exceção apresentadas, agora é possível tratar todas as exceções de um programa. Com isso, é possível detectar qual linha do código foi responsável pelo erro e fazer as correções de forma mais rápida, Além de conseguir detectar a falha e fazer com que o sistema se recupere desse problema. Aproveite a oportunidade para se destacar e implemente também tratamentos personalizados com a cláusula `throws`.

Avançando na prática

Tratamento de erros para conexões de rede

Descrição da situação-problema

Uma empresa de automação está desenvolvendo um sistema para monitoramento residencial, e o primeiro elemento a ser desenvolvido é o sensor de temperatura. Esse sensor envia os dados via rede Wi-Fi utilizando o protocolo *user datagram protocol* (UDP). Todavia, é necessário aplicar um sistema de tratamento de erros para garantir que o leitor sempre esteja disponível para enviar os

dados. Você foi designado para implementar essa solução e foi lhe passado o código do Quadro 2.16, que possui a primeira versão sem o tratamento de exceção. Dessa forma, é necessário colocar as instruções de **try** e **catch** para cada linha dos métodos em `LeitorTemperatura()` e `recebeDados()`.

Quadro 2.16 | Sistema de recepção de dados via UDP

```
import java.net.*;

public class LeitorTemperatura {

     private DatagramSocket clientSocket;
     private DatagramPacket pacoteReceber;

     public LeitorTemperatura()
     {
          ip = pIp;
          clientSocket = new DatagramSocket(8000);
     }

     public void recebeDados()
     {
          byte[] dados = new byte[1024];
          pacoteReceber = new DatagramPacket(dados,
dados.length);
          clientSocket.receive(pacoteReceber);
          System.out.println("Temperatura " + new
String(pacoteReceber.getData()));
          clientSocket.close();
     }
}
```

Fonte: elaborado pelo autor.

Resolução da situação-problema

O Quadro 2.17 apresenta o sistema com o tratamento. Observe que receberam tratamento de exceção os métodos `LeitorTemperatura()` e `recebeDados()`. Dessa forma, caso ocorra algum erro, é possível detectar.

Quadro 2.17 | Sistema de recepção de dados com tratamento de exceção

```
package U2S2;

import java.io.IOException;
import java.net.*;

public class LeitorTemperatura {

      private DatagramSocket clientSocket;
      private DatagramPacket pacoteReceber;
      private String ip;

      public LeitorTemperatura()
      {
        try {
               ip = pIp;
               try {
                           // Cria um socket UDP para
receber pacotes na porta 8000
                     clientSocket = new
DatagramSocket(8000);
               } catch (SocketException e) {
                     e.printStackTrace();
               }
        }catch(Exception e)
        {
               e.printStackTrace();
        }
      }

      public void recebeDados()
      {

        try {
               byte[] dados = new byte[1024];
               // Cria um pacote para receber os dados
do sensor
               pacoteReceber = new DatagramPacket(dados,
dados.length);
               try {
                           // Usa a conexão para
receber os dados
               clientSocket.receive(pacoteReceber);
```

```
} catch (IOException e) {
                    e.printStackTrace();
                }
                System.out.println("Temperatura " + new
String(pacoteReceber.getData()));
                // Fecha o socket
                clientSocket.close();
        }catch (Exception e) {
                e.printStackTrace();
        }
      }
}
```

Fonte: elaborado pelo autor.

Faça valer a pena

1. O tratamento de exceção é responsável pelo controle de fluxos alternativos que um software pode ter. Certos eventos podem ser considerados como errados e devem ser tratados para que o caminho principal do sistema seja executado. Para isso, a linguagem Java utiliza algumas palavras reservadas para fazer esse controle.

Quais opções representam palavras reservadas do tratamento de exceção do Java?

a) `try`, `catch` e `error`.

b) `try`, `catch` e `finally`.

c) `throw`, `throws` e `bit`.

d) `int`, `main` e `go`.

e) `try`, `catch` e `throwt`.

2. O Java possui algumas formas para o tratamento de uma exceção. Esses métodos ou construtores são modelados para enviar uma exceção que pode ocorrer, como uma conexão via rede que não está disponível ou ainda valores fora do intervalo permitido.

Qual das palavras reservadas do Java força o tratamento de exceção em método ou construtor?

a) `throw`.

b) `throws`.

c) `try`.

d) `catch`.

e) `error`.

3. Para executar o tratamento de exceção em Java, o bloco `try` faz a execução regular das ações, o bloco `catch` faz o tratamento caso ocorra um erro, enviando mensagens ou arquivando esses logs em arquivos ou banco de dados para serem consultados. O último bloco, `finally`, tem um papel importante para garantir a robustez do sistema.

Dada as informações abaixo:

I. O bloco `finally` é indicado para fechar os recursos utilizados no bloco `try`.

II. As versões mais novas do Java incorporam a finalização de recursos no bloco `try`.

III. O bloco `catch` é sempre executado.

IV. O bloco `finally` é sempre executado.

V. O `try` consiste no tratamento do caminho relacionado ao erro da aplicação.

Escolha a opção que apresente apenas as afirmações corretas:

a) Apenas as afirmações I e II estão corretas.

b) Apenas as afirmações III e V estão corretas.

c) Apenas as afirmações I, II e V estão corretas.

d) Apenas as afirmações I, II, III e V estão corretas.

e) Todas as afirmações estão corretas.

Seção 2.3

Programação em Java utilizando elementos para sincronização em Java

Diálogo aberto

Os recursos computacionais podem ser comparados a produtos de supermercados: existem diversos itens a serem comprados. Todavia, se dois clientes diferentes forem comprar o mesmo item, ao mesmo tempo, será necessário algum tipo de controle para evitar conflitos. As threads em um sistema computacional podem ser comparadas aos clientes, e os itens do mercado são relacionados à memória, aos dispositivos de entrada e de saída ou a outros recursos do computador. Como forma de utilizar o hardware de maneira mais eficiente ou fazer com que uma aplicação seja capaz de executar diversas tarefas ao mesmo tempo, utilizamos as threads. Com elas, é possível criar linhas de execução independentes no mesmo processo. Durante a programação paralela, é necessário garantir que certos recursos sejam utilizados por apenas uma thread. Já imaginou se duas threads tentarem escrever ao mesmo tempo em um arquivo de texto ou uma conexão com o banco de dados? Para isso, é necessário usar mecanismos da linguagem que impeçam a utilização de recursos simultâneos.

Você trabalha em uma empresa especializada em migração de sistemas legados e está efetuando um projeto de migração de um sistema universitário. Como membro da equipe de desenvolvimento, está fazendo o processamento dos dados do vestibular e precisa garantir a obtenção, ao final do processo, de um relatório consolidado do processamento que será gerado no software e depois inserido no banco. Como os dados serão consolidados em apenas uma classe, é necessário que os métodos sejam sincronizados para evitar problemas. O objetivo é que durante o processamento dos dados o sistema faça uma estatística da quantidade de alunos de cada curso e acumule em apenas uma classe. Para isso, produza uma classe que guarde a quantidade de alunos de cada curso que o sistema está processando. Ela deve conter métodos sincronizados

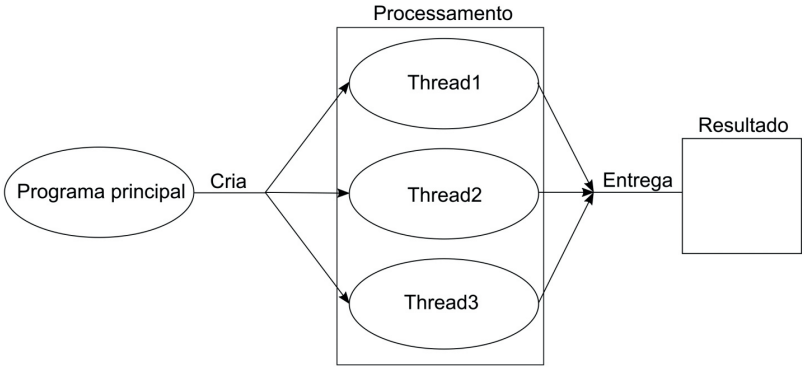
para incrementar as quantidades de cada curso e, assim, possibilitar que a thread que está fazendo o processamento informe os valores. Para resolver esse problema será apresentado o uso das classes *Timer* e *TimerTask* para encapsular a complexidade das threads e as formas de controle de acesso de métodos por diversas threads (sincronismo). Utilize sua visão e sua capacidade de abstração, agora você tem de controlar diversas tarefas ao mesmo tempo.

Mãos à obra!

Não pode faltar

Existem diversas formas de tratar as exceções em um código orientado a objetos e paralelo, mas há problemas que podem ocorrer em sistemas concorrentes que não geram erros que são tratados pelos mecanismos do próprio Java. A utilização de threads em um código é essencial para que um software execute mais de uma tarefa ao mesmo tempo e, com isso, propicie mais interação com o usuário e use o hardware de maneira mais completa, entre outros cenários (MANZANO, 2014). Todavia, o uso da programação paralela pode gerar problemas decorrentes da execução de duas ou mais tarefas simultâneas. Para ilustrar esses cenários, podemos pensar em uma classe básica em que quantidades de elementos são somadas ou subtraídas. Nessa situação, imagine que diversas threads utilizam a mesma instância dessa classe para centralizar essas estatísticas. Esse tipo de cenário em programação é bem comum, e a Figura 2.3 apresenta um caso em que um programa principal cria três threads para fazer um processamento e, ao final, todas elas escrevem seus dados em apenas um elemento central.

Figura 2.3 | Sistema de processamento utilizando processamento paralelo



Fonte: elaborada pelo autor.

O Quadro 2.18 apresenta um exemplo de classe para centralizar dados vindos de diversas threads. `Contador` representa o Resultado na Figura 2.3. Como é possível observar, a classe não possui nenhum recurso especial em termos de programação, mas em termos de funcionamento sim, pois como os dados devem ser centralizados usando-se várias threads, todas elas utilizam a mesma instância da classe `Contador`.

Quadro 2.18 | Código para controle da quantidade de alunos utilizado por diversas threads

```
package U2S3;

public class Contador {
      private int quantidadeAlunosCurso;

      public void incrementa()
      {
        quantidadeAlunosCurso++;
      }

      public void decrementa()
      {
        quantidadeAlunosCurso--;
      }
}
```

Fonte: elaborado pelo autor.

No método `incrementa()`, o valor do `quantidadeAlunosCurso` é incrementado com operador ++, dessa forma, a operação consiste em:

- Recuperar o valor de `quantidadeAlunosCurso`.
- Incrementar o valor de `quantidadeAlunosCurso` em 1.
- Armazenar o valor alterado de `quantidadeAlunosCurso`.

O método `decrementa()` segue a mesma regra, apenas fazendo um decremento no valor `quantidadeAlunosCurso`. Em um cenário de apenas uma linha de execução não há problemas, porém, ao pensar que duas ou mais threads utilizarão a mesma referência dessa classe, podemos ter um problema de interferência de threads (DEITEL; DEITEL, 2016). Veja o que pode ocorrer quando duas threads fazem o acesso a um mesmo método na mesma referência:

- Thread 1: chama o `incrementa()`.
- Thread 2: chama o `decrementa()`.
- Thread 1: recupera o valor de `quantidadeAlunosCurso`.
- Thread 2: recupera o valor de `quantidadeAlunosCurso`.
- Thread 1: incrementa o valor de `quantidadeAlunosCurso` em 1, agora `quantidadeAlunosCurso` é 1.
- Thread 2: decrementa o valor de `quantidadeAlunosCurso` em 1, agora `quantidadeAlunosCurso` é -1.
- Thread1:armazenaovaloralteradode`quantidadeAlunosCurso`, agora `quantidadeAlunosCurso` é 1.
- Thread 2: armazena o valor alterado de `quantidadeAlunosCurso`, agora `quantidadeAlunosCurso` é -1.

Repare que o processamento que a thread 1 realizou foi sobrescrito e o valor do `quantidadeAlunosCurso` 1 está errado, pois as duas threads fizeram o acesso ao mesmo tempo. Dessa forma, são necessárias estratégias para que a própria linguagem de programação e o sistema operacional garantam que certos métodos sejam acessados por apenas uma thread por vez. Para isso, existe a palavra reservada **`synchronized`**, utilizada no método que vai garantir que cada thread acesse de forma única, a fim de evitar inconsistência dos dados (HORSTMANN, 2016). Assim, a *Java Virtual Machine* (JVM), que faz a execução dos códigos junto ao

sistema operacional, consegue garantir que cada thread acesse os métodos de forma individual (FURGERI, 2015).

Reflita

Grandes sistemas de processamentos como os de declaração de imposto de renda ou os controles de cartão de crédito necessitam de muitos computadores, processos e threads para garantir desempenho e disponibilidade. Quando utilizamos os controles para garantir o acesso de uma thread por vez em um método, estamos afetando o desempenho de forma positiva ou negativa?

No Quadro 2.19 temos um exemplo do uso do mecanismo de sincronização. Nas linhas 2 e 3 foram utilizados métodos synchronized, com isso, quando diversas threads acessarem esses métodos, **a própria JVM, junto ao sistema operacional, consegue garantir que o acesso** seja individual.

Quadro 2.19 | Quadro de contagem com métodos sincronizados

```
    package U2S3;

    public class ContadorSync {
1.      private int quantidadeAlunosCurso;

2.      public synchronized void incrementa()
        {
          quantidadeAlunosCurso++;
        }

3.      public synchronized void decrementa()
        {
          quantidadeAlunosCurso--;
        }
}
```

Fonte: elaborado pelo autor.

Com os métodos **synchronized**, é possível produzir códigos paralelos confiáveis. Alguns cenários são propícios para o uso de threads, por exemplo, monitoramento dos recursos de rede, que acessam arquivos no disco de forma periódica, entre outras ações. Para a produção de uma thread que faça ações recorrentes, é necessário encapsular a ação dentro do método `run()`, como no Quadro 2.20.

Assimile

Métodos `synchronized` são utilizados para garantir que cada thread terá acesso exclusivo ao método. Com isso, é possível evitar problemas de concorrência ou interferência entre threads.

No código do Quadro 2.20, na linha 1 é implementada a interface `Runnable` para que seja possível criar threads e utilizar o método `run()` da linha 9 (lembrando que ele é obrigatório, pois é por onde a thread começará a ser executada).

O item `pausa` na linha 5 (dentro do construtor da classe) define o intervalo que a ação da thread será executada que, nesse caso, foi definido como `pausa = 1000;`. Quando a thread é instanciada, `th = new Thread(this)`, e iniciada, `th.start()`, o método `run()` entra em ação. Nesse método, é necessário inserir o comando: `Thread.sleep(pausa)` logo no início e depois do processamento, para garantir que a thread não ocupe os núcleos de processamento, sem permitir que outros processos (como do sistema operacional) sejam executados. Usando essas *pausas*, é possível garantir que todas as threads façam seu processamento e, ao implementar os métodos com a diretiva **synchronized**, os resultados serão convergidos ao final. Porém, para a criação de threads ainda temos de pensar nas questões da orientação a objetos, nos aspectos da coesão e do acoplamento. No código do Quadro 2.20, a classe que controla a thread também faz o processamento e também é necessário fazer o controle manual do intervalo de processamento da thread utilizando o `Thread.sleep(pausa);`. Seguindo a ideia da orientação a objetos, vamos procurar uma forma mais adequada para fazer essas tarefas?

Quadro 2.20 | Implementação de thread para execução de ações periódicas

```
      package U2S3;

 1.   public class RepetidorThread implements Runnable{

 2.        private int pausa;
 3.        private boolean executa;
 4.        private Thread th;

           public RepetidorThread()
           {
 5.             pausa = 1000;
 6.             executa = true;
 7.             th = new Thread(this);
 8.             th.start();
           }

 9.        public void run() {
                try {
10.                  Thread.sleep(pausa);
                } catch (InterruptedException e) {
                     e.printStackTrace();
                }
11.             while(executa)
                {
                     // faz o processamento
                     try {
12.                       Thread.sleep(pausa);
                     } catch (InterruptedException e) {
                          e.printStackTrace();
                     }
                }
           }
      }
```

Fonte: elaborado pelo autor.

Para evitar os problemas das questões de modelagem e ainda aumentar o acoplamento das classes que utilizam threads, é possível utilizar as classes *Timer* e *TimerTak* (HORSTMANN, 2016). A *Timer* é responsável por controlar a thread para que seja executada de forma periódica com um intervalo definido pelo usuário. A *TimerTaks* é abstrata e implementa a interface `Runnable`, ou seja, é necessário fazer uma especialização dela e implementar o método `run()`. Para que o sistema funcione, é preciso implementar as duas classes, pois na classe *Timer* o método que inicia o processamento espera uma instância da classe *TimerTask*. Com isso, a parte que cuida do

controle da thread (*Timer*) fica separada da que faz o processamento (*TimerTask*). Dessa maneira, a coesão das classes aumenta, criando formas de manutenção mais amplas.

Exemplificando

Os sistemas *customer relationship management* (CRM) são responsáveis pelo controle da relação para envio de propagandas, brindes e outros elementos entre uma empresa e seus clientes. Esses sistemas possuem milhares de linhas de código. O uso do *Timer* e do *TimerTask* representa um aumento da coesão das classes e encapsulamento, assim, as manutenções ficam facilitadas.

Veja como utilizar as classes *Timer* e *TimerTask* no Quadro 2.21. A linha 1 `import java.util.Timer;` declara qual *timer* utilizar (não se deve confundir com o `javax.swing.Timer`), e a linha 2 declara o objeto `timer` do tipo *Timer*. Na linha 3 é declarado um objeto da classe `RepetidorTimeTarefa`. Essa classe é a especialização da `TimerTask`, que veremos logo a seguir. Na linha 4 foi criada uma variável chamada `pausa`, e esse valor será passado na classe *Timer* para controlar o tempo das threads. No construtor da classe `public RepetidorTimer()`, instanciamos a classe *Timer* e a `tarefa` do tipo `RepetidorTimeTarefa` e também definimos o valor da variável `pausa = 1000` (em milissegundos). Por fim, configuramos a tarefa que o *Timer* fará. Em `timer.schedule(tarefa, 0, pausa);`, o parâmetro `tarefa` (uma especialização da classe *TimerTaks*) define qual ação esse *timer* fará, o valor 0 define o atraso para iniciar a tarefa e o valor `pausa` define o intervalo em que a thread será executada.

Quadro 2.21 | Utilização da classe *Timer*

```
     package U2S3;
1.   import java.util.Timer;


     public class RepetidorTimer{


2.        private Timer timer;
3.        private RepetidorTimeTarefa tarefa;
4.        private int pausa;
```

```
    public RepetidorTimer()
          {
5.          timer = new Timer();
6.          pausa = 1000;
7.          tarefa = new
8.  RepetidorTimeTarefa("arquivoDados.txt");
            timer.schedule(tarefa, 0, pausa);
          }
          public static void main(String[] args) {
            RepetidorTimer rt = new RepetidorTimer();
          }
}
```

Fonte: elaborado pelo autor.

A classe do Quadro 2.22 representa o **private** `RepetidorTimeTarefa tarefa;` do Quadro 2.21 e tem o papel de especificar e executar a tarefa da thread. Vamos aproveitar e ver uma aplicação de um *TimerTask*? O Quadro 2.22 apresenta o uso da classe *TimerTask*. Veja que na linha 1 a classe `RepetidorTimeTarefa` faz uma especialização da *TimerTask* (**extends**). Com essa implementação, a classe `RepetidorTimeTarefa` é uma especialização da classe *TimerTask*, e a classe *TimerTask* implementa a interface *Runnable*. Dessa forma, a classe `RepetidorTimeTarefa` deve criar o método `run()`, como foi feito na linha 3. O método `run()` faz a busca de um arquivo na linha 5 e, na linha 6, ele verifica se o arquivo existe. Caso verdadeiro, nas linhas 7, 8 e 9 é feito o acesso ao arquivo e a leitura de seus dados, e na linha 10 é feita a impressão das linhas de texto do arquivo.

Quadro 2.22 | Exemplo de utilização da *TimerTask*

```
    package U2S3;

    import java.io.*;
    import java.util.TimerTask;


1.  public class RepetidorTimeTarefa extends
    TimerTask {


2.     private String arquivo;
```

```
   public RepetidorTimeTarefa(String parquivo)
   {
      this.arquivo = parquivo;
   }
3. public void run() {
      System.out.println("Buscando arquivo.");
      try {
5.       File f = new File(arquivo);
         System.out.println("Arquivo
6. encontrado.");
         if(f.exists() == true)
         {
            String line = null;
7.          FileReader fileReader = new
   FileReader(f);
8.          BufferedReader bufferedReader =
9.                new BufferedReader(fileReader);
10.         while((line = bufferedReader.
   readLine())
                                       != null) {
               System.out.println(line);
            }
         }
      }catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Fonte: elaborado pelo autor.

O Quadro 2.23 apresenta o resultado do processamento feito de 1 em 1 segundo. Repare que a classe que faz o processamento `RepetidorTimeTarefa` não necessariamente precisa ser utilizada dentro de cenários de uma thread. Com isso, o acoplamento entre as classes fica baixo, portanto, aumenta as chances de reaproveitamento de código.

Quadro 2.23 | Resultado do processamento

```
Buscando arquivo.
Buscando arquivo.
Buscando arquivo.
Buscando arquivo.
Arquivo encontrado.
Aluno1;10;10
Aluno2;9;9
Aluno3;9;9
```

Fonte: elaborado pelo autor.

Com essa seção encerramos a parte de programação paralela. Você pode conhecer diversas formas de implementar esse poderoso recurso, porém, a escolha correta da maneira de fazer o processamento garante que um sistema seja mais robusto, com manutenção mais fácil e que seus componentes possam ser reaproveitados.

Pesquise mais

Existem diversas fontes para avançar nos conhecimentos sobre programação paralela em Java:

- SOHAM. Java - Making a timer. [S.l.], 31 dez. 2014. Disponível em: <https://www.youtube.com/watch?v=36jbBSQd3eU>. Acesso em: 19 jul. 2018.
- SHISHIDO, H. Y.; Huff, A. **Algoritmo paralelo do teorema de convolução em imagens em OpenMP para Java**. Computer on the Beach, [s.l.], 2018. Disponível em: <https://siaiap32.univali.br/seer/index.php/acotb/article/download/5345/2802>. Acesso em: 19 jul. 2018.

Sem medo de errar

Atualmente, você trabalha na equipe de *backend* de uma empresa especializada em migração de sistemas legados. O projeto consiste na migração de um sistema de universidades e o seu objetivo é produzir o mesmo código que foi feito com threads para processamento dos dados do vestibular. Porém, agora será preciso aumentar a coesão, manter o acoplamento baixo e encapsular mais o código, além de ser necessário que todas as threads acumulem os dados em apenas uma instância para gerar dados estatísticos. Com

isso, deve-se utilizar métodos sincronizados, além das classes *Timer* e *TimerTask*.

No Quadro 2.24 está apresentada a primeira classe que possui os *Timers* em que são feitas as configurações do tempo de execução e o que deve ser processado. Na linha 1 é declarado um objeto da classe que fará a junção das estatísticas, a linha 2 possui o *timer* que fará a criação da thread e a linha 3 possui a classe que fará o processamento. Repare que no método `main`, na linha 6, é criada uma instância da classe que fará a centralização dos dados e a mesma instância é passada para as outras classes na linha 7 e 8.

Sem medo de errar

Quadro 2.24 | Classe para controle do tempo de execução e de tarefa

```
     package U2S3;

     import java.util.Timer;

     public class ControladorArquivos{


1.        private EstatisticaUso est;
2.        private Timer timer;
3.        private ProcessadorArquivos processador;


          public ControladorArquivos(String
pCaminhosArquivos, EstatisticaUso pEst)
4.        {
               timer = new Timer();
               est = pEst;
5.             processador = new
ProcessadorArquivos(pCaminhosArquivos,est);
               timer.schedule(processador, 0,1000);
          }
6.        public static void main(String[] args) {
7.
8.             EstatisticaUso st = new
EstatisticaUso();
               ControladorArquivos pa1 = new
ControladorArquivos("c:\\dados1\\",st);
               ControladorArquivos pa2 = new
ControladorArquivos("c:\\dados2\\",st);
          }
     }
```

Fonte: elaborado pelo autor.

O Quadro 2.25 apresenta a classe que fará o processamento e informa a classe `EstatisticaUso` das informações.

Quadro 2.25 | Classe que processa as informações

```
package U2S3;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Scanner;
import java.util.TimerTask;


public class ProcessadorArquivos extends TimerTask {
     private int qtdFiles;
     private int arquivoAtual;
     private String caminhosArquivos;
     private EstatisticaUso est;


      public ProcessadorArquivos(String
pCaminhosArquivos,EstatisticaUso pEst) {


          caminhosArquivos = pCaminhosArquivos;
          est = pEst;
     }
     @Override
     public void run() {
          File diretorio = new File(caminhosArquivos);
          File[] listaDeArquivos = diretorio.
listFiles();
          qtdFiles = listaDeArquivos.length;
          for (int i = 0; i < listaDeArquivos.length;
i++) {
               arquivoAtual = i;
               if (listaDeArquivos[i].isFile()) {
                    String nomeArquivos =
listaDeArquivos[i].getName();


processaArquivo(caminhosArquivos+nomeArquivos);
               }
               System.out.println("Processando arquivo
" + getAtual() + " de " + getTotal());
          }
```

```
    }
    public void processaArquivo(String nomeArquivo)
    {
        Scanner in;
        try {
            in = new Scanner(new
FileReader(nomeArquivo));
            while(in.hasNext() == true)
            {
                String dados[] = in.next().
split(";");
                if(dados[1].compareTo("Ciência da
computação") == 0)
                {
                 est.incrementa();
                }


    }
    public void processaArquivo(String nomeArquivo)
    {
        Scanner in;
        try {
            in = new Scanner(new
FileReader(nomeArquivo));
            while(in.hasNext() == true)
            {
                String dados[] = in.next().
split(";");
                if(dados[1].compareTo("Ciência da
computação") == 0)
                {
                 est.incrementa();
                }


            }


        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    public int getAtual()
    {
        return arquivoAtual;
    }
```

```
    public int getTotal()
    {
        return qtdFiles;
    }
}
```

Fonte: elaborado pelo autor.

O Quadro 2.26 apresenta a classe que possui os métodos que guardarão os dados para gerar os relatórios. Observe que os métodos são sincronizados.

Quadro 2.26 | Classe sincronizada para acúmulo de dados

```
package U2S3;

public class EstatisticaUso {
    private int quantidadeAlunosCurso1;

    public synchronized void incrementa()
    {
      quantidadeAlunosCurso1++;
    }

    public synchronized void decrementa()
    {
      quantidadeAlunosCurso1--;
    }
    public int getDados()
    {
      return quantidadeAlunosCurso1;
    }
}
```

Fonte: elaborado pelo autor.

Avançando na prática

Monitor de espaço em disco

Descrição da situação-problema

Você trabalha em uma empresa que desenvolve sistemas de visão computacional. O projeto do qual está participando consiste em um sistema de processamento de imagens astronômicas, sendo necessário detectar quando a unidade de armazenamento está ficando sem espaço livre. Dessa forma, você deve criar uma thread que execute a cada 10 segundos e que verifique e avise se o espaço em disco está acabando.

Resolução da situação-problema

Para resolver essa questão utilizando a classe *Timer* do Java, são necessárias duas classes, uma que usará *Timer* como no Quadro 2.27 e outra apresentada no Quadro 2.28, que especializa a classe *TimerTask*. No Quadro 2.28, a linha 2 declara o uso da classe que especializa a *TimerTask,* as linhas 3, 4 e 5 configuram o *Timer*, instanciam a classe que faz o processamento e, por fim, o comando na linha 5: `timer.schedule(tarefa,0, 10000)` inicia a `tarefa` com intervalo de 10 segundos (10000 milisegundos).

Quadro 2.27 | Classe que controla a thread para monitoramento do disco

```
     import java.util.Timer;

     public class MonitorUso{
1.       private Timer timer;
2.       private BuscadorUso tarefa;
         public MonitorUso()
         {
3.         timer = new Timer();
4.         tarefa = new BuscadorUso();
5.         timer.schedule(tarefa,0, 10000);
         }
         public static void main(String[] args) {
           MonitorUso rt = new MonitorUso();
         }
     }
```

Fonte: elaborado pelo autor.

A classe do Quadro 2.28 especializa a classe *TimerTask*. Nessa classe, o método `run()` busca o espaço livre em disco (linha 2) e as linhas 3, 4 e 5 verificam e informam o usuário se o espaço está chegando no fim.

Quadro 2.28 | Classe que busca no sistema operacional no espaço em disco

```
      import java.io.File;
      import java.util.TimerTask;
      import javax.swing.JOptionPane;

      public class BuscadorUso extends TimerTask {


1.          public void run() {
              try {
2.                  long tamanho = new
3.    File("c:").getFreeSpace();
4.                  tamanho = (tamanho/1024/1024);
                    if(tamanho < 1000)
5.                  {
                          JOptionPane.
      showMessageDialog(null, "Unidade de armazenamento
      ficando sem espaço disponível");
                    }
              }catch (Exception e) {
                    e.printStackTrace();              }
          }
      }
```

Fonte: elaborado pelo autor.

Faça valer a pena

1. Diversas linguagens de programação propõem maneiras diferentes de tratar o paralelismo. Algumas apresentam formas nas quais a complexidade é mais alta, como na linguagem C. A linguagem Java possui diversas formas de paralelismo para cenários mais simples ou mais complexos.

Qual das opções abaixo apresenta recursos para o paralelismo em Java?

a) `java.util.Timer` e `java.lang.Thread`.

b) `java.swing.JFrame` e `java.swing.JFrame`.

c) `java.lang.StringBuilder` e `java.lang.StrungBuffer`.

d) `java.util.Threads` e `java.lang.Parallel`.

e) `java.lang.Proc` e `java.swingx.Procstt`.

2. Ao implementar um sistema paralelo, é necessário utilizar os mecanismos que a linguagem propicia. Porém, certos recursos como conexões de banco de dados, variáveis que acumularão resultados e outros devem ser acessados por apenas uma thread por vez.

Leia as afirmações abaixo:

I - Os métodos *synchronized* são utilizados para garantir o acesso a métodos de forma singular.

II - O Java possui um limite de thread que pode ser lançado baseado no *clock* do processador.

III - O Java quando utilizado no Microsoft Windows não possui suporte a threads.

IV - A classe *Timer* em Java encapsula a utilização de threads.

V - Não é possível utilizar as classes Thread e *Timer* no mesmo sistema em Java.

Escolha a opção que apresente apenas as afirmações corretas:

a) Apenas I e II estão corretas.

b) Apenas I, II e III estão corretas.

c) Apenas I e IV estão corretas.

d) Apenas IV e V estão corretas.

e) Nenhuma afirmação está correta.

3. A classe *Timer* do Java é muito útil pois encapsula o comportamento da classe Thread, tornando o processo da programação paralela em Java mais simples do que utilizar um sistema diretamente com as threads. Ainda, a utilização da classe *Timer* facilita nos processos de coesão e acoplamento de um sistema orientado a objetos, pois separa em duas classes as ações de controle da Thread e o processamento.

As classes *Timer* e *TimerTask* possuem papéis diferentes. Quais são eles, respectivamente?

a) Processamento e controle de execução das *threads*.

b) Busca do horário de sistema e apresentação dos tempos.

c) Criação de *pool* de *threads* e processamento.

d) Remoção de dados e separação de dados.

e) Controle de execução das *threads* e processamento.

Referências

DEITEL, P.; DEITEL, H. **Java - como programar**. 10. ed. São Paulo: Pearson, 2016. 968 p.

FURGERI, S. **Java 8 - Ensino Didático - Desenvolvimento e Implementação de Aplicações**. São Paulo: Érica, 2015. 320 p.

HOPE, Computer. **Computer processor history.** 2017. Disponível em: <https://www.computerhope.com/history/processor.htm>. Acesso em: 20 jul. 2018.

HORSTMANN, C. **Core Java Volume I - Fundamentals**. 10. ed. New York: Prentice Hall, 2016. 1040 p.

MANZANO, J. A. G.; COSTA JR., R. **Programação de Computadores com Java**. São Paulo: Érica, 2014. 160 p.

TANENBAUM, A. S.; BOS, H. **Sistemas operacionais modernos**. 4. ed. São Paulo, 2016. 864 p.

WINDER, R.; GRAHAM, R. **Desenvolvendo Software em Java**, 3. ed. Rio de Janeiro: LTC, 2009. 696 p.

Padrões de projeto, ferramentas e métodos ágeis

Convite ao estudo

No início da computação o software era considerado um elemento com menos importância no computador, algo que tinha um papel menor comparado ao hardware de um grande computador. Todavia, como a utilização do computador se tornou algo mais geral, a quantidade dos usuários e desenvolvedores de software aumentou, necessitando de mais ferramentas e mais softwares. De forma gradativa, o software foi se tornando altamente relevante, ao ponto de, em muitos casos, ser mais importante que o hardware que o executará. Com esse avanço na área de software, houve um aumento na quantidade de pessoas fazendo a codificação de um programa e na necessidade de qualidade dessas soluções.

Com a devida importância dada ao software, nessa unidade você vai conhecer e compreender os padrões de projeto, o relacionamento de classes e ferramentas para teste, considerando as boas práticas de programação orientada a objetos. Para isso, serão abordadas formas padronizadas de resolução de problemas orientados a objetos, conhecidas como padrões de projetos. Você também conhecerá ferramentas para gerenciar a forma de geração da documentação do software e fazer o controle de versão, além de novas metodologias para o desenvolvimento mais dinâmico e para a produção de software ligada às necessidades e importâncias que o cliente os usuários prezam.

Imagine que você trabalha em uma empresa de migração de sistemas legados. Em um primeiro momento, estava na equipe de *frontend*, mas após demonstrar seu potencial, você

foi enviado para a equipe de *backend*, onde pode desenvolver códigos mais complexos e ganhar mais experiência. Todavia, diversos problemas estão sendo enfrentados pela sua equipe e você foi escolhido para buscar as formas mais modernas de aplicar a solução. A primeira questão é como testar uma aplicação de forma automatizada para reduzir os defeitos no código, como documentar os códigos de maneira padronizada e como utilizar um controle de versões e trabalho coletivo. O segundo problema está relacionado à maneira de resolver problemas recorrentes no projeto de forma padronizada. A última situação é a aplicação de um banco de dados com mais desempenho para a parte de processamento de maior porte do sistema. Assim, quais são as formas de organizar os códigos com grandes equipes? Como são resolvidos os problemas orientados a objetos de forma padronizada? Qual tipo de banco deve-se utilizar para ter mais desempenho?

Nessa unidade serão apresentadas ferramentas para teste unitário, forma de documentação de código Java e controle de versão utilizando Eclipse e Git, as maneiras de padronizar os problemas recorrentes em um projeto orientado a objetos e, por fim, a utilização de metodologias ágeis em projetos orientados a objetos.

Seção 3.1

Ferramentas para programação em linguagens orientadas a objetos

Diálogo aberto

O desenvolvimento de software pode ser comparado com a construção de um prédio: certas tarefas podem ser feitas em paralelo ou por mais de uma pessoa. A construção das paredes de um cômodo pode ser feita por mais de uma pessoa, cada uma assentando seus blocos individualmente, porém, existe um projeto detalhado para padronizar a construção. No desenvolvimento de software temos necessidades semelhantes de controle de trabalho paralelo, sendo preciso recorrer a determinados padrões para garantir a qualidade do sistema.

Você faz parte da equipe de desenvolvimento *backend* de uma empresa especializada em migração de sistemas legados. O desenvolvimento do projeto da migração de um sistema de universidade estava indo muito bem, todos os prazos estavam sendo atingidos. Todavia, como o código começou a ficar cada vez maior, diversos problemas foram aparecendo. A primeira indicação de que algo estava errado foram os chamados de suporte feitos pelos clientes, cada um relacionado a módulos diferentes do sistema. Os problemas eram simples de resolver, mas estavam minando qualidade na visão do usuário. Ainda, como os módulos eram diferentes e codificados por pessoas diferentes, em alguns casos a correção de uma pessoa era incorporada à versão enviada ao cliente e em outros casos as modificações eram sobrescritas. Além disso, quando uma pessoa responsável pelo módulo estava ausente, o tempo de resposta subia muito devido à falta de documentação.

Por causa disso, você foi escalado para resolver esses problemas e apresentar a maneira como o fez. Dessa forma, você deve indicar recursos para testar um código e minimizar os defeitos, além de documentar e controlar a alteração do código pela equipe. Para isso, utilize o código do Quadro 3.1.

Quadro 3.1 | Classe para apresentar as formas de controle de testes, versão e documentação

```
package U3S1;

public class ControleNotas {
     private double N1;
     private double N2;
     private double N3;
     private double mediaFinal;

     public ControleNotas(double n1, double n2, double
n3) {
          N1 = n1;
          N2 = n2;
          N3 = n3;
          mediaFinal = 0.0;
     }
     public double calculaNotaFinal()
     {
          mediaFinal = (N1*0.4)+ (N2*0.3) + (N3*0.3);
          return mediaFinal;
     }
}
```

Fonte: elaborado pelo autor.

Nessa etapa será importante melhorar os aspectos de qualidade do código para refletir no produto final, tornando os programadores e o sistema mais profissionais. Assim, vamos produzir sistemas grandes e de qualidade? Para isso, serão estudadas as formas de controlar a versão do código e ferramenta para documentar os códigos e fazer testes de método da aplicação.

Não pode faltar

Os testes em software são essenciais para que o produto entregue tenha os requisitos de qualidade imposto pelo cliente ou os cenários nos quais o sistema será implementado. Para avaliar esses testes, devemos primeiramente observar três conceitos básicos sobre testes (AMMANN, 2008):

1. **Defeito**: implementação incorreta feita em uma aplicação.
2. **Erro**: manifestação desse defeito ou falha.
3. **Falha**: comportamento externo que resulta em problemas de execução de um sistema (por exemplo, problemas de rede ou hardware).

Dados esses cenários, podemos criar um grande arcabouço para fazer com que uma aplicação tenha seu número de defeitos minimizado e, assim, menor probabilidade de erros. Para isso, podemos executar diversas formas de testes obter um cenário com poucos erros. Também, é possível investir em uma documentação acurada e em um sistema de controle de versão de código, a fim de aumentar a confiabilidade do sistema e seu desenvolvimento. A forma inicial para a produção de testes são os testes unitários.

Testes unitários em Java

Existem diversas categorias de testes, tais como de módulos, de integração, de sistema e de validação (SOMMERVILLE, 2016). Um dos testes iniciais que podem ser feitos é o unitário, cujo objetivo é verificar se os métodos estão funcionando. Umas das ferramentas utilizadas para realizar esse tipo de teste é o JUnit, com o qual é possível fazer diversas avaliações para confirmar se o código está funcionando corretamente (TAHCHIEV, 2010). O teste unitário pode ser comparado a testar e validar todas as peças de um carro antes de fazer a montagem. Se as peças estão funcionando corretamente, as chances de a montagem funcionar aumentam.

Vamos utilizar um exemplo para demonstrar a criação de testes unitários. Considere a classe `Calculadora` criada no Quadro 3.2. Ela faz um conjunto de operações básicas, como somar (linhas 3 e 4), subtrair (linhas 5 e 6), dividir (linhas 7 e 8) e multiplicar (linhas 9 e 10). Com base nessa classe vamos desenvolver os testes unitários.

Quadro 3.2 | Classe exemplo para a construção de testes unitários

```
1.  package U3S1;
2.  public class Calculadora {
3.       public double somar(double a, double b)    {
4.            return a + b;
          }
5.       public double subtrair(double a, double b){
6.            return a - b;
          }
7.       public double dividir(double a, double b) {
8.            return a / b;
          }
9.       public double multiplicar(double a , double
     b) {
10.           return a * b;
          }
     }
```

Fonte: elaborado pelo autor.

Assimile

Os testes unitários têm o objetivo de validar o funcionamento de um método, sendo, assim, possível detectar problemas e validar a coesão e o acoplamento de cada elemento. Desta forma, pode-se validar o funcionamento das partes, minimizando o erro quando esses componentes forem conectados.

Para executar esses testes, é necessário que alguma ferramenta auxilie no processo juntamente com o JUnit. O IDE Eclipse já possui integração para fazer os testes unitários, cuidando da sua criação e execução através de uma determinada classe. Ao utilizar o Eclipse, é preciso criar um *JUnit Test Case,* seguindo o menu *File->New->JUnit Test Case*. Na tela exibida, escolha uma pasta para armazenar os testes, um nome para o projeto e uma classe para testar no campo "*Class under test*". Os demais parâmetros você pode deixar com os valores padrão.

Utilize o link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/programacao-orientada-a-objetos-II/u3/s1/imagem_poo_II.jpg> ou o QR Code para ver a tela de criação de uma *New JUnit Test Case*.

O Quadro 3.3 apresenta um conjunto de testes elaborados para testar cada método da classe `Calculadora`. As linhas de 1 a 4 definem o pacote, as bibliotecas e a classe (isso é configurado de forma automática quando se cria a classe pelo Eclipse).

Dentro da classe `CalculadoraTeste` temos quatro métodos, cada um relacionado a um método da classe `Calculadora`. A marcação `@Test` se refere ao JUnit para afirmar que o método imediatamente abaixo é um teste. Utilizaremos como exemplo o teste entre as linhas de 6 a 9 chamado **`void`** `testSomar()`. Repare que nesse método, na linha 7, a classe `Calculadora` (`Calculadora c = new Calculadora();`) é instanciada e o método **`double`** `res = c.soma(10, 50);` é utilizado, guardando o resultado em uma variável. Por fim, na linha 8 é utilizado o *`assertEquals`*`(60, res);`. O `assert` vem do inglês afirmar ou confirmar, então, esse método vai comparar o `60` com o valor de `res`. Se esses valores forem iguais, é considerado como sucesso, mas caso seja diferente, significa que o teste falhou. Os outros métodos seguem a mesma lógica, fazendo a instância da classe que se quer testar e utilizando o `assert` para validar o resultado. Assim, a lógica dos métodos pode ser testada, evitando enviar para o cliente resultados inesperados.

Quadro 3.3 | Classe com os testes unitário para a classe "Calculadora".

```
1.  package testes;
2.  import static org.junit.jupiter.api.Assertions.*;
3.  import org.junit.jupiter.api.Test;

4.  class CalculadoraTeste {
5.      @Test
6.      void testSomar() {
7.          Calculadora c = new Calculadora();
8.          double res = c.somar(10, 50);
9.          assertEquals(60, res);
        }
```

```
10.  @Test
11.       void testSubtrair() {
12.            Calculadora c = new Calculadora();
13.            double res = c.subtrair(10, -15);
14.            assertEquals(25,res);
           }
15.            @Test
16.       void testDividir() {
17.            Calculadora c = new Calculadora();
18.            double res = c.dividir(10.0, 0);
19.            assertNotEquals(Double.isFinite(res),
     false);
           }
           @Test
20.       void testMultiplicar() {
21.            Calculadora c = new Calculadora();
22.            double res = c.multiplicar(120.115,
     12.5465);
23.            assertEquals(1507.01,0.5,res);
24.       }
25.  }
```

Fonte: elaborado pelo autor.

O Quadro 3.4 apresenta mais alguns exemplos de `assert` que podem ser utilizados.

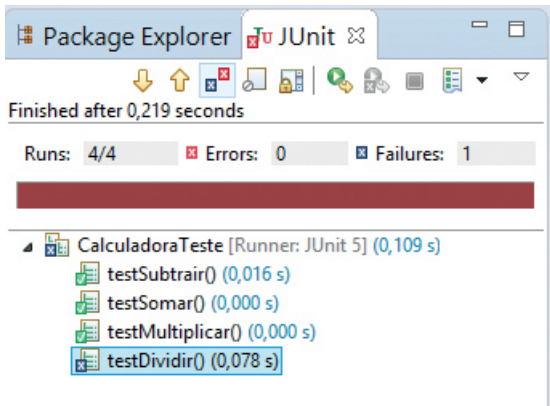
Quadro 3.4 | Alguns tipos de testes para a resposta dos sistemas

Tipo de **assert**:	Descrição:
`assertArrayEquals(float[] expected, float[] actual)`	Verifica se `float[] expected` é igual ao `float[] actual`, existem diversas sobrecargas para **double, int** e outros.
`assertNull(Object actual)`	Verifica se a instância passada é `null`.
`assertTrue(boolean condition)`	Verifica se a condição passa é `true`.

Fonte: elaborado pelo autor.

A Figura 3.1 apresenta o resultado do uso do Eclipse para essa classe de teste. O processo de execução dos testes é o mesmo que o de um software. Você pode clicar com o botão direito do mouse na classe de teste e escolher *Run As*, e o próprio Eclipse detecta que a classe é um teste, troca a interface utilizando os arcabouços do JUnit e exibe os resultados. Nesses testes é possível reparar que um deles (*testDividir*) falhou, sendo necessário validar se o código realmente possui um defeito ou se foi elaborado de forma incorreta. Nesse caso, o teste esperava que o resultado não fosse "Infinity", todavia, o resultado de um número divido por zero utilizando o tipo **`double`** é sempre "Infinity". Dessa forma, o programador deve retornar à classe `Calculadora` e implementar uma forma de evitar divisão por zero.

Figura 3.1 | Resultado da execução do teste



Fonte: captura de tela do Eclipse, elaborada pelo autor.

Dessa forma, é possível verificar que os testes unitários são a primeira linha de defesa contra os bugs, o que ajuda a garantir que cada parte de um sistema esteja funcionando e, ainda que em caso de modificações, o sistema ainda mantenha seu funcionamento. Em certos cenários de desenvolvimento e algumas ferramentas não permitem que o software seja liberado ou integrado a uma aplicação sem que antes todos os testes unitários tenham sucesso. Outra forma de garantir que defeitos não sejam gerados em um sistema é a utilização correta da documentação de um sistema.

Reflita

Os testes unitários são a primeira forma de teste de um sistema, com eles é possível validar cada método de um sistema. Porém, todos os métodos implementados são testáveis quando desenvolvemos sistemas? Todos retornam valores para que seja validado o seu resultado?

Documentação de código fonte com o Javadoc

O Javadoc é uma ferramenta produzida pela própria Oracle que documenta todos os códigos do sistema implementado em Java. Com ela é possível fazer diversas marcações no decorrer do código para gerar uma versão em HTML para explicar o funcionamento de cada classe, método e outros de uma aplicação (DEITEL; DEITEL, 2016). Essa documentação tem o objetivo de explicar a função de cada elemento do código para os desenvolvedores, e não diretamente para o usuário (HORSTMANN, 2016). A documentação gerada por essa ferramenta é feita por meio de anotações específicas, utilizadas para identificar as partes do código e estruturar o documento HTML para que fique organizado. Algumas anotações usadas com frequência estão especificadas no Quadro 3.5.

Quadro 3.5 | Exemplos de marcação do Javadoc

Marcação	Descrição
@author	Define o autor da classe.
@see	Classes correlatas à que está sendo desenvolvida.
@since	Versão da classe.
@param	Parâmetro utilizado no método.
@return	Descrição do retorno do método.
@code	Altera a fonte do texto para indicar que é um código Java.

Fonte: elaborado pelo autor.

O Quadro 3.6 apresenta a classe `Calculadora` que foi alterada para ser documentada com o Javadoc.

Quadro 3.6 | Exemplo de utilização do Javadoc

```
/**
 * A classe calculadora é responsável por executar as
operações matemáticas básicas
 * @author  Fabio Andrijauskas
 * @see     java.lang.Math
 * @since   1.0
 */
public class Calculadora {
     /**
      * O método ({@code somar} faz a soma de dois nú-
meros
      *
      * @param  a valor do primeiro número a ser soma-
do.
      * @param  b valor do segundo número a ser soma-
do.
      * @return valor da soma de a e b
      * @see Math
      */
     public double somar(double a, double b)    {
          return a + b;
     }
     /**
      * O método ({@code subtrair} faz a subtração de
dois números
      *
      * @param  a valor do primeiro número a ser sub-
traido.
      * @param  b valor do segundo número a ser sub-
traido.
      * @return valor da subtração de a e b
      * @see Math
      */
     public double subtrair(double a, double b){
          return a - b;
     }
     /**
```

```
     * O método ({@code dividir} faz a divisão de
dois números
     *
     * @param  a valor do primeiro número a ser divi-
dido.
     * @param  b valor do segundo número a ser divi-
dido.
     * @return valor da divisão de a e b
     * @see Math
     */
    public double dividir(double a, double b) {
        return a / b;
    }
    /**
     * O método ({@code multiplicar} faz a multipli-
cação de dois números
     *
     * @param  a valor do primeiro número a ser mul-
tiplicado.
     * @param  b valor do segundo número a ser multi-
plicado.
     * @return valor da multiplicação de a e b
     * @see Math
     */
    public double multiplicar(double a , double b) {
        return a * b;
    }
}
```

Fonte: elaborado pelo autor.

O Eclipse permite gerar toda a documentação com o Javadoc utilizando a opção *Project > Generate Javadoc*. A Figura 3.2 apresenta uma porção da documentação gerada da classe Calculadora. É possível verificar que ele utiliza a mesma formatação da documentação do Java, pois usa a mesma ferramenta. A documentação é criada na pasta do projeto de origem.

Figura 3.2 | Porção da documentação gerada

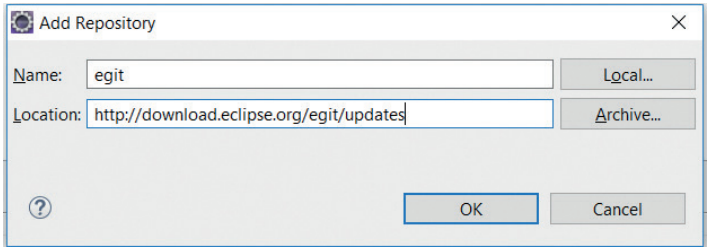
Fonte: elaborada pelo autor.

A junção dos testes unitários e o mecanismo de documentação são um grande avanço na qualidade do desenvolvimento. Além desses dois elementos, é de grande importância uma forma de gerenciar as versões de um software e as contribuições de diversas pessoas da equipe.

Sistema de controle de versão

Existem diversos sistemas de controle de versão, como o *Concurrent Versions System* (CVS), *Apache Subversion* (SVN), *git* e outros. Um que está em grande destaque é o *git*, um sistema de controle de versão muito simples e versátil e, para utilizá-lo, é possível fazer a integração via Eclipse ou usar diretamente no terminal (CHACON, 2018). Faremos com o Eclipse, pois propicia uma interface gráfica. O primeiro passo consiste em instalar o *plugin* do *git* no eclipse, em *Help* > *Install new software*. Então, deve-se adicionar um novo repositório (botão *Add* ao lado do *Manage*) e utilizar o link do *plugin*, disponível em: <http://bit.ly/2MG17oX> (acesso em: 20 ago. 2018), conforme Figura 3.3.

Figura 3.3 | Inclusão de repositório do *egit* para o Eclipse



Fonte: elaborada pelo autor.

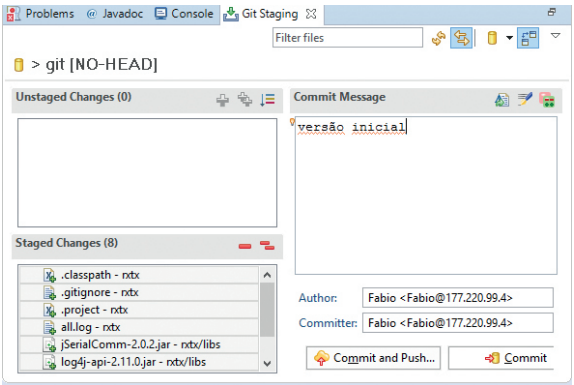
Após esse passo, é possível selecionar o *egit* em *Work with*. Deixe o restante de forma padrão, selecionando todos os componentes. Depois da instalação, é necessário criar um repositório *git* local. Ao clicar com o botão direto do mouse sob o projeto no *Package Explorer*, é possível selecionar a opção *Team* > *Share Project*. Na tela exibida, clique em *Create* para criar um repositório local (não pode ser no mesmo local do projeto). Com isso, o projeto estará com sistema de controle de versão *git*. Em seguida, é necessário fazer o primeiro *push* (envio dos arquivos) e *commit* (gravação das modificações do repositório). Para isso, é necessário clicar em *Team* > *Commit* para ver a interface da Figura 3.4. Nela você deve selecionar todos os arquivos e clicar no botão . Em seguida, no campo *Commit Message*, crie uma mensagem inicial e, por fim, utilize a opção *Commit and push*. Com isso, é possível configurar um servidor remoto para fazer a sincronização dos dados. Existem diversos serviços que proveem um repositório *git*, por exemplo o github.

Assimile

Com o uso do sistema de versão, o procedimento de desenvolvimento acontece da seguinte maneira:

- Cria-se o projeto.
- Compartilha-se em um servidor de *git*.
- Envia-se a primeira versão.
- Desenvolve-se o código.
- Envia-se para o servidor com o *Commit and Push*.

Figura 3.4 | Envio da primeira versão utilizando o *git*



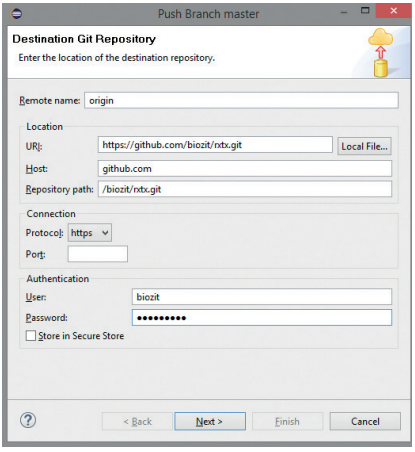
Fonte: elaborada pelo autor.

A criação da conta do repositório é feita no endereço <https://github.com> (acesso em: 20 ago. 2018). Uma URL é gerada e deve ser utilizada na etapa de configuração do Eclipse após o *Commit and push*, conforme a Figura 3.4.

Exemplificando

Existem diversas formas de utilizar os repositórios, e uma delas é da Figura 3.5, em que se cria um repositório no gitbub. Nessa interface é necessário inserir os dados da conta do github.

Figura 3.5 | Criação de repositório no github



Fonte: elaborada pelo autor.

Com isso, todas as modificações feitas no código devem ser salvas localmente e enviadas clicando com o botão direito do mouse no projeto em *Team* > *Commit and push*.

Com essa etapa dos estudos é possível avançar nas formas de produção, controle e documentação, além de produzir código profissionais e duradouros.

Pesquise mais

Existem diversas fontes que contêm mais exemplos e formas com os testes unitários e controle de versão:

BECHTOLD, S. et al. **JUnit 5 User Guider**. JUnit, [s.l.], [s.d.]. Disponível em: <http://bit.ly/2N8CIVS>. Acesso em: 20 ago. 2018.

ECLIPSE FOUNDATION. **EGit/User Guide**. Eclipse Foundation, [s.l.], [s.d.]. Disponível em: <http://bit.ly/2OR2IWj>. Acesso em: 20 ago. 2018.

GIT. **Documentation**. Git, [s.l.], [s.d.]. Disponível em: <http://bit.ly/2vXJfMM>. Acesso em: 20 ago. 2018.

CHACON, S.; STRAUB, B. **Pro Git**. Git. Apress: Nova Iorque. 9 nov. 2014, 2. ed., 456 p. Disponível em: <http://bit.ly/2wfwmNk>. Acesso em: 20 ago. 2018.

Sem medo de errar

Você trabalha na equipe de *backend* de uma empresa especializada em migração de sistemas legados. O sistema estava sendo desenvolvido de forma tranquila e sem apresentar problemas sérios nos códigos, porém, com o decorrer do projeto e o aumento significativo da quantidade de classes e métodos, diversos problemas começaram a ocorrer, como:

- Problemas de testes básicos em métodos.
- Falta de documentação e entendimento das classes e métodos.
- Problemas de sincronização de códigos quando alterados ou desenvolvidos pela equipe que aumentou.

Para resolver essa questão, em primeiro momento foi aplicada uma ferramenta de teste unitário, gerando o código no Quadro 3.7.

Quadro 3.7 | Testes unitários para a classe ControleNotas

```
package U3S1;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class ControleNotasTest {

     @Test
     void testControleNotas() {
          ControleNotas c = new ControleNotas(10.0,
10.0, 10.0);
          assertNotNull(c);
     }

     @Test
     void testCalculaNotaFinal() {
          ControleNotas c = new ControleNotas(10.0,
10.0, 10.0);
          double notafinal = c.calculaNotaFinal();
          assertEquals(10.0, notafinal);
     }
}
```

Fonte: elaborado pelo autor.

Para questão de documentação, deve-se utilizar o Javadoc, conforme o Quadro 3.8.

Quadro 3.8 | Classe de cálculo de nota documentada

```
package U3S1;
/**
 * A classe para calcular a nota final do aluno.
 * @author  Fabio Andrijauskas
 * @see     java.lang.Math
 * @since   1.0
 */

public class ControleNotas {
```

```
    /**
     * Nota do primeiro bimestre do aluno.
     */
    private double N1;

    /**
     * Nota do segundo bimestre do aluno.
     */
    private double N2;

    /**
     * Nota da avaliação final semestral do aluno.
     */
    private double N3;

    /**
     * Media final do semestre
     */
    private double mediaFinal;

    /**
     * O construtor da classe ({@code ControleNotas})
para média final dos alunos
     *
     * @param n1 Nota do primeiro bimestre do aluno.
     * @param n2 Nota do segundo  bimestre do aluno.
     * @param n3 Nota do final bimestre do aluno.
     * @see Math
     */

    public ControleNotas(double n1, double n2, double
n3) {
          N1 = n1;
          N2 = n2;
          N3 = n3;
          mediaFinal = 0.0;
    }
```

```
    /**
     * O método ({@code calculaNotaFinal} para média
final dos alunos
     *
     * @return média final do aluno
     * @see Math
     */

    public double calculaNotaFinal()
    {
        mediaFinal = (N1*0.4)+ (N2*0.3) + (N3*0.3);
        return mediaFinal;
    }
}
```

Fonte: elaborado pelo autor.

Para fazer o controle de versão, é necessário:

1. Instalar o *plugin* egit no Eclipse.
2. Compartilhar o código fonte mais avançado com a opção *Team > Share project*.
3. Fazer o *Commit and Push* nos casos necessários.

Avançando na prática

Robustez de um sistema de controle de temperatura

Descrição da situação-problema

Uma empresa que produz sistemas para controle de monitoramento de data center lhe contratou para a equipe que desenvolve o sistema de monitoramento ambiental. O projeto consiste em um controle de temperatura para data center e precisa guardar o valor de N temperaturas do ambiente. Porém, por diversas vezes o sistema informou valores errados, conforme a classe em questão que o Quadro 3.9 apresenta. Para resolver isso, seu líder solicitou a implementação de um teste unitário que fará a verificação desse método. Com isso, antes de o código entrar em produção, é possível verificar o seu funcionamento.

Quadro 3.9 | Classe em que se deve implementar o teste unitário

```
package U3S1;

public class MediaTemperatura {

     private int qtdMax;
     private double[] temperaturas;
     private int atual;

     public MediaTemperatura(int parQtdMax)
     {
          qtdMax = parQtdMax;
          temperaturas = new double[qtdMax];
          atual = 0;
     }

     public void setLeitura(double parTemp)
     {
          temperaturas[atual] = parTemp;
          atual++;
     }

     public double mediaTemp()
     {
          double soma = 0.0;
          for (int i = 0; i < atual; i++) {
              soma = soma + temperaturas[i];
          }
          return soma/atual;
     }
}
```

Fonte: elaborado pelo autor.

Resolução da situação-problema

Para resolver e validar a questão, deve-se utilizar testes unitários. O Quadro 3.10 apresenta a classe que implementa esse teste.

Quadro 3.10 | Teste unitário para a média da temperatura

```
package U3S1;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class MediaTemperaturaTest {

     @Test
     void testMediaTemp() {
          MediaTemperatura m = new MediaTemperatura(5);
          m.setLeitura(25.0);
          m.setLeitura(30.0);
          m.setLeitura(30.0);
          m.setLeitura(28.0);
          m.setLeitura(20.0);
          double res = m.mediaTemp();
          assertEquals(26.6, res,0.1);

     }
}
```

Fonte: elaborado pelo autor.

Faça valer a pena

1. Existem diversos tipos de testes que podem ser elaborados para verificar e validar um software. Alguns deles não necessitam de código-fonte para serem feitos e são elaborados utilizando a mesma interface que um usuário usaria. Porém, há testes que necessitam do código-fonte para serem elaborados, e um deles é o teste unitário.

Qual é a função principal de um teste unitário utilizando JUnit?
a) Controlar a versão dos testes do sistema.
b) Verificar o funcionamento da classe como um todo.
c) Verificar o funcionamento dos métodos de uma classe.
d) Testar o sistema de controle de documentação.
e) Validar o sistema e seus módulos.

2. A documentação de um sistema é um elemento de grande importância para que seja possível explicar o que e como uma aplicação foi desenvolvida e quais são as formas de uso de um método, informando o objetivo da classe e os métodos, além de descrever os parâmetros necessários para utilizar os métodos apresentados. Para o Java, é possível utilizar o Javadoc.

Assinale a opção que possui apenas as marcações do Javadoc.
a) @author, @see, @since, @param e @return.
b) @option, @see, @control, @param e @return.
c) !author, !see, !since, !param e !return.
d) author, see, since, param e return.
e) @attr, @see, @since, @param e @opt.

3. Para garantir que diversas pessoas utilizem um código, é essencial usar um sistema que gerencie as versões e os conflitos. Esses sistemas de controle de versão podem receber códigos de diversas linguagens. No Java é possível utilizar o Eclipse com o *plugin* para ter acesso ao *git*.

Utilizando o *git*, qual opção é necessária para enviar e notificar o sistema de versão de novos arquivos e enviá-los para servidor?
a) control.
b) get.
c) commit.
d) push.
e) commit e push.

Seção 3.2

Padrões de projetos em orientação a objetos

Diálogo aberto

Na construção de uma casa são utilizados diversos elementos que resolvem os problemas de maneira padronizada. Por exemplo, existem vários tamanhos de blocos para construir as paredes, que já seu uso está previsto no projeto e, portanto, seguem um padrão. Pensando em elementos padronizados para resolver problemas durante o desenvolvimento de um sistema, quem nunca questionou: será que essa é a forma correta de resolver esse problema? Existe algum padrão de como implementar esse problema? Nessa seção conheceremos alguns padrões que podem ser seguidos para garantir a qualidade do software.

Você trabalha em uma empresa especializada em migração de sistemas legados e está alocado na equipe de *backend*, no projeto de migração de um sistema universitário. Seus colegas de trabalho aceitaram suas sugestões sobre as ferramentas para maximizar a produtividade, mas alguns elementos de programação ainda estão causando problemas, e um dos mais impactantes está relacionado à conexão do programa com o sistema de gerenciamento de banco de dados. Em diversos pontos do código, os desenvolvedores instanciam conexões com o banco, causando uma utilização de recursos em exagero, no sistema e no servidor de banco de dados. Consequentemente, em alguns casos, o grande fluxo de informações causa indisponibilidade no servidor que hospeda o sistema de gerenciamento de banco de dados. Para resolver a situação, você deve:

- Procurar nos padrões de projeto qual deles pode ser aplicado para resolver esse problema.
- Desenvolver o código com a conexão com o banco de dados utilizando esse padrão de projeto.

Para resolver essa questão, vamos estudar um padrão de cada tipo, com sua descrição, exemplo e aplicação. Com isso, você estará apto a utilizar esses padrões e familiarizado com o design *patterns*

e, assim, poderá buscar soluções para problemas recorrentes nos projetos orientados a objeto.

Não pode faltar

O desenvolvimento de software deve ser encarado como uma atividade correlacionada com a produção de outros elementos, como de circuito eletrônico, de uma casa ou de um motor. Em alguns momentos, é necessária a produção de um item específico para que o projeto funcione, por exemplo, um resistor de potência ou resistência que não pode ser encontrado no mercado geral ou um tipo ou tamanho de bloco de resistência que não está no catálogo de produção da fábrica. Todavia, pensando no processo como um todo, a maioria das peças utilizadas para resolver os problemas do cotidiano pode ser comprada, sem a necessidade de pedidos especiais. Porém, no desenvolvimento de software, não pensamos dessa forma, ou seja, criamos soluções mais genéricas e, em diversos casos, elas são para problemas recorrentes em diversos projetos de software, para os quais alguém já pode ter disponibilizado a solução.

Pense em um software que deve se conectar em um sistema de gerenciamento de banco de dados (SGBD), como um sistema *Enterprise Resource Planning* (ERP), que fará a conexão com o SGBD e, de forma geral, é necessária apenas uma conexão. Utilizando os mecanismos da orientação a objetos é possível garantir que, ao utilizar as classes de conexão do banco de dados, apenas uma conexão seja disponibilizada para todas as outras classes? Pensando nesse tipo de problema, a utilização de padrões de projetos oferece recursos para minimizar ou solucionar diversos problemas.

Um padrão é a descrição de uma solução para uma estrutura de projeto, tornando essa abordagem reutilizável por todo o software (GAMMA, 2000). Os padrões de projeto são utilizados para minimizar os problemas que podem ocorrer no desenvolvimento de um software. Além disso, estão relacionados à construção de softwares, e os consumidores diretos não são os usuários finais, mas sim os desenvolvedores. Eles podem ser aplicados a todos os tamanhos de projeto, mas alguns fazem sentido apenas em projetos de médio ou de grande porte, pois nesses tipos, as restrições de modelagem são muito maiores.

Assimile

Os padrões de projeto são soluções a serem aplicadas em cenários bem definidos durante o desenvolvimento de um projeto. Os usuários são outros desenvolvedores que farão a reutilização que o padrão propõe.

Existem 23 padrões de projeto catalogados pelos autores Erich Gamma, Richard Helm, John Vlissides e Ralph Johnson. Todavia, pesquisas trazem novas soluções para outros problemas ou para outras formas de programação, como a orientação a aspectos. Focaremos em alguns que possuem aplicação mais geral. Os padrões de projetos são divididos por propósito e escopo. Nesse livro veremos sobre o propósito, o qual podemos dividir em três tipos:

- **Criação**: tem o objetivo de encapsular a criação de elementos como subclasses ou objetos. Esse processo está literalmente relacionado à forma de manter a complexidade dentro das classes ou objetos.
- **Estrutural**: os padrões estruturais têm o objetivo de apresentar uma forma de entender uma parte do sistema de maneira mais simples e padronizada, sempre pensando nos elementos de coesão e acoplamento.
- **Comportamental**: o padrão comportamental tem o foco em apresentar as formas de como um conjunto de objetos podem se relacionar de maneira controlada, deixando claro qual é o fluxo de informação ou notificação.

Exemplificando

O Quadro 3.11 apresenta um exemplo de padrões de projeto, descritos na literatura pelo seu propósito e divididos pelo seu tipo. Cada um deles tem uma aplicação específica para resolver situações, dado o cenário apresentado.

Quadro 3.11 | Padrões de projeto organizados dado o propósito de sua aplicação

Propósito		
De criação	*Estrutural*	Comportamental
Factory Method	*Adapter*	*Interpreter*
Abstract Factory	*Bridge*	*Template Method*
Builder	*Composite*	*Chain of Responsibility*
Prototype	*Decorator*	*Command*
Singleton	*Façade*	*Iterator*
	Flyweight	*Mediator*
	Proxy	*Memento*
		Observer
		State
		Strategy
		Visitor

Fonte: adaptado de Gamma (2000, p. 26).

Para exemplificar o uso, vamos ver a aplicação de um padrão de cada tipo. Do tipo de criação, observaremos o *Singleton*, do tipo estrutura o *Façade* e do tipo comportamental o *Template Method*. Esses padrões são relacionados a situações que podem ocorrer em projetos de todos os portes. Portanto, veremos como fazer um código que poderá ser reutilizado, garantindo o acoplamento e a coesão do código.

Reflita

Em grandes projetos, os *design patterns* são essenciais para a produção de código de qualidade, e sua aplicação está sob responsabilidade de diversas pessoas durante a arquitetura do sistema até a implementação. Em qual etapa se deve pensar no uso dos padrões de projeto? Onde é mais indicado pensar na aplicação dos padrões: durante a produção dos diagramas UML ou na implementação?

O primeiro padrão que vamos explorar será o projeto de criação chamado *Singleton*. O padrão de projeto *Singleton* está relacionado à necessidade de manter apenas uma instância de certa classe para

o sistema inteiro, criando um ponto de acesso global para o sistema (GAMMA, 2000). Como exemplos de situações, podemos destacar a necessidade de garantir que se tenha apenas uma conexão ao SGDB, o acesso único ao sistema de controle (como a conexão serial de um Arduino) ou ainda uma fila de impressão.

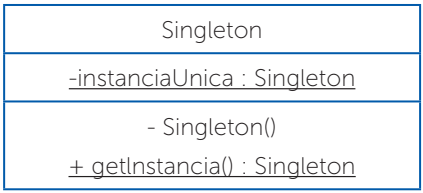
O padrão de projeto *Singleton* usa diversos arcabouços da orientação a objetos para prover a capacidade de fornecer apenas uma instância de algum objetivo específico. Vamos trabalhar com um cenário em que é necessário fornecer apenas uma instância de uma classe que implementa a conexão serial com um Arduino (hardware que possui um microcontrolador com interface de fácil programação. Para mais informações, acesse https://www.arduino.cc/, acesso em: 21 ago. 2018). Esse microcontrolador utiliza um sensor de temperatura e envia as medidas para o computador através de uma porta serial (normalmente emulada pela porta USB). Esse tipo de conexão não pode ser utilizado por mais de uma instância por causa de suas próprias restrições. Caso seja usado, o sistema informará que já existe uma conexão aberta.

Antes de analisarmos o padrão de projeto, vamos relembrar os conceitos de métodos e atributos estáticos no Java. Esses elementos marcados com a palavra-chave ***static*** não precisam da instância da classe para serem utilizadas, e um exemplo do uso é a classe *java.lang.Math*, em que temos diversos métodos ***static***, como *Math.sqrt()*, que calcula a raiz quadrada de um número, ou *Math.PI* que contém o número *pi*. Veja que nesses casos não é necessário usar a palavra ***new***, ou seja, não é criada uma instância da classe *Math*. O uso de métodos ***static*** é direcionado a métodos ou atributos que necessitam de configuração inicial, não sendo necessário um construtor. O mesmo ocorrerá em outros casos, como veremos no padrão *Singleton*.

Antes de programarmos a solução em Java, vamos observar o diagrama da Figura 3.6, que representa a visão do padrão de projeto *Singleton*. Repare que o construtor é privado (- *Singleton()*), assim, apenas elementos de dentro da classe podem instanciar o objeto. Além disso, o atributo do tipo *Singleton* estático (*- instanciaUnica : Singleton*) também é privado, então, novamente, apenas elementos dentro da classe podem utilizá-lo. Essa configuração permite construir a classe apenas dentro dela mesma e, ainda, o atributo

estático só pode ser acessado pelo método estático *getInstancia()*, que é público (veja o sinal de + antes do nome do método) e retorna um objeto da classe *Singleton*, portanto, a única forma de acesso a uma instância da classe é o método *getInstancia()*.

Figura 3.6 | Diagrama de classe do padrão de projeto *Singleton*



Fonte: adaptada de Freeman e Freeman (2007, p. 156).

Para que fique claro o padrão *Singleton*, vamos analisar o código dele implementado em um cenário da conexão serial do Arduino. O Quadro 3.12 apresenta esse código utilizando uma biblioteca *JSerialCom* para leitura da porta (disponível em: <http://bit.ly/2MIcZ9O>. Acesso em: 21 ago. 2018). A linha 2 apresenta o ***import*** necessário para a biblioteca *JSerialComm*. A classe *ComSerialSingleton* possui um atributo não estático na linha 4 do tipo *SerialPort* que depende da instância da classe e um atributo estático na linha 5 (*ComSerialSingleton*). O construtor privado da classe na linha 6 até a 12 faz a configuração quando ela é instanciada pelo método estático na linha 13. Repare que, no método *getInstancia(),* na linha 14, se o atributo estático *comSerial* for **`null`**, ele vai criar uma instância da classe na linha 15 e no método *getInstancia()* ele é **`synchronized`**, com isso, mesmo que diferentes threads chamem o método, será executada apenas uma por vez, garantindo a criação de apenas uma instância da classe. Por fim, no método *retornaDados()* das linhas 17 a 27, é feita a leitura da serial. Esse método também é **`synchronized`** para impedir que duas threads diferentes peçam a leitura da porta serial ao mesmo tempo.

Quadro 3.12 | Implementação do padrão *Singleton* para conexão serial

```
1.  package U3S2;

2.  import com.fazecast.jSerialComm.SerialPort;

3.  public class ComSerialSingleton {
        // instância privada da classe
4.      private SerialPort comPort;
        // atributo static para guarda a instância da
5.  classe
        private static ComSerialSingleton comSerial;

6.      // construtor privado para evitar a instância
    da classe sem controle
7.      private ComSerialSingleton() {
8.
9.          comPort = null;
            try {
10.             comPort = SerialPort.getCom-
11. mPort("COM4");
12.             comPort.openPort();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
13.
        public static synchronized  ComSerialSingle-
    ton getInstancia()
        {
14.         // se a classe nunca foi construida
            if(comSerial == null)
            {
                // usa o construtor private para
15. construir a classe
                comSerial = new ComSerialSingle-
    ton();
            }
```

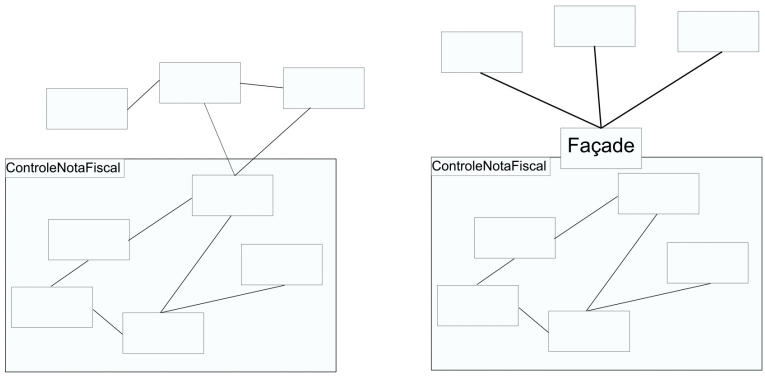
```
16.             return comSerial;
        }
        // como é a mesma instância não se pode dei-
    xar duas threads fazerem a leitura ao mesmo, com
    isso é necessário synchronized
17.     public synchronized String retornaDados()
        {
18.         while (comPort.bytesAvailable() == 0)
19.             try {
20.                     Thread.sleep(20);
21.             } catch (InterruptedException e) {
                        e.printStackTrace();
22.             }
            // cria o buffer de leitura da porta se-
    rial
23.         byte[] readBuffer = new byte[comPort.by-
    tesAvailable()];
24.         // faz a leitura da porta serial
            int numRead = comPort.readBytes(readBuf-
    fer, readBuffer.length);
25.         // cria uma string com os dados vindos
    da porta serial
            String dados = new String(readBuffer);
            return dados;
26.     }
27. }
```

Fonte: elaborado pelo autor.

Além dos padrões de criação, temos os padrões para descrever estruturas de um sistema. Eles são muito úteis para descrever um sistema e corrigir problemas de encapsulamento, coesão e acoplamento, além de terem uma visão mais global do sistema. Um deles é *Façade*, cujo objetivo é prover uma forma de acesso mais simples ao subsistema em um grande software. Como exemplo, podemos pensar em um sistema de processamento de notas fiscais. Para fazer a emissão é necessário usar as classes produto, pedido, nota fiscal, transmissor, validador, impressor e outras. A Figura 3.7 apresenta uma visão da utilidade desse padrão no contexto de processamento de notas. Com ele é possível aumentar a coesão e

diminuir o acoplamento. Repare que na imagem sem *Façade* (lado esquerdo) é necessário que diversas classes tenham relação com classes de todos os níveis do pacote *ControleNotaFiscal*, porém, com o *Façade* (lado direito) é possível ter apenas uma classe de comunicação, não sendo necessário ter acesso a pacotes internos e a complexidade do pacote.

Figura 3.7 | Exemplo de diagrama de classe sumarizado com e sem o padrão *Façade*



Fonte: adaptada de Gamma (2000, p. 179).

O código no Quadro 3.13 apresenta um exemplo da aplicação do padrão *Façade*. Com ele é possível deixar toda a complexidade do uso de um pacote em apenas uma classe. Repare que entre as linhas 3 e 8 são declaradas todas as classes do pacote como atributo. O construtor entre as linhas 9 e 15 fazem a instanciação e o método ***public void*** *criaNotaFiscal()* faz a toda a operação.

Quadro 3.13 | Exemplo de utilização do *Façade*

```
1.  package U3S2;

2.  public class FacadeControleNotaFiscal {

3.      private Produtos[] lstProdutos;
4.      private Pedido pedido;
5.      private NotaFiscal nota;
6.      private Transmissor enviar;
7.      private Validador valida;
8.      private Impressor impressora;
```

```
9.      public  FacadeControleNotaFiscal(int nPedido)
        {
10.         pedido = new Pedido(nPedido);
11.         lstProdutos = pedido.getProdutos();
12.         nota = new NotaFiscal();
13.         enviar = new Transmissor();
14.         valida = new Validador();
15.         impressora = new Impressor();
        }

16.     public void criaNotaFiscal()
        {
17.         nota.insereProdutos(lstProdutos);
18.         enviar.enviarNota(nota);
19.         valida.validar(nota);
20          impressora.imprimir(nota);
        }
    }
```

Fonte: elaborado pelo autor.

Para finalizar a apresentação dos padrões de projeto, temos os padrões comportamentais, cujo objetivo é padronizar a maneira como é feita a comunicação entre as classes. Um deles é o *Template Method,* que define um esqueleto ou um padrão para um conjunto de operações, que são implementadas por subclasses.

Como exemplo, pense em sensores de temperatura via rede sem fio (wi-fi) para automação residencial, mas eles são de marcas diferentes e suas conexões são feitas de formas diferentes. Embora cada marca tenha suas particularidades, o processo padrão consiste em conectar o sensor e retornar à temperatura. Primeiramente, é necessário criar uma classe abstrata (não pode ser instância, possui apenas um modelo e deve ser especializada para ser utilizada). O Quadro 3.14 apresenta um exemplo de classe abstrata, que é o primeiro passo do *Template Method*. Nela definimos a sequência padrão de acesso ao dispositivo, no método `public float retornaTemperatura(String ip, int porta)` entre as linhas 3 e 5. Veja que primeiro será executado o método `conecta()` e depois o `retornaValor()`. Nas linhas 6 e 7, os métodos são definidos com parâmetro abstrato, ou seja, deverão ser sobrescritos na classe que especializar.

Quadro 3.14 | Exemplo de *Template Method* (classe abstrata)

```
1.  package U3S2;

2.  public abstract class ConectorSensor {

3.      public float retornaTemperatura(String ip,
    int porta)
        {
4.          conecta(ip, porta);
5.          return retornaValor();
        }

6.      protected abstract boolean conecta(String
    ip, int porta);
7.      protected abstract float retornaValor();
}
```

Fonte: elaborado pelo autor.

Após criar a classe abstrata, é necessário estender a classe `ConectorSensor` e implementar os métodos `conecta()` e `retornaValor()` com as especificidades de cada sensor. Como amostra, o Quadro 3.15 apresenta um sensor que aceita conexões TCP. Na linha 4, a classe `SensorModeloTCP` que especializa a classe `ConectorSensor`. As linhas 5 e 6 descrevem os atributos para a conexão TCP. Nas linhas 7 a 12 está a implementação do método `conecta()`, que sobrescreve a classe abstrata para definir como conectar no sensor, bem como no método `retornaValor()`, descrito nas linhas de 13 a 24. O ponto principal desse padrão é a utilização dessas implementações entre as linhas entre as 26 a 29. Repare que é declarada a classe `ConectorSensor` que recebe a implementação do `SensorModeloTCP`, assim, quem utiliza a classe tem acesso apenas aos métodos `conectar()` e `retornaValor()`.

Quadro 3.15 | Especialização da classe **ConectorSensor**

```
1.  package U3S2;

2.  import java.io.*;
3.  import java.net.Socket;

4.  public class SensorModeloTCP extends ConectorSen-
    sor{

5.      private Socket recebeSocket;
6.      private BufferedReader leitor;

        @Override
7.      public boolean conecta(String ip, int porta)
8.  {
9.              try {
10.                recebeSocket = new Socket(ip, porta);
11.             } catch (Exception e) {
12.                e.printStackTrace();
                }
                return false;
        }
13.
14.     @Override
15.     public float retornaValor() {
            try {
16.                leitor = new BufferedReader(new In-
    putStreamReader(recebeSocket.getInputStream()));
17.                String dados = leitor.readLine();
18.                return Float.valueOf(dados);
19.             } catch (IOException e) {
20.                e.printStackTrace();
21.             }finally {
22.                try {
23.                     recebeSocket.close();
24.                } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
```

```
25.             return 0;
                }
 26             public static void main(String[] args) {
27.                 ConectorSensor sensor = new SensorMode-
28. loTCP();
29.                                 sensor.conec-
ta("192.168.0.1", 7894));
                System.out.println(sensor.retornaVa-
lor();
                }
}
```

Fonte: elaborado pelo autor.

Os padrões de projeto são ótimas ferramentas para resolver problemas recorrentes em um projeto orientado a objetos, e seu uso torna os sistemas mais robustos e com mais chances de manutenção e reutilização.

Pesquise mais

Algumas fontes possuem mais informações sobre padrões de projeto, por exemplo:

- **Refactoring Guru** disponível em: <http://bit.ly/2wkrIO2>. Acesso em: 21 ago. 2018.
- **Source Making** disponível em: <http://bit.ly/2OVzpSF>. Acesso em: 21 ago. 2018.

Sem medo de errar

Você trabalha em uma empresa especializada em migrar sistema legados, e seu projeto atual consiste em atualizar o sistema de uma universidade. Durante o desenvolvimento do projeto, detectou-se que diversos desenvolvedores estavam instanciando uma conexão com o sistema de gerenciamento de banco dados para cada classe ou módulo do sistema. Isso está causando um grande problema no sistema, por utilizar muita memória, e no servidor que, em alguns momentos, fica indisponível pelo grande volume de conexões abertas. Para resolver a questão, deve-se encontrar um padrão de projeto para essa situação e mostrar o código-fonte.

Esse problema pode ser solucionado utilizando o *Singleton*, que provê uma forma de garantir apenas uma instância para todo o sistema. No Quadro 3.16 temos uma implementação da conexão do sistema de gerenciamento com o banco de dados utilizando o *Java Database Connectivity* (JDBC) e o *Singleton*. Veja que o construtor da classe é `private`, logo, somente o método `getInstanciaBD()` pode instanciá-la.

Quadro 3.16 | Conexão utilizando JDBC e *Singleton*

```
package U3S2;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;

public class SingletonConexaoBancoDeDados {

     private static SingletonConexaoBancoDeDados conexao;

     private Connection conJDBC;
     private Statement st;
     private final String URLDB = "jdbc:mysql://localhost:3306/bd";
     private final String usuario = "user";
     private final String senha = "senha";

     private SingletonConexaoBancoDeDados()
     {
          try {
               Class.forName("com.mysql.cj.jdbc.Driver");
               conJDBC = DriverManager.getConnection(URLDB, usuario, senha);
               st = conJDBC.createStatement();

          } catch (Exception e) {
```

```
                e.printStackTrace();
            }
        }
        public Statement getStatement()
        {
            return st;
        }
        public Connection getConexao()
        {
            return conJDBC;
        }

        public static SingletonConexaoBancoDeDados getIn-
stanciaBD()
        {
            if(conexao == null)
            {
                conexao = new SingletonConexaoBancoDe-
Dados();
            }
            return conexao;
        }
}
```

Fonte: elaborado pelo autor.

Utilizando a classe que possui o *Singleton*, a construção fica restrita apenas à própria classe. Dessa forma, é possível controlar como e quando essa conexão será feita.

Avançando na prática

Controle de acesso para estacionamento

Descrição da situação-problema

Você trabalha em uma empresa que desenvolve sistemas de controle de acesso físico a ambientes e sua equipe trabalha na parte de modelagem dos sistemas para que a área de desenvolvimento

faça as implementações. A etapa atual do projeto está em modelar a parte de leitura de um cartão com código de barras e a verificação da permissão, dados o código e o acionamento da cancela para a passagem do carro. Seu líder comentou que quer encontrar uma forma de garantir que os desenvolvedores façam essas operações em três métodos diferentes, que sejam sempre executados nessa sequência e ainda que, em casos de protocolos de acesso diferentes para cada tipo de leitor ou atuador, a chamada dessa ação seja a mesma. Sua tarefa consiste em apresentar um padrão de projeto que tenha essa característica e apresentar a classe base para que seja feita a implementação pelos desenvolvedores.

Resolução da situação-problema

O padrão de projeto a ser utilizado nesse cenário é o *Template Method*, com o qual é possível descrever um esqueleto para que sejam definidos os passos e sua sequência. No Quadro 3.17 está a codificação dessa sequência que os desenvolvedores serão obrigados a seguir.

Quadro 3.17 | Utilização do padrão *Template Method* para sistema de controle de acesso

```
package U3S2;

public abstract class TemplateMethodControleAcesso {

    public boolean controlaAcesso()
    {
        String numero = lerCodigodeBarras();
        boolean acessoStatus = acesso(numero);
        if(acessoStatus == true)
        {
            acionaCancela();
            return true;
        }else {
            return false;
        }
    }
```

```
    protected abstract String lerCodigodeBarras();
    protected abstract boolean acesso(String numero);
    protected abstract boolean acionaCancela();

}
```

Fonte: elaborado pelo autor.

Faça valer a pena

1. Os padrões de projeto são amplamente utilizados em sistemas de todos os portes, e sua aplicação depende do conhecimento das pessoas que estão fazendo a implementação e a modelagem do sistema. Com isso, é possível aumentar as chances de reutilização das classes e dos módulos.

Qual é o objetivo dos padrões de projeto?
a) Apresentar o uso de classes e interfaces.
b) Mostrar a informações das classes de forma clara e direta.
c) Descrever como declarar os atributos de uma classe.
d) Fornecer soluções para problemas recorrentes em sistemas orientados a objetos.
e) Fornecer soluções para problemas incomuns em sistemas orientados a objetos.

2. Dentro de diversos padrões de projeto, podemos citar o *Singleton* como uma forma de garantir que certa instância de uma classe seja única em todo o sistema. Ele permite o uso de recursos como conexões a dispositivos externos ou outros elementos que necessitam de apenas uma instância.

Quando se utiliza o *Singleton*, qual é a função do construtor privado?
a) Garantir que não seja possível construir o objeto de fora da classe.
b) Permitir que a classe seja vista por todo o sistema.
c) Gerar elementos de controle da classe.
d) Permitir que o objeto seja construído fora da classe.
e) Bloquear o acesso das threads.

3. O padrão *Template Method* tem o objetivo de fornecer uma maneira padronizada de utilização de um recurso. Com ele é possível definir quais

passos e qual sequência de passos as classes devem seguir, através dos mecanismos da orientação a objeto que permitem a reescrita de métodos.

Análise as afirmações e escolha a opção correta.
I – Uma classe abstrata é usada para definir padrões na utilização de métodos.
II – Na classe que especializa uma abstrata, somente os métodos que serão usados precisam ser sobrescritos.
III – No padrão *Template Method*, as classes abstratas são usadas para especificar os métodos, que descrevem os passos do processo a ser mapeado.
a) Somente a afirmação I está correta.
b) Somente a afirmação II está correta.
c) Somente a afirmação III está correta.
d) Somente as afirmações I e II estão corretas.
e) Somente as afirmações I e III estão corretas.

Seção 3.3

Métodos ágeis em orientação a objetos

Diálogo aberto

O desenvolvimento de software pode ser, didaticamente, comparado ao pedido de um sanduíche: você precisa especificar o tipo e o formato do pão, o recheio, o molho e as outras opções, e seu referencial pode ser a foto que está na frente da lanchonete. Todavia, depois do pedido feito e da execução iniciada, é complicado fazer alterações, você não pode acompanhar todas as etapas e, em diversas vezes, o resultado não é o esperado. No desenvolvimento de software temos cenários parecidos, o cliente pede um sistema com um conjunto de requisitos e, ao final, recebe algo que não esperava. Como podemos fazer para que esse processo seja mais preciso e o cliente receba aquilo que precisa/requisitou?

Você trabalha em uma empresa especializada na migração de sistema de legados e está na equipe de desenvolvimento de *backend*. Seu grupo está tendo dificuldades com as entregas do software, pois em alguns casos o usuário reclama que o tempo de desenvolvimento é muito grande e o que é entregue não é importante para o sistema. Em outras situações, o usuário comenta que as entregas não condizem com o que é necessário para resolver os problemas apresentados. Isso está se tornando recorrente e o gerente do projeto está pensando em contratar mais desenvolvedores para resolver a questão. A fim de tentar organizar isso, seu gerente fez uma tabela que apresenta os requisitos definidos pelo usuário e suas prioridades, como no Quadro 3.18.

Quadro 3.18 | Requisitos do usuário

Requisito	Prioridade
O sistema novo deve receber as notas dos alunos do sistema antigo, sendo organizadas nas novas tabelas.	Alta
O sistema novo deve gerar relatório com as médias de cada turma e apenas o coordenador deve ter acesso a esse recurso.	Média

Requisito	Prioridade
O sistema deve gerar um relatório com nome e registro dos alunos para ser feita a chamada em cada aula.	Baixa

Fonte: elaborado pelo autor.

Você, como parte da equipe, deve apresentar uma solução para os problemas utilizando métodos ágeis de desenvolvimento de software. Para isso, faça uma tabela que apresente o problema formatado em história de usuários, organizando as tarefas em um *sprint*. Nessa etapa de estudos vamos conhecer a origem dos métodos ágeis e uma forma de aplicação desses métodos chamado *scrum*, em que será possível dividir e organizar as tarefas, de forma a priorizar de acordo com o que o usuário necessita. Dessa maneira, vamos produzir software com foco no que podemos entregar para o cliente. Bons estudos!

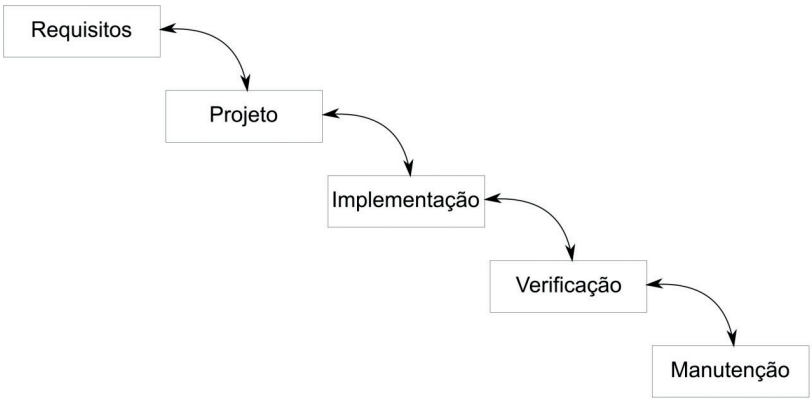
Não pode faltar

O desenvolvimento de software é essencial para todas as atividades, e podemos encontrar exemplos desta afirmação em todas as áreas, basta olharmos ao redor. Durante o início da computação, o software não recebia a sua devida importância, e o hardware recebia todo o crédito pelas aplicações computacionais. Contudo, esse contexto mudou rapidamente, devido à popularidade das soluções que o computador provia. A maneira de desenvolvimento de software seguia uma metodologia semelhante aos projetos de hardware, com formas rígidas de desenvolvimento, ignorando as necessidades de diversas partes. O modelo de cascata foi a primeira maneira de desenvolvimento e é considerado o modelo direto para o desenvolvimento de software. A Figura 3.8 apresenta as etapas clássicas de desenvolvimento de software nesse modelo (SOMMERVILLE, 2011):

1. **Requisitos:** levantar, entender e desenvolver as necessidades que levam à criação do software.
2. **Projeto:** elaborar os diagramas de *Unified Modeling Language* (UML - linguagem de modelagem unificada) e escolher as arquiteturas de software para o projeto.
3. **Implementação:** desenvolver o software utilizando os requisitos e o projeto feitos nas etapas anteriores.

4. **Verificação:** executar os testes no software e fazer a instalação.
5. **Manutenção:** verificar e corrigir problemas que foram detectados durante a operação do software.

Figura 3.8 | Modelo clássico cascata de desenvolvimento de software



Fonte: elaborada pelo autor.

O modelo em cascata apresenta todas as etapas necessárias para que um software possa ser desenvolvido. Nesse cenário, se todas as etapas forem seguidas, o software estará pronto de acordo com o que foi pedido pelo cliente e com todo o planejamento elaborado pela etapa de projeto. Todavia, no decorrer do tempo os softwares podem apresentar diversos problemas, tais como (STELLMAN, 2015):

1. As etapas de requisitos e de projeto tomam muito tempo, comparadas ao projeto completo, assim, depois de certo tempo, apenas a documentação poderia ser entregue ao cliente, e não o software.
2. O software entregue, em diversos casos, não atende às necessidades do cliente, mesmo ele tenha as detalhado.
3. Durante o desenvolvimento, o projeto muda devido a dificuldades ou limitações, levando a uma implementação diferente do projeto.
4. Como o tempo de desenvolvimento pode ser grande, as etapas anteriores não geram entregáveis. Além disso, a etapa de testes pode ser negligenciada, validando os requisitos levantados no início do projeto, que podem não coincidir com o estado atual do desenvolvimento do sistema.

Devido às limitações que o modelo em cascata apresenta, surgiram outras formas de desenvolvimento de software. Esses modelos são caracterizados pela sua visão incremental dos estágios do modelo canônico cascata. Com isso, as etapas são feitas diversas vezes no projeto, até que o sistema esteja pronto. Dois modelos relacionados a esse tipo de abordagem são o em espiral e o de protótipo, que evitam diversos problemas que o modelo em cascata apresenta, tais como a entrega tardia do projeto e a distância entre requisitos e projeto com o software. Todavia, as próprias metodologias apresentam limitações em um aspecto essencial que os programas de computadores possuem, pois a maioria dessas aplicações é utilizada por pessoas que não são aquelas que fazem o pedido ou que estão envolvidas no processo. Ao final, mesmo com todo o aspecto incremental, as entregas continuam a enviar para o cliente algo que ele não esperava ou, ainda, algo em uma ordem não relacionada à importância para o cliente.

Diante desse cenário de diversas inconsistências, com o objetivo de encaminhar as formas de resolução para as questões de requisitos, prioridade de entrega, mudanças no projeto e outros, em 2001 foi elaborado um manifesto com 4 valores que devem ser levados em consideração durante o desenvolvimento de software (COHN, 2015):

1. Indivíduos e interações mais que processos e ferramentas.
2. Software em funcionamento mais que documentação abrangente.
3. Colaboração com o cliente mais que negociação de contratos.
4. Responder a mudanças mais que seguir um plano.

Esses quatro valores são permeados por 12 princípios que os detalham, reforçando que as necessidades do cliente têm prioridade sob a metodologia. Nesse processo, as pessoas que fazem parte do uso, gerência, pedido e outros devem fazer parte do desenvolvimento, e as entregas devem ser feitas de forma constante e por prioridade que o cliente definir, mantendo a simplicidade do sistema.

Assimile

Os métodos ágeis possuem características semelhantes das metodologias incrementais antigas. Todavia, foram incluídos diversos elementos de interação e necessidades do usuário/cliente que tornaram as metodologias ágeis populares.

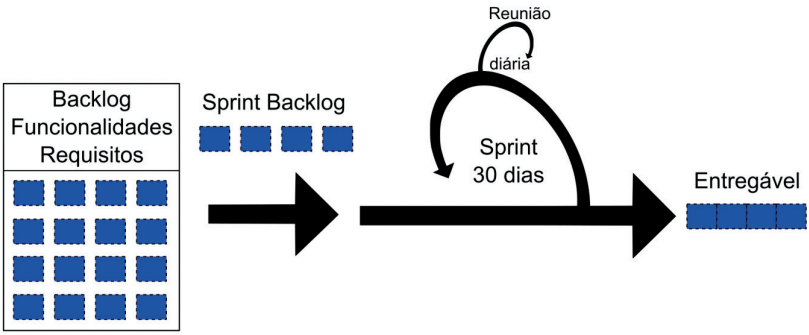
O *scrum* é uma das diversas metodologias que usam conceitos ágeis, em que são definidos 3 papéis (STELLMAN, 2015):

1. *Product owner* (PO):
    a. Pessoa responsável pelo produto a ser entregue.
    b. Determina o que será desenvolvido e em qual ordem.
    c. Faz a comunicação entre o time de desenvolvimento e as outras entidades para garantir a correta visão do que está sendo desenvolvido.
2. *Scrum master*:
    a. Possui familiaridade pela etapa atual de desenvolvimento podendo ajudar toda equipe.
    b. Ajuda a manter os princípios, valores e papéis do *scrum*.
    c. Protege o time de desenvolvimento de interferências externas.
    d. Resolve conflitos e problemas técnicos e pessoais para garantir a produtividade.
    e. Não tem papel de gerência externa.
3. Time de desenvolvimento:
    a. Grupo de pessoas que se auto-organiza para atingir o objetivo proposto pelo PO.
    b. Com 5 a 9 pessoas, coletivamente possui todas as habilidades necessárias para realizar o projeto.

A Figura 3.9 apresenta a visão geral da metodologia *scrum*, em que o primeiro elemento a ser analisado é o *backlog*, que consiste em um conjunto de requisitos/funcionalidades elencadas pelo *project onwer,* junto com a equipe de desenvolvimento e o *scrum master*. A equipe de desenvolvimento pode opinar em relação aos requisitos, mas o PO tem prioridade no processo. O *sprint backlog* consiste em um conjunto de funcionalidades vindas do *backlog*, selecionadas

pelo PO (com certa orientação técnica do *scrum master* e do time de desenvolvimento). Tais tarefas devem ser realizadas e entregues em certo período de tempo, chamado de *sprint*. Ao final de cada *sprint* é gerado um empregável para o cliente. Todos os dias são feitas reuniões com a equipe de desenvolvimento (*daily meeting*) para entender como está o desenvolvimento da *sprint*.

Figura 3.9 | Visão geral das etapas do *scrum*



Fonte: elaborada pelo autor.

Reflita

A criação do *backlog* é responsabilidade do PO, que deve levantar e entender as histórias do usuário. Porém, como se deve mensurar o tempo de cada tarefa? Será que o time de desenvolvimento acertará o *sprint* em 30 dias todos os meses?

Para criar o *backlog* são utilizadas histórias de usuários trazidas pelo PO, que contêm todas as necessidades do software. As histórias devem ter um formato específico, para aumentar as chances de o desenvolvimento ser correto:

Como usuário <*tipo de usuário*>, eu preciso <*objetivo*>, pois <*necessidade*>

O valor <*tipo de usuário*> deve explicar qual usuário fará a ação, por exemplo, o usuário padrão, administrador ou outros. O parâmetro <*objetivo*> descreve o que deve ser feito, contendo as suas necessidades e justificativas para implementação. O elemento que explica a razão auxilia no entendimento das necessidades daquela funcionalidade e em que momento deve ser priorizada.

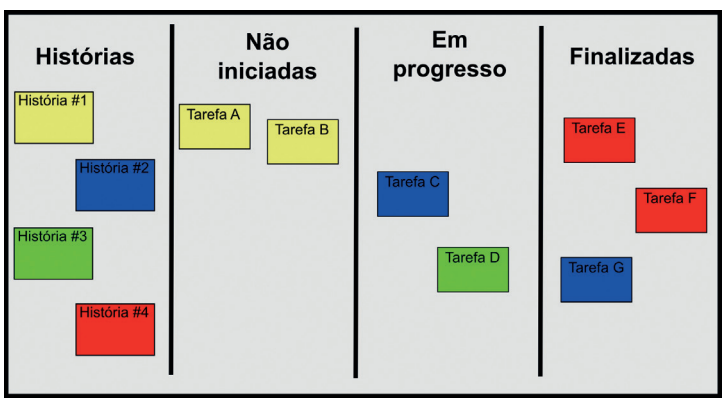
Exemplificando

Podemos citar alguns exemplos de histórias de usuários:

- Como usuário padrão, eu preciso acessar o sistema utilizando a minha impressão digital, pois ele possui diversas informações confidenciais.
- Como usuário administrador, eu preciso verificar quem conseguiu acesso ao sistema, pois ele possui diversas informações confidenciais.

Essas histórias são transformadas em tarefas, e o processo consiste em dividir as histórias em tarefas que possuem um nome e uma estimativa de tempo. Esse processo é muito importante, pois o tempo que se espera para concluir uma tarefa é utilizado para planejar o *sprint*. Essas tarefas levam em conta a criação das classes, pois estas serão criadas a partir dos substantivos das frases. Além disso, os atributos das classes poderão ser identificados pelos adjetivos e o relacionamento entre classes através dos verbos (DEITEL; DEITEL, 2016). A Figura 3.10 apresenta uma forma de organizar as tarefas. Pode ser utilizado um quadro branco ou até uma parede com blocos de papel adesivo para organizar as histórias de usuários, as tarefas iniciadas, as que estão em progresso e as que já foram finalizadas. Todas elas devem levar os conceitos da orientação a objetos e todas as formas corretas de planejamento.

Figura 3.10 | *Scrum board* para controle das tarefas e histórias



Fonte: elaborada pelo autor.

Exemplificando

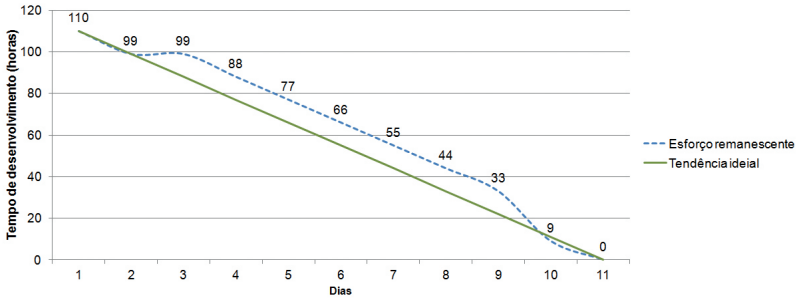
Além das histórias de usuários, o *backlog* pode ter outros recursos, tais como tarefas técnicas ou elementos de controle:

- Popular a base de dados.
- Verificar o sistema de versão do time de desenvolvimento.
- Liberar uma versão específica.

A atualização do quadro com as histórias e as tarefas que estão sendo feitas pode ocorrer tanto no momento em que chega nova demanda, quanto durante a *sprint* de desenvolvimento. Todos os dias, na reunião diária, as tarefas são atualizadas e todos informam o que foi feito. Além do quadro, é feito um gráfico *burndown*, que apresenta uma relação entre os dias da *sprint* e o esforço em horas da soma das tarefas, sendo *T* o tempo total de cada tarefa *N* e *F* o tempo utilizado para a tarefa *N* que foi efetivo no seu desenvolvimento. Isso mostra o tempo total ainda necessário para terminar a *sprint*.

A Figura 3.11 apresenta um exemplo desse gráfico, em que temos uma *sprint* de 10 dias, com uma estimativa de 110 horas de trabalho. Nos primeiros dois dias é possível reparar que o desenvolvimento estava indo como o planejado, pois a linha tracejada coincide com a linha solida, ou seja, a quantidade de horas planejadas dividida pelo número de dias da *sprint* está de acordo com o valor informado pela equipe do tempo que foi gasto nas etapas. Porém, no dia 3 o tempo reportado foi mais baixo, deixando horas a serem realizadas nos outros dias. Quase ao final da *sprint* foi possível realizar as tarefas atingindo o tempo necessário para terminar o desenvolvimento no prazo. Durante a *sprint*, o comprometimento é sempre com a entrega do que o PO apresentou como prioridade. Todavia, no *backlog* os desenvolvedores podem incluir elementos de documentação ou testes para garantir que as etapas consideradas sensíveis sejam feitas de maneira precisa.

Figura 3.11 | Exemplo do gráfico *burndown*



Fonte: elaborada pelo autor.

As metodologias ágeis trouxeram fatores mais humanos para que o produto final seja realmente o que o cliente espera, tornando o desenvolvimento mais exato, assim, causando menos retrabalhos e menos atritos.

Pesquise mais

Existem diversos materiais que podem ser utilizados para buscar mais informações sobre métodos ágeis e *scrum*:

- Manifesto para desenvolvimento ágil de software, disponível em: <http://agilemanifesto.org/>. Acesso em: 22 ago. 2018.
- SOMMERVILLE, Ian. Engenharia de software. 9. ed. São Paulo: Pearson Prentice Hall, 2011. 544 p.
- MBA NA PRÁTICA. **Projetos ágeis e tradicionais:** qual é a diferença? 16 fev. 2016. Disponível em: <http://bit.ly/2N9Rj3H>. Acesso em: 22 ago. 2018.

Mesmo com todos os aspectos incrementais, humanos e interativos que os métodos ágeis possuem, é sempre necessária a correção de problemas que um código-fonte pode ter. O sistema em seu início tem uma estrutura orientada a objetos, e os métodos são bem planejados para ter as características de coesão e acoplamento. Cada correção arruma os problemas, porém, tende por alterar a estrutura do software (COHN, 2015). Para resolver essa questão ou melhorar um código que foi mal elaborado, existe a refatoração. Esta etapa consiste em alterar a estrutura de um código sem modificar o seu comportamento, o que implica que será

necessário utilizar tempo no *sprint* para alterar o código que já foi implementado e está funcionando.

Reflita

Se a refatoração consiste em alterar um código que foi implementado, essa etapa pode ser considerada retrabalho? Como podemos evitar isso? Será que técnicas como *pair programming* (quando dois programadores fazem o mesmo código ao mesmo tempo) podem ajudar a evitar o retrabalho?

As razões que levam a essa demanda são diversas. Podemos citar alguns cenários que podem ser os sinais de que é necessário utilizar tempo para refatorar o código (MESZAROS, 2007):

- Nomes de variáveis, métodos ou atributos sem significado para o sistema.
- Partes do código duplicadas.
- Códigos extensos que poderiam ser reaproveitados de bibliotecas ou de outros projetos.
- Porções de códigos que não passaram nos testes (nesse caso, os testes unitários ganham destaque para forçar uma refatoração).
- O tempo gasto em correções dos bugs ser alto comparado ao tempo de desenvolvimento.
- Os programadores, quando têm que corrigir um problema em módulos de autoria de outras pessoas, utilizam muito mais tempo que nos de própria autoria.

Todos esses sintomas devem ser avaliados durante as etapas de desenvolvimento para desencadear os processos de refatoração. Existem momentos indicados para que seja feito o processo de correção do código, que podem ser:

- Quando se adiciona uma nova característica ao software, quando é possível ver quais partes são utilizadas e, então, corrigir os problemas.
- No próprio processo de correção de bugs pode-se averiguar as condições do código e aplicar as correções.
- Em casos mais complexos, pode ser necessário fazer uma

revisão de código e aplicar a refatoração, dessa forma, tenta-se minimizar o tempo de correção e entendimento do código.

Os processos de refatoração podem ser descritos como uma revisão e melhor entendimento de cada parte do código. Há metodologias precisas para descrever como um código deve ser refeito para se tornar melhor. Aqui abordaremos as formas mais gerais:

- Métodos devem ser claros, e a sua função precisa ser muito bem descrita. A função de um método deve ser uma tarefa direta e simples.
- As classes necessitam sempre buscar a alta coesão e o baixo acoplamento, ou seja, fazer ações específicas de maneira independente.
- Durante a refatoração não se deve incluir novas funções, o foco deve ser deixar o código mais legível e preciso.
- Todos os testes ao final da refatoração devem ter sucesso, e essa é uma medida de qualidade essencial em projetos.

A refatoração consiste em observar o código e avaliar se é necessário melhorar a base já implementada para que o projeto tenha uma duração longa. Levando em conta os recursos vistos nessa seção, você aumenta as chances de sucesso do seu projeto!

Sem medo de errar

Você trabalha em uma empresa especializada em migração de sistema legado, e o projeto atual consiste na criação de um novo sistema para uma universidade. Todavia, o cliente está reclamando que as entregas demoram muito para serem feitas e, em diversos casos, o que é entregue não é o esperado. O seu líder destacou você para encontrar uma saída para resolver esse problema. Portanto, você deve implementar o *scrum* para resolver essas questões e, para isso, é necessário:

1. Explicar a metodologia para o cliente, a fim de que ele entenda como será o processo de entregas.
2. Definir o tempo de cada *sprint* junto com a equipe e com o cliente, assim, é possível ajustar as expectativas.
3. Dividir as tarefas em histórias de usuário e alinhar com o cliente quais são as de maior prioridade.

4. Dentre a equipe, escolher o *scrum master* que tem mais afinidade com o método e com essa etapa de desenvolvimento.
5. Criar o *scrum board* e preparar o gráfico *burndown*.

Como primeiro passo, deve ser considerado o Quadro 3.19, que apresenta as tarefas em formato de história de usuário com a prioridade. Assim, é possível dividir em tarefas menores e organizar a *sprint*.

Quadro 3.19 | Requisitos no formato de histórias de usuário

ID	História de usuário	Prioridade
#1	Como usuário administrador, eu preciso de uma rotina para migrar as notas do sistema antigo para o novo, pois o antigo será desligado em breve.	Alta
#2	Como coordenador, eu preciso gerar um relatório com a média de todas as turmas do semestre, pois é importante acompanhar o progresso dos alunos.	Média
#3	Como professor, eu preciso gerar um relatório com todos os alunos, pois é necessário verificar a presença deles durante as aulas.	Baixa

Fonte: elaborado pelo autor.

Como está na equipe de *backend*, você não deve produzir as interfaces gráficas, e sim as classes para fazer o processamento. O Quadro 3.20 apresenta essa divisão.

Quadro 3.20 | Tarefas derivadas das histórias de usuário

ID	Tarefa	Prioridade	Tempo em horas
A	Criação da classe de importação de notas.	Alta	20
B	Criação da tabela para notas e alunos.	Alta	10
C	Criação da classe para gerar relatório de média das turmas.	Média	10
D	Criação de classe para relatório de alunos.	Baixa	10

Fonte: elaborado pelo autor.

Observando o Quadro 3.20, pode-se consultar quantos membros a equipe tem e mensurar a *sprint*. Além disso, é possível deixar claro para o usuário o que e quando certos módulos do sistema serão entregues.

Avançando na prática

Mensurar desenvolvimento de sistema de emissão de notas fiscais

Descrição da situação-problema

Você está trabalhando em uma empresa produtora de software, cujo principal produto é um planejamento de recursos empresariais (ERP, do inglês *enterprise resource planning*). Sua equipe é responsável pela produção da interface gráfica utilizada no sistema e está iniciando a utilização de *scrum*. A lista de tarefas já foi efetuada e está no Quadro 3.21.

Quadro 3.21 | Lista de tarefas já efetuadas

ID	Tarefa	Prioridade	Tempo em horas
A	Criação da interface de cadastro de produto	Alta	25
B	Correção da interface de cadastro de cliente.	Alta	10
C	Criação da classe para envio de informações de produto	Média	10

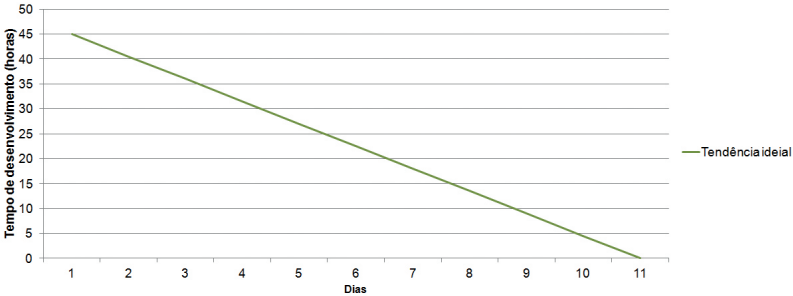
Fonte: elaborado pelo autor.

Inicialmente, a *sprint* será feita com as tarefas e, em seguida, a criação do gráfico *burndown*. Para isso, utilizando uma planilha eletrônica, gere um gráfico em que no eixo X tenha os 10 dias da *sprint* e, no eixo Y, o tempo de desenvolvimento. Deixe-o pronto para que, durante o progresso da *sprint*, os desenvolvedores possam atualizar o tempo que falta para cada tarefa.

Resolução da situação-problema

A Figura 3.12 apresenta o gráfico com a tendência ideal para o desenvolvimento da *sprint*. Com ele é possível vislumbrar o modelo ideal, considerando a previsão do tempo necessário de cada tarefa a ser feita no período de 10 dias.

Figura 3.12 | Gráfico *burndown* a ser utilizado na *sprint*



Fonte: elaborado pelo autor.

Faça valer a pena

1. O desenvolvimento de software sofreu grandes mudanças no decorrer do tempo. No princípio, cada etapa era muito bem definida, levando à geração da documentação completa antes da produção de código e outros efeitos. Com isso, todo o processo de desenvolvimento transcorria sem a interação com o cliente, criando, assim, casos em que a entrega final não era o que cliente esperava.

Observe as afirmações abaixo:
I - Indivíduos e interações mais que processos e ferramentas.
II - Software em funcionamento mais que documentação abrangente.
III - Colaboração com o cliente mais que negociação de contratos.
IV - Responder a mudanças mais que seguir um plano.
V - Elementos de programação mais que processos de controle.

Quais delas pertencem aos valores do manifesto ágil?
a) Apenas as afirmações I, II e V.
b) Apenas as afirmações I, II, III e V.
c) Apenas as afirmações I, II, III e IV.
d) Apenas as afirmações II e III.
e) As afirmações I, II, III, IV e V.

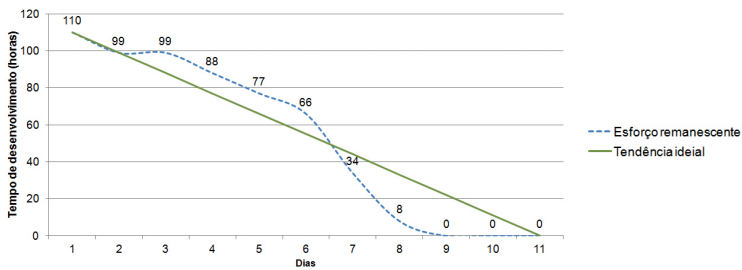
2. Para o levantamento de requisitos, há diversas formas de buscar o entendimento do que o cliente procura no software solicitado. Uma delas é a história de usuários, que são pequenos textos que refletem o que deve ser feito, por quem e qual é a razão.

Durante o desenvolvimento de um software utilizando o *scrum*, onde são aplicadas as histórias de usuário?
a) As histórias de usuário são aplicadas durante a *sprint* para verificar qual é o modelo de teste necessário.
b) As histórias de usuário são usadas apenas na etapa de planejamento.
c) As histórias de usuário são utilizadas apenas para verificar se a *sprint* deve ter mais ou menos que 30 dias.
d) As histórias de usuário são utilizadas para montar o *product backlog* e, com isso, criar as tarefas da *sprint*.
e) As histórias têm papel figurativo, servindo apenas para ilustrar um cenário já consolidado.

3. Para controle da *sprint* se utiliza o gráfico *burndown*, com o qual é possível verificar se o tempo estimado de cada tarefa está sendo bem mensurado e como está o desenvolvimento daquela etapa do software. No eixo X temos os dias da *sprint* e no eixo Y temos o tempo utilizado/estimado das tarefas.

O que significa quando a linha de esforço remanescente chega a 0 antes do final da *sprint,* como na Figura 3.13?

Figura 3.13 | Gráfico *burndown* utilizando no *scrum*



Fonte: elaborada pelo autor.

a) O tempo de desenvolvimento está muito lento comparado ao tempo das tarefas da *sprint*.
b) O tempo de desenvolvimento está adequado comparado ao tempo das tarefas da *sprint*.
c) A equipe está fazendo a tarefa de forma muita rápida, o que reflete uma estimativa errada do tempo de cada tarefa.
d) A equipe está fazendo a tarefa no tempo correto.
e) Isso prova que o desenvolvimento ágil é uma forma ótima para a produção de software.

Referências

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. 1. ed. New York: Cambridge University Press, 2008. 346 p.

CHACON, S.; STRAUB, B. **Pro git**. 2. ed. New York: Apress, 2018. 515 p.

COHN, M. **Desenvolvimento de software com scrum**. 1 ed. Porto Alegre: Bookman, 2011. 471 p.

DEITEL, P.; DEITEL, H. **Java - como programar**. 10. ed. São Paulo: Pearson, 2016. 968 p.

FREEMAN, E.; FREEMAN, E. **Use a cabeça! Padrões de projetos**. 2. ed. Rio de Janeiro: Alta Books, 2007. 496 p.

FURGERI, S. **Java 8 - Ensino Didático - Desenvolvimento e Implementação de Aplicações**. São Paulo: Érica, 2015. 320 p.

GAMMA, E. et al. **Padrões de projeto**: Soluções reutilizáveis de software orientado a objetos. 1 ed. Porto Alegre: Bookman, 2000. 370 p.

HORSTMANN, C. S. **Core Java Volume I - Fundamentals**. 10. ed. New York: Prentice Hall, 2016. 1040 p.

MANZANO, J. A. G.; COSTA JUNIOR, R. **Programação de Computadores com Java**. São Paulo: Érica, 2014. 160 p.

MESZAROS, G. **XUnit Test Patterns: Refactoring Test Code.** Boston: Addison Wesley, 2007. 944 p.

SOMMERVILLE, I. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011. 544 p.

STELLMAN, Andrew; GREENE, Jennifer. **Learning agile**. 1 ed. California: O'Reilly Media, 2015. 419 p.

TAHCHIEV, P. et al. **Junit in action, second edition**. 2. ed. New York: Manning Publications, 2010. 504 p.

Novas tecnologias para programação em banco de dados

Convite ao estudo

Ao comparar a computação com a matemática ou física, é notável que a primeira é uma ciência muito mais nova que as demais e, mesmo assim, já passou por diversos avanços e mudanças de paradigma em todas as suas áreas. Os sistemas de gerenciamento de banco de dados (SGBD) também receberam avanços em suas implementações; podemos comparar esse processo com o da construção civil. No passado, as pontes, viadutos e casas tinham estruturas muito mais robustas comparadas com as que temos atualmente e, em alguns casos, a quantidade de material utilizado era exagerada. Esse excesso de uso de material era fruto da falta de informação, ou ainda problemas na padronização na produção da matéria-prima. No caso dos SGDB o cenário é semelhante. No início, os sistemas eram robustos e tentavam solucionar todos os problemas de apenas uma maneira, todavia, conceitos antigos empregados com foco em novas tecnologias trouxeram o modelo NoSQL.

O objetivo dessa parte do nosso estudo é conhecer e compreender o uso de banco de dados não relacional em um ambiente orientado a objetos, habilitando você a construir aplicações Java com banco de dados não relacional. Essa visão é de extrema importância, pois o NoSQL é um tema que está sendo abordado de forma recorrente nas empresas e nas publicações científicas, sendo algo utilizado por diversas companhias em grandes projetos, e ainda muitos pesquisadores estão buscando os caminhos mais viáveis para o uso correto.

Você está se saindo muito bem na equipe de *backend* da empresa especializada na migração de sistemas legados.

Seus avanços em programação foram notados e agora seu papel está na busca de novas tecnologias para utilizar a grande quantidade de informação que o projeto de migração do sistema de uma universidade possui. A utilização desse volume de informação de maneira correta necessita de uma visão mais precisa sob a forma de utilizar os dados. Para ter sucesso nessa tarefa, será necessário utilizar as novas formas de armazenamento NoSQL. Com isso, será possível adequar a informação em estruturas e formas de acesso mais indicadas para obter o máximo de desempenho possível. Como que é feita a estrutura de um banco de dados NoSQL? Como escolher o tipo correto de um NoSQL para cada aplicação? Como fazemos as operações de escrita, alteração e remoção sem usar o SQL?

Na primeira etapa do seu estudo em novas tecnologias para programação em banco de dados, seu foco será entender essa nova tecnologia e estudar os diversos tipos e formatos do NoSQL. A segunda etapa tratará em como instalar, configurar e conectar em um sistema NoSQL muito utilizado, chamado MongoDB. Por fim, você verá como fazer as operações de inserção, seleção, atualização e remoção de dados utilizando um sistema MongoDB.

Seção 4.1

Banco de dados NoSQL

Diálogo aberto

O armazenamento de dados estruturados é um dos pilares da computação. Desde o início do uso mais abrangente da computação, os bancos de dados possuem papel importante para garantir a persistência e disponibilidade dos dados. Todavia, o dinamismo necessário na modelagem dos dados, o grande volume de informação disponível (*big data*) e acesso a informação fez com que o modelo clássico de banco de dados fosse renovado. Agora, é necessário ter mais dados de maneira menos estruturada. Isso é possível com os sistemas de banco de dados atuais? Ao invés de um sistema centralizado e rígido, atualmente faz-se necessário criar sistemas de armazenamento de forma mais diversificada, que é conhecida como *not only SQL* (NoSQL). Os NoSQL são sistemas de banco de dados que possuem elementos mais relacionados ao desempenho, à modelagem dinâmica e à escalabilidade. Dessa forma, propicia formas mais precisas de tornar os sistemas mais ágeis a constante mudanças.

A empresa que você trabalha está desenvolvendo uma atualização de software legado, e você está alocado na equipe de *backend*. O projeto atual consiste em um sistema que conta com grande volume de dados, contendo nomes de alunos e notas, e esses dados foram migrados e um sistema muito antigo. Com tecnologias para *big data*, *machine learning* e *time series* se tornando cada vez mais comum, o seu líder técnico recebeu como pedido do cliente a tarefa de utilizar essas técnicas, derivadas da inteligência artificial, para encontrar quais seriam os indicativos mais sutis para realizar uma intervenção preventiva para suprir certas necessidades dos alunos. Com isso, sua tarefa consiste em pesquisar e indicar, dentre os diversos modelos de NoSQL, qual deve ser utilizado e qual produto usa esse tipo de modelo. Para isso, você deve verificar nos diversos modelos de NoSQL:

1. Tipo de modelo de armazenamento;
2. Projetos e indicativos de uso;

3. Aplicação do modelo de dados do projeto no modelo de dados do NoSQL;
4. Exemplo de como seria o armazenamento no modelo NoSQL escolhido.

Nessa etapa dos estudos você será apresentado ao conceito de bancos de dados NoSQL, aprenderá quais são os principais bancos que implementam essa abordagem, bem como suas formas de organização. Ainda, serão abordadas as formas mais comuns de modelagem de um NoSQL, suas aplicações e quais são os projetos que utilizam cada modelo.

Não pode faltar

A computação é uma ciência nova no mundo comparado à física ou matemática. As formas de armazenamento, processamento e transmissão dos dados muda e avança a cada geração de computadores, softwares e métodos. Existem diversos marcos que culminaram em padrões que são utilizados até hoje, a própria arquitetura e organização dos computadores são referências a projetos antigos, e esse processo também ocorreu com os sistemas de gerenciamento de banco de dados (SGBD). Antes de existir uma forma padronizada de armazenamento, consulta, alteração e remoção, os dados eram armazenados em arquivos e as aplicações faziam acesso direto a eles, assim, as formas de acesso e controle de concorrência eram feitas diretamente pela aplicação. Nesse cenário, qualquer defeito poderia causar inconsistências e problemas com perda de informação (PLUGGE; MEMBREY e HAWKINS, 2010). Além disso, cada programador precisa aprender uma forma de fazer acesso aos dados, o que leva a aplicações mais especializadas e mais dependentes de profissionais específicos.

Em 1970 foram criadas as primeiras versões dos SGDB relacional (tabelas e suas relações com chaves) e da linguagem de consulta estruturada (SQL, do inglês *Structured Query Language*), iniciando o padrão de armazenamento de informação que vigora até hoje para as mais diversas aplicações (HEUSER, 2011). Todavia, com os avanços no hardware e aumento exponencial na computação, contando com celulares, notebooks, smartphones e diversos outros dispositivos que fazem parte do cotidiano, surgiu a necessidade de

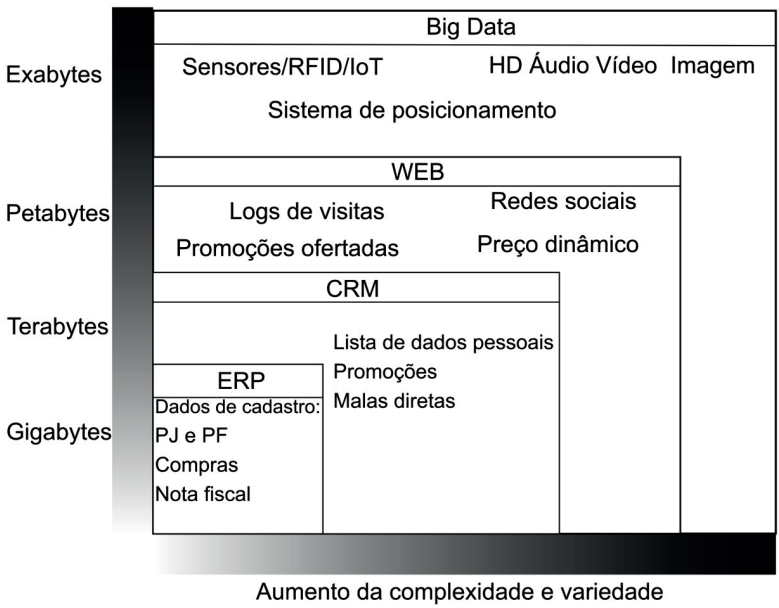
outra forma de controle dos dados. Hoje, existe muita informação disponível e com necessidade de ser processada, chegando a ser chamada atualmente de *big data* (HAN, E e LE, 2011).

Os métodos para tratar grande volume de informação têm como foco acessar e tratar relacionamentos não triviais nos dados, ou ainda, realmente entender o que está armazenado, pois os dados podem não estar em tabelas ou serem relacionados. Com isso, temos cenários de aumento constante no volume de informação e com um agravante sério a serem tratados; em diversos casos, esses dados não são estruturados (não possuem uma forma organizada de recuperar a informação) e ainda não são nem textuais, nem numéricos, são vídeos ou imagens. Além disso, caso exista uma estrutura, ela pode ser extremamente dinâmica, causando dificuldade em sua modelagem.

A Figura 4.1 apresenta uma visão, com exemplos, de como é a quantidade de informação relacionando a aplicação e sua complexidade de variedade. Olhando o primeiro quadrante da Figura (o quadro menor) temos o planejamento de recursos da empresa (EPR, do inglês *Enterprise Resource Manager*) contendo, em sua maioria, informações estruturadas e alocadas em tabelas com seus tipos de dados bem definidos e ainda todos os dados relacionados. Esse tipo de cenário consiste em um sistema no qual a entrada de dados é controlada de maneira a combinar com os tipos da tabela, sem liberdade ao usuário. A quantidade de dados em um sistema ERP é relacionada diretamente ao número de operações, todavia, como é textual, não ocupa tanto espaço como um conjunto de imagens que são aplicados no próximo cenário. No segundo quadrante, podemos alocar os dados de gestão de relacionamento com o cliente (CRM, do inglês *Customer Relationship Manager*), em que se utiliza a base do ERP, completada com outros dados, considerando diversos conjuntos de propagandas, documentos com folhetos, imagens e outros. Ainda no CRM, mesmo tendo mais dados que o ERP, os dados também são estruturados, sendo alocados de forma organizada pela interface com valores controlados e legíveis pelo computador. Ao avançar para o próximo quadrante (Web), já começamos a considerar cenários mais dinâmicos, nos quais o volume de informação cresce de forma muito mais rápida. Sendo aqui considerados imagens, vídeos, áudios e outros conjuntos de informação que já não são mais legíveis pelo computador, necessitando de uma análise para se entender sua

semântica. Unindo todos esses elementos com as informações de sensores, identificação por rádio frequência (RFID, do inglês *radio frequency identification*) e internet das coisas (IoT, do inglês *Internet of Things*), chegamos a um cenário onde temos muitas informações diversificadas. Com isso, a semântica necessária é aplicada por processos de inteligência artificial, aprendizado de máquina, séries temporais (*time series*) e outras formas de entendimento por máquina.

Figura 4.1 | Conjunto de dados e suas aplicações



Fonte: adaptada de Moniruzzaman e Hossain (2013, p. 2).

Unindo os argumentos já citados com outros itens, podemos resumir os diversos desafios de armazenamento de dados do período atual na lista:

1. Grande volume de informação;
2. Informações de diversos tipos, em forma estruturada e não estruturada;
3. A onipresença do software e a constância de mudanças;
4. Métodos ágeis de desenvolvimento, nos quais as mudanças de uma aplicação são bem-vindas;

5. Cenários em que a redução de custos é sempre um objetivo das empresas.

Dados todos os argumentos e cenários, podemos considerar que até hoje, os SGDB relacionais são suficientes para resolver esse tipo de problema. Todavia, existem diversos custos para que esse tipo de banco de dados cumpra todos os novos requisitos, como exemplo:

1. Custos de hardware para escalabilidade de volume de dados e desempenho;
2. A construção das tabelas de um SGDB relacional e suas consultas é ligada a aplicação;
3. Demanda de pessoal especializado para manter o desempenho do SGBD;
4. Em sistema orientado a objetos, é necessário criar tanto a modelagem das classes para serem utilizadas no sistema quanto o modelo de armazenamento.

Para atender essa demanda latente do novo cenário de volume de informação, em 2006 aconteceram diversos esforços vindos de empresas como Google e Amazon (LEAVITT, 2010). Tais empresas possuem grande quantidade de dados a serem tratados e, em diversos casos, essas informações necessitam ser divididas em vários computadores. Frente a esses movimentos foram criados/revisitados os conceitos de NoSQL. Esse "novo" modelo de armazenamento de informação foi proposto a partir da necessidade de sistemas de banco de dados relacionais mais leves, já sendo trabalhado desde 1998. O termo **NoSQL** se refere a "***Not Only SQL***", ou seja, não apenas SQL (PLUGGE, MEMBREY e HAWKINS, 2010). O nome faz analogia a não obrigatoriedade de usar uma linguagem de consulta estruturada (por exemplo, o SQL), pois o próprio modelo de armazenamento não é estruturado na forma de tabelas e colunas. De forma direta, os bancos de dados NoSQL tratam a informação como um formato em que a estrutura não precisa ser definida. Ainda, a forma de buscar a informação não obriga o programador a saber como os dados estão organizados, diferente do modelo relacional. Dessa forma, o desenvolvimento se torna mais dinâmico e mais adaptável para a as constantes mudanças que podem ocorrer durante o processo, deixando assim mais próximo o desenvolvimento das metodologias ágeis.

Porém, antes de analisarmos os conceitos de um banco de dados NoSQL, é importante deixar claro que ele não é a solução para todos os casos. Cenários mais clássicos, como instituições financeiras, sistemas legados e operações críticas em que a relação de diversas tabelas é necessária, o uso do NoSQL deve ser analisado com cautela. Uma das limitações dos sistemas NoSQL refere-se a propriedade de atomicidade das operações, lembrando que a atomicidade é uma característica do SGBD, na qual uma operação deve ser executada por completo ou falhar, por exemplo, a transferência de dinheiro entre contas bancárias. Com isso, a utilização de NoSQL em cenários em que esse tipo de operação é essencial deve ser analisada com cuidado.

Assimile

Sistemas de banco de dados NoSQL são indicados para os casos em que a estrutura dos dados não precisa ser definida, para que seja possível fazer as operações de consulta, alteração, inserção e exclusão. Com isso é possível obter outras vantagens, como a escalabilidade no aumento da quantidade de informação e o dinamismo do desenvolvimento.

O NoSQL possui um conjunto de características básicas que devem ser utilizadas para guiar a tomada de decisão no uso dessa nova tecnologia, tais como as descritas por HAN, E e LE (2011):

1. **Independente de esquema**: o armazenamento não possui uma organização rígida, levando a um conjunto de bibliotecas para acesso mais simples;
2. **Livre de junções de tabelas:** não são indicados os joins, mesmo que possíveis em alguns modelos, tornando a utilização menos complexa;
3. **Crescimento horizontal:** a maioria das soluções NoSQL são escaláveis, sendo anexado mais computadores e não necessariamente computadores mais caros;
4. **Hardware mais simples:** é possível utilizar mais hardware barato pela própria arquitetura da solução;

5. **Autocontido:** cada computador que funciona como armazenamento é independente, levando em diversos casos cópias das bases de dados;
6. **Controle de usuário:** necessita de poucos elementos de controle de usuário, transação e concorrência.

Reflita

A troca de qualquer tecnologia na produção de um software pode causar diversos custos, desde financeiros para a empresa até emocionais para os programadores. Na troca de um SGDB relacional para um NoSQL, como podemos mensurar esses custos?

Modelos de dados para os NoSQL

Como a estrutura de acesso a informação em um NoSQL é simples, torna-se possível variar a estrutura interna de controle. Existem muitos modelos para organizar a informação dentro de um NoSQL, todavia, existem quatro modelos mais comuns (GESSERT et al., 2016):

1. *Key-value*;
2. *Document stores*;
3. Orientado a colunas;
4. Grafo.

Como o NoSQL é uma tecnologia em crescimento e que vem conquistando seu espaço no mercado, é complicado afirmar qual é o melhor modelo. Existem muitas variáveis a serem consideradas, e cada projeto de NoSQL afirma que as suas formas de implementação são as melhores. Assim, para cada cenário é importante avaliar onde cada solução foi efetivada e seus resultados.

O sistema de *key-value* é uma estrutura mais simples, em que cada conjunto de dados possui apenas uma chave principal. O Quadro 4.1 apresenta um exemplo deste sistema, onde na coluna *Key* temos um número que identifica o registro e na coluna *value* existem os valores armazenados. Repare que não é necessário padronizar os dados, sendo que os valores da entrada 1 são diferentes da entrada 2.

Quadro 4.1 | Exemplo de organização no modelo *key-value*

Dados	
Key	***Value***
1	Aluno: João Curso: Ciência da Computação Conclusão: 2020
2	Aluno: Pedro Curso: Ciência da Computação Conclusão: 2022 Origem: Transferência interna

Fonte: elaborado pelo autor.

Um exemplo desse tipo de modelo é o Amazon SimpleDB. Esse tipo de organização permite as seguintes vantagens:

1. Flexibilidade na modelagem: os *Values* podem ser inseridos e alterados relacionado a cada registro;
2. Operação simples: é necessário apenas buscar a *Key* e já se tem acesso aos dados;
3. Escalabilidade e disponibilidade: pela sua simples estrutura, as técnicas computacionais para aumentar a capacidade e redundância são facilmente aplicadas.

Outro modelo muito utilizado é o *document stores*, em que se utiliza diversos arquivos formatados para fazer o armazenamento dos dados. Nesse modelo, em diversos casos, se utiliza o *Extensible Markup Language* (XML) ou o *JavaScript Object Notation* (JSON) para fazer a formatação dos dados que são armazenados no NoSQL. O Quadro 4.2 apresenta um exemplo de um JSON que pode ser utilizado por um banco de dados NoSQL. Observe que é possível criar diversas estruturas em cada registro, e ainda no próprio NoSQL você pode utilizar diversos arquivos diferentes para guardar os dados.

Quadro 4.2 | Armazenamento de dados utilizando o JSON em uma estrutura do MongoDB

```
{
    '_id' : 1,
    'name' : { 'Nome' : Pedro, 'Sobrenome' : 'Silva' },
    'curso' : [ 'Ciência da Computação', 'Análise e
desenvolvimento de sistemas']
}
```

Fonte: elaborado pelo autor.

Um exemplo de uso desse tipo de implementação é o MongoDB (REDMOND e WILSON, 2012). Esse modelo apresenta as seguintes vantagens:

1. Capacidade organizacional: junto do JSON é possível armazenar arquivos e outros tipos de dados binários;
2. Forma organizada: existem grande quantidade de informação já colocada em JSON, dessa forma a migração é algo menos complicado;
3. Informação variada: pela sua estrutura é possível ter dados em que alguns elementos do registro estão disponíveis e outro não (*null*).

O sistema orientado a colunas organiza a informação de maneira semelhante ao modelo de *document stores*. Todavia, ele cria estruturas separadas para juntar informações pertinentes a uma entidade. Sua aplicação é considerada mais específica, pois os casos de uso são relacionados a sistemas com grande volume de dados já estruturados e com necessidade de processamento muito rápida. O Quadro 4.3 apresenta um exemplo desse sistema; repare que a estrutura dos dados é mais presente.

Quadro 4.3 | Organização de um sistema NoSQL baseado em colunas

Família de super coluna: Clientes **RowID: 1**
Super Coluna: Nome Nome: João Sobrenome: Silva
Super Coluna: Curso Nome: Ciência da Computação Conclusão: 2020
RowID: 2
Super Coluna: Nome Nome: Pedro Sobrenome: Silva
Super Coluna: Curso Nome: Engenharia de Computação Conclusão: 2020

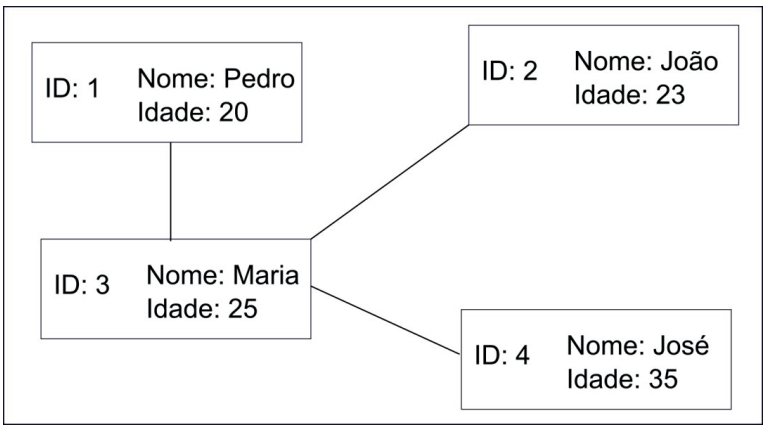
Fonte: elaborado pelo autor.

Exemplos de implementação desse modelo é o Apache Cassandra, aplicado para processamento dos dados da Netflix e da Organização Europeia para Pesquisa Nuclear (CERN, do inglês *European Organization for Nuclear Research*).

O último modelo consiste em uma estrutura de dados baseada em grafos, em que a informação pode ter uma estrutura inicial de *key-value*. Porém, existe algum tipo de relacionamento entre cada uma das entradas dos dados. A Figura 4.2 apresenta um exemplo desse tipo de forma de armazenamento, no qual o item de ID 1 possui ligação com o item 3, e o item 3 possui ligações com o 2 e 4. Um exemplo desse tipo de modelo é neo4j.

Exemplificando

Figura 4.2 | Sistema de armazenamento por forma de grafos



Fonte: elaborado pelo autor.

Existem diversos cenários em que um NoSQL baseado em grafos pode ser utilizado:

- Sistema de recomendações de filmes;
- Armazenamento de sistema forense;
- Sistema de armazenamento para machine learning;
- Relacionamento entre pessoas em uma rede social;
- Comunicação de sistema de *internet of things* (Iot).

Podemos então citar diversos modelos de sistemas NoSQL:

- Amazon SimpleDB. Disponível em: <https://aws.amazon.com/pt/simpledb/>. Acesso em: 23 jun. 2018;
- Apache Cassandra. Disponível em: <https://cassandra.apache.org/>. Acesso em: 23 jun. 2018;
- neo4j. Disponível em: <https://neo4j.com/>. Acesso em: 23 jun. 2018;
- MongoDB. Disponível em: <https://www.mongodb.com>. Acesso em: 23 jun. 2018;
- RavenBD. Disponível em: <https://ravendb.net/>. Acesso em: 23 jun. 2018;
- Redis. Disponível em: <https://redis.io/>. Acesso em: 23 jun. 2018.

Fique atento! Por ser uma tecnologia nova, é essencial o estudo de diversas implementações e testes para validar a migração para um NoSQL.

Pesquise mais

Existem diversas fontes de informação, todavia, como o NoSQL ainda está ficando popular entre os programadores, é essencial a utilização de informação confiável e sem tendências para fabricantes:

- NoSQL Guide. Disponível em: <http://nosql-guide.com/>. Acesso em: 23 jun. 2018.
- Introdução ao NoSQL - Curso de NoSQL | Devmedia. Disponível em: <https://www.youtube.com/watch?v=890222oPMmM>. Acesso em: 23 jun. 2018.
- SADALAGE, P. J.; FOWLER, M. **Nosql essencial:** Um Guia Conciso para o Mundo Emergente da Persistência Poliglota. 1. ed. São Paulo: Novatec, 2013. 216 p.

Sem medo de errar

Você está trabalhando em uma empresa especializada em projetos de migração de sistemas legados. Depois de fazer diversas

tarefas e já estar com grande parte do projeto pronto, seu líder recebeu uma nova demanda para detectar com antecedência, com base nos dados do novo sistema, algumas necessidades dos alunos. Para isso, será necessário aplicar técnica de inteligência artificial, porém, antes de iniciar esse processo, é preciso tornar os dados mais propícios para esse processamento com mais desempenho. Dessa forma, é necessário verificar na tecnologia do NoSQL:

1. Tipo de modelo de armazenamento;
2. Projetos e indicativos de uso;
3. Aplicação do modelo de dados do projeto no modelo de dados do NoSQL.

Dos diversos modelos de NoSQL, podemos destacar:

1. *Key-value*: sistema que utiliza um valor de índice que aponta para uma estrutura que guarda os dados. Utilizado em sistemas mais gerais, com grande facilidade em escalabilidade;
2. *Document stores*: modelo que utiliza um sistema de arquivos formatados para fazer o armazenamento. Utilizando também em sistemas mais gerais, e com facilidade de migração em sistemas que utilizam a mesma formatação dos dados;
3. Orientado a colunas: uma grande coluna de dados em que os dados são armazenados e uma relação. Modelo dedicado para sistema de processamento para conversão de dados, em que eles são relacionados;
4. Grafo: dados que possuem uma relação entre eles, como amigos em uma rede social.

Dados esses modelos, podemos pensar em utilizar os sistemas mais gerais, no qual o modelo de armazenamento do NoSQL não ofereça uma especialização. Nesse caso, podemos escolher os sistemas baseados em *key-value* ou *document stores*, podendo escolher Amazon SimpleDB ou MongoDB. Como alguns campos podem ainda não estar preenchidos, devido ao processo de migração, o MongoDB com o seu sistema por *document store* se apresenta com mais vantagens. Um exemplo de como os dados serão armazenados é apresentado no Quadro 4.3. Nele, cada dado será armazenado no formato JSON, de forma mais organizada.

Quadro 4.4 | Exemplo de como armazenar os dados no MongoDB

```
{
    '_id' : 1,
    'nome' : { 'Nome' : João, 'Sobrenome' : 'Silva' },
    'curso' : [ 'Ciência da computação'],
    'notaMedia' : [ 9.8]
}
```

Fonte: elaborado pelo autor.

Para finalizar, complete sua pesquisa ilustrando o armazenando usando JSON de outras tabelas do sistema. Com esses dados em mãos, você poderá emitir sua indicação de modelo de NoSQL.

Avançando na prática

Sistema para reconhecimento de imagens por análise de padrão

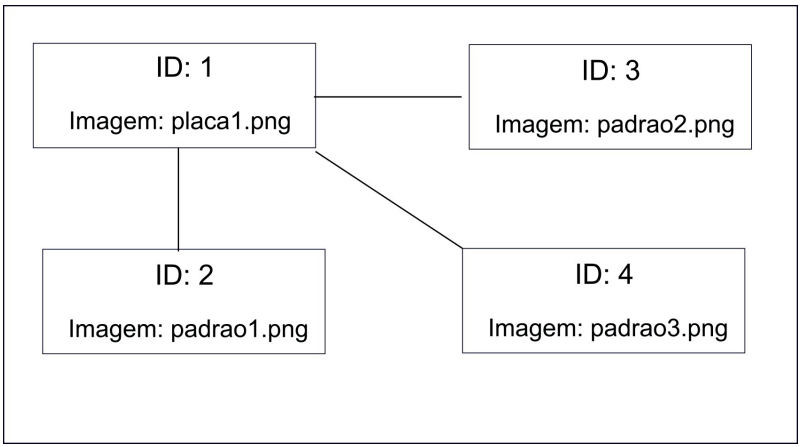
Descrição da situação-problema

Você está trabalhando em uma empresa dedicada a análise de imagens, e o seu projeto atual consiste em detectar falhas em circuitos impressos utilizando imagens. Nesse caso, é necessário guardar no banco de dados as informações da imagem e quais são os padrões que estão relacionados para então averiguar quais placas possuem ou não possuem defeito. Sua tarefa consiste em escolher um modelo e descrever como serão armazenados os dados das imagens e seus respectivos padrões.

Resolução da situação-problema

Para esse tipo de cenário é necessário utilizar um sistema NoSQL. Nesse cenário, uma boa opção é utilizar o modelo de grafos, pois assim mantem-se uma relação entre os dados da estrutura. A Figura 4.3 apresenta um exemplo de como esses dados podem ser organizados.

Figura 4.3 | Organização no modelo grafo em NoSQL



Fonte: elaborado pelo autor.

Faça valer a pena

1. Os sistemas de gerenciamento de banco de dados (SGDB) são serviços essenciais para garantir o armazenamento e acesso aos dados. Com os SGDB relacionais é possível compartilhar, proteger um grande conjunto de informação e descrever quais são as ligações dos dados. Todavia, o movimento NoSQL tem o objetivo de tornar esse processo mais simples.:

Qual das características a seguir representa o objetivo de um sistema NoSQL?

a) Dependência de esquema de estrutura de dados.
b) Junções de tabelas.
c) Crescimento vertical.
d) Independência de esquema de organização.
e) Dependência de sistema de centralização.

2. Os modelos NoSQL são relacionados ao tipo de dado que será armazenado, dessa forma, cada projeto deve explicar qual é sua forma interna de armazenamento e explicar aos desenvolvedores quais são os cenários ideais. Dados os diversos cenários que os NoSQL operam, são necessários modelos diferentes para propósitos variados.

Quais são os quatro modelos de NoSQL mais utilizados?

a) *Key-value*, *document store*, orientado a linha e grafo.
b) *Key-key*, *document colun*, orientado a coluna e grafo.
c) *Key-table*, *document store*, orientado a coluna e árvore.
d) *Key-value*, *document store*, orientado a coluna e árvore.
e) *Key-value*, *document store*, orientado a coluna e grafo.

3. Dados os diversos modelos de NoSQL é possível pensar em cenários mais adequados para cada um deles. Em um cenário no qual cada uma das entradas pode ter um relacionamento com outras entradas, esse modelo pode ser feito com um sistema de gerenciamento de banco de dados relacional. Todavia, existe um modelo de NoSQL direcionado a esse tipo de dado.

Pensando em um cenário no qual os registros devem ter relacionamento com outros registros, qual dos modelos deve ser utilizado?

a) *Key-value*.
b) *Document store*.
c) Orientado a colunas.
d) Grafo.
e) *Key-key*.

Seção 4.2

Introdução ao desenvolvimento em Java usando MongoDB

Diálogo aberto

Novas tecnologias necessitam de novas formas de utilização, e dependendo do cenário, necessita-se de uma curva de aprendizado maior. Você já pensou que quando é lançada uma versão nova de um smartphone é necessário verificar quais opções e quais recursos estão disponíveis para este novo dispositivo? Isso também ocorre quando se pensa na utilização de um banco de dados NoSQL, quando comparado a um sistema de gerenciamento de banco de dados relacional. Diante das novas possibilidades é necessário apropriar-se do conhecimento necessário à utilização dos novos recursos.

Dando continuidade ao seu trabalho na equipe de *backend* no projeto de migração de um sistema legado, foi lhe solicitado usar novas tecnologias que sejam mais aderentes aos recursos de inteligência artificial, para prever possíveis falhas que podem remeter a evasão de um curso. Como esse sistema é novo e necessita de um desempenho satisfatório, foi escolhido o uso de um banco de dados NoSQL, chamado MongoDB. Sua tarefa nesse momento, consiste em montar o ambiente de desenvolvimento dos sistemas atendendo aos seguintes pontos:

1. Instalação dos componentes;
2. Definir como é feito o acesso básico;
3. Definição e configuração inicial da forma de armazenamento para os seguintes dados:
    a. Nome;
    b. Curso;
    c. Nota média de todas as disciplinas;
    d. Data de matricula;
    e. Data de evasão.

Para isso é necessário descrever os passos de como é feita a instalação, a definição de acesso básico, assim como inserir um dado de exemplo no banco de dados recém-criado.

Nessa etapa do estudo, focaremos na instalação inicial do MongoDB juntamente com o Java e todas as dependência necessárias. Após isso, serão apresentadas a conexão básica com o NoSQL e as possíveis configurações necessárias.

Não pode faltar

Existem diversas soluções e projetos que possuem os conceitos de NoSQL. Considerando o fato de ser uma tecnologia relativamente nova, com foco em cenários que necessitam armazenar grandes quantidades de dados, é necessário certo cuidado nas escolhas (GESSERT, 2016). Entre diversos projetos, o MongoDB apresenta maturidade e estabilidade, o que remete a escolha por esse projeto (HAN, E, e LE, 2011). O MongoDB tem seu nome derivado da palavra *humongous* que significa enorme ou monstruoso, declarando um novo sistema de gerenciamento de banco dados com as premissas do NoSQL (PLUGGE; MEMBREY e HAWKINS, 2010).

O MongoDB utiliza uma estrutura do sistema NoSQL chamada *document stores*, em que são utilizados arquivos formatados para fazerem o armazenamento da informação. Esses arquivos utilizam o formato JSON (*JavaScript Object Notation*), sendo os dados gravados em modo binário. Este formato tem sido muito utilizado devido ao seu processo de leitura (por humanos e computadores) ser considerado simples quando comparado aos sistemas mais complexos, como grafos ou formatos proprietários. Além disso, já existem diversas ferramentas com processo consolidado para a conversão do *Extensible Markup Language* (XML) para o JSON, tornando a utilização e migração menos complexa.

O MongoDB possui 3 elementos básicos para o armazenamento, que consistem no banco de dados, na coleção e no registro. Cada banco de dados possui diversas coleções, essas coleções são conjuntos de registros com a mesma estrutura, já os registros são os dados armazenados. Existem diversas versões disponíveis deste projeto, tais como a Atlas, o *Community Server* e o *Enterprise Server*. A versão Atlas é uma versão do MongoDB para ser utilizada na nuvem, composta por formas simples de criar instâncias e conjuntos do SGDB (REDMOND; WILSON, 2012). A versão que utilizaremos será a *Community Edition* que pode ser utilizado de forma livre. A versão *Enterprise Edition* envolve custos de licenciamento, porém

possui algumas características a mais que a versão *Community* que consistem em:

- Sistema que executa diretamente na memória RAM;
- Criptografia dos dados;
- Sincronismo com o LDAP e Kerberos (sistema de controle de usuários e senhas).

Veja os passos necessários para a configuração em um ambiente Windows da versão *Community*.

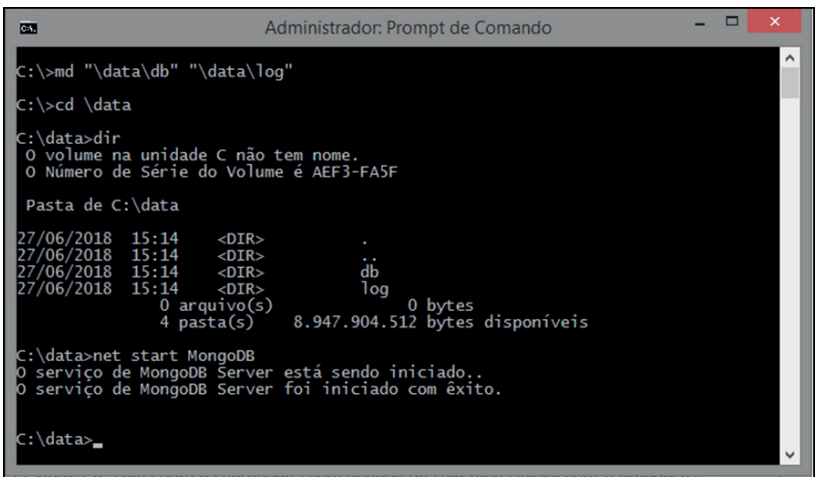
- Fazer o donwload em <https://www.mongodb.com/download-center?jmp=nav#community> (aba *Community Server*). Existem versões para GNU/Linux, Microsoft Windows e Mac OSX; nesse caso se deve selecionar a versão para Microsoft Windows versão 4.0, a arquitetura do computador deve ser x86. Durante a instalação, selecione a versão "*Custom*" e certifique que todas os componentes estão selecionados e serão instalados.
- Após essa etapa, selecione a opção "*Install MongoD as a Service*". No item "*Data Directory*", insira o seguinte caminho: ***C:\data\db*** e no item "*Log Directory*" insira: ***C:\data\log***. Utilizar a opção de instalar como serviço garante que o Microsoft Windows controla o início de forma independe.
- Após essa etapa, selecione o item que define a instalação do "*MongoDB Compass*".

O processo de instalação é simples e intuitivo, siga os passos que forem indicados. Depois da instalação, é necessário verificar se as pastas, onde serão armazenados os bancos de dados e os arquivos de log do MongoDB, foram criados (*C:\data\db* e *C:\data\log*). Para isso, você pode utilizar o Explorador de Arquivos do Microsoft Windows. Caso não estejam criados, você deverá criar as pastas indicadas. Também é possível fazer esse processo utilizando o *prompt* de comando do Windows (*Command Prompt* ou CMD). Para abrir o CMD você pode fazer uma pesquisa usando o campo de buscas (busque o programa *prompt* de comando) ou, dependendo da versão do Windows, clique com o botão direito do mouse sobre o ícone do menu iniciar e selecione *prompt* de comando. O CMD deve ser executado como Adminstrator.

Caso os diretórios não estejam lá, eles devem ser criados com o CMD, para serem utilizados para armazenar o próprio

arquivo de dados e de logs do MongoDB. Esse SGDB procura no diretório raiz onde ele foi instalado, porém isso pode ser alterado caso o programador deseje. Por padrão, o MongoDB é instalado no C:\, portanto, é necessário criar os diretórios C:\data\db e C:\data\log. Com o comando *md "\data\db" "\data\log"* é possível criar os dois diretórios. A Figura 4.4 apresenta o resultado dos comandos no *prompt*. Repare que ao usar o comando *cd \data* o diretório corrente do CMD é definido como *C:\data* e com o comando *dir* é possível ver os dois diretórios internos *db* e *log*.

Figura 4.4 | CMD com os comandos para criação dos diretórios inicial para o MongoDB e inicialização



Fonte: captura de tela do *prompt* de comando, elabora pelo autor.

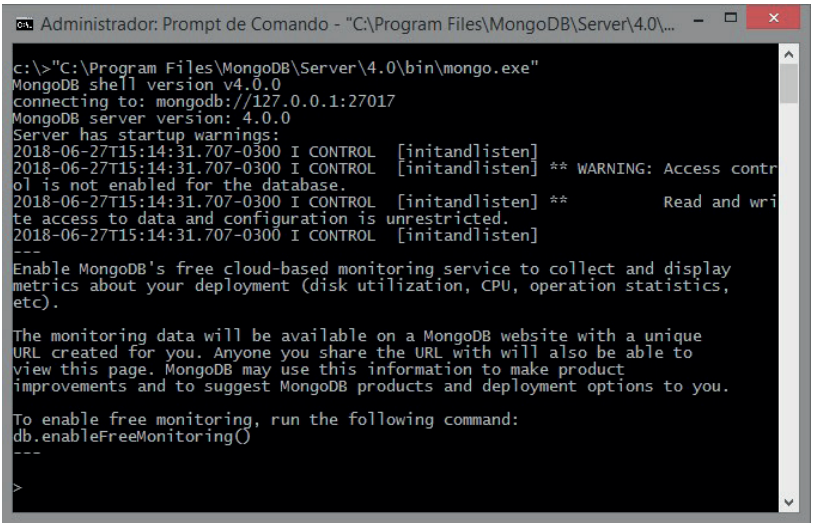
Assimile

As pastas *C:\data\db* e *C:\data\log* são usadas pelo MongoDB para armazenar os bancos de dados e os arquivos de log, respectivamente. Ao executar o MongoDB, caso ele não encontre tais pastas, será dado um erro e o SGBD não será aberto.

Antes de iniciar o uso do MongoDB é necessário garantir que o servidor *"mongod"* foi iniciado. Para isso, ainda no *prompt* e como

administrador, é necessário utilizar o comando "*C:\Program Files\MongoDB\Server\4.0\bin\mongod.exe*" (sendo obrigatório utilizar as aspas). Ao "subir" o servidor, você verá na última linha, dentre várias mensagens, a instrução "*waiting for connections on port 27017*". Veja que o servidor está esperando por conexões na porta 27017, que é a padrão usada pelo MongoDB. Agora, você deve abrir uma nova janela de *prompt* (sem fechar a que está aberta) e utilizar o comando "*C:\Program Files\MongoDB\Server\4.0\bin\mongo.exe*". A Figura 4.5 apresenta o resultado de executar o *MongoDB shell*; repare que no final da imagem a entrada de comando está apenas com o sinal de maior (>), isso significa que o *MongoDB shell* está esperando por comandos.

Figura 4.5 | Entrada para o *MongoDB shell*



Fonte: captura do *MongoDB Shell*, elaborada pelo autor.

Pesquise mais

No canal da Bóson Treinamentos você poderá assistir ao vídeo **MongoDB-Instalando no Microsoft Windows** (disponível em: <https://www.youtube.com/watch?v=pABKEjwZBUw>. Acesso em: 02 ago. 2018), que mostra a instalação e a configuração do MongoDB. Nesse vídeo, há uma configuração adicional na variável Path do sistema, para não precisar o caminho completo para acessar os arquivos necessários à execução.

Existem diversas operações que podem ser feitas nessa interface textual do MongoDB; o Quadro 4.5 apresenta comandos que podem ser utilizados. Fique atento que o comando *use <nome BD>* é utilizado para criar a base de dados se ele não existir, caso exista, é definido como banco de dados ativo.

Quadro 4.5 | Comandos básico do *MongoDB shell*

Comando	Descrição
show dbs	Apresenta os bancos de dados disponíveis.
show colletions	Exibe as *collections*, que representam o conjunto de dados que estão no banco de dados.
show users	Apresenta os usuários do banco de dados corrente.
use <nome BD>	Define o banco de dados corrente ou cria o banco de dados, se puder existir diversos bancos por sistema (trocando o <nome BD> pelo nome do banco desejado).
db.<collection>.find()	Exibe os dados armazenados na coleção com o nome que será especificado no parâmetro <collection>.

Fonte: elaborado pelo autor.

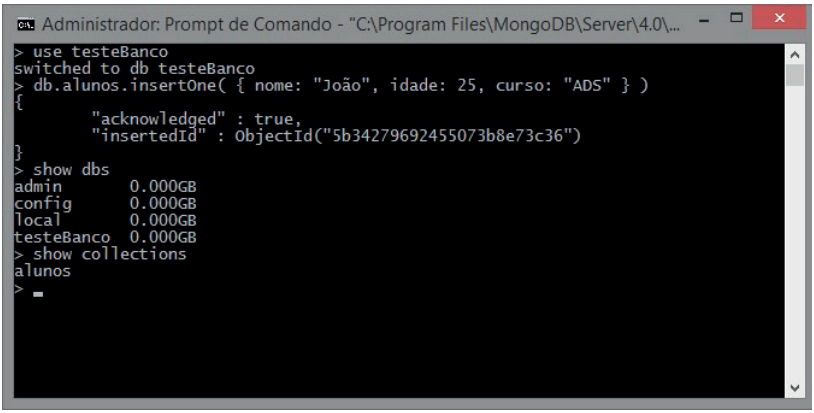
Reflita

A utilização da linha de comando necessita que o administrador do banco de dados lembre-se dos comandos e seus parâmetros. Todavia, em situações emergenciais, o tempo que se economiza em utilizar a linha de comando pode ser vital. Dessa forma, qual é mais vantajoso: utilizar uma interface gráfica ou a linha de comando para administração de um banco de dados?

É possível criar um banco executando o comando: `"use testeBanco"`, onde *testeBanco* é o nome do banco a ser criado (Figura 4.6). Ao confirmar o comando com "*enter*", recebemos a resposta "*switched to db testeBanco*", porém, ao executar o comando "*show dbs*" ele não aparece na lista (mesmo tendo sido criado). Para que ele apareça na lista é necessário que tenha algum registro contido nesse banco. Então, para fazer a inserção de algum

registro pelo terminal, é necessário utilizar o comando *db.alunos.insertOne({ nome: "João", idade: 25, curso: "ADS" })*, onde o *alunos* é o nome da *collection*, os dados são separados por vírgula, e o texto é definido por aspas duplas.

Figura 4.6 | Criação de um banco de dados utilizando o MondoDB *shell*



Fonte: captura de tela do MongoDB, elaborada pelo autor.

Assimile

Para a criação de um banco de dados no MongoDB, via linha de comando, é necessário seguir seguintes os passos:

1. Abrir o CMD do Microsoft Windows como Administrador;
2. C:\Program Files\MongoDB\Server\4.0\bin\mongod.exe
3. Abrir outra janela de CMD do Microsoft Windows como Administrador;
4. Executar o comando C:\Program Files\MongoDB\Server\4.0\bin\mongo.exe.
5. No *MongoDB shell* você deve utilizar os seguintes comandos (substituindo os valores entre < > por suas escolhas):
    a. use <*nome do banco*>
    b. db.<*nome da collection*>.insertOne({ chave1: "Valor1", chave2: 25 })
    c. show dbs
    d. show collections

O comando *db* possui diversos outros elementos que podem ser usados. O Quadro 4.5 apresenta um conjunto dos mais utilizados. Tais comandos são muitos úteis para administrar o banco, de maneira rápida, podendo inclusive gerar scripts para automatizar tarefas.

Quadro 4.6 | Comandos para o banco de corrente derivados do comando *db*

Comando	Descrição
db.dropDatabase()	Remove o banco de dados corrente.
db.getName()	Apresenta o nome do banco de dados corrente.
db.stats()	Apresenta estatísticas do banco de dados corrente, tais como número de *collections*, utilização de espaço em discos e outros.
db.getLastError()	Apresenta o último erro da base dados corrente.

Fonte: elaborado pelo autor.

Repare que a criação e a inserção de dados no MongoDB, ocorre sem a obrigatoriedade de uma modelagem, na qual se definem tabelas, tipos de dados e relacionamentos. No MongoDB as "colunas" são criadas automaticamente, à medida que os dados são inseridos. O único passo obrigatório é definir em qual coleção os dados serão inseridos.

Exemplificando

Para criar um conjunto de dados é necessário o comando *insertOne*. Nesse sentido, se temos um conjunto de dados com os seguintes campos:

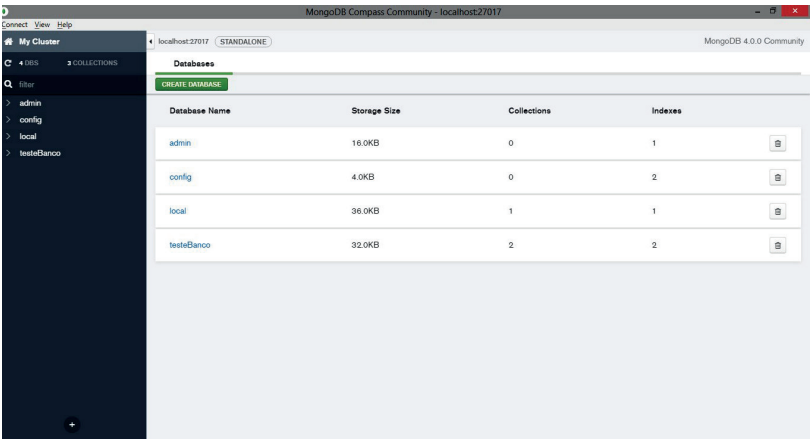
- Nome completo;
- Logradouro;
- Número do logradouro;
- Bairro;
- CEP:

O comando necessário é:

```
db.cadastro.insertOne({ nomeCompleto: "João da Silva", logradouro: "Rua das Amoreiras", numeroLograroudo: 120, bairro: "Centro", cep: "12912-020" })
```

Outra forma de se administrar o banco de dados é utilizar o *MongoDB Compass*, que é instalado junto com o pacote. Com ele é possível fazer as mesmas ações que estão disponíveis na linha de comando, mas usando uma interface gráfica. A Figura 4.7 apresenta uma visão da interface gráfica para controle do MongoDB.

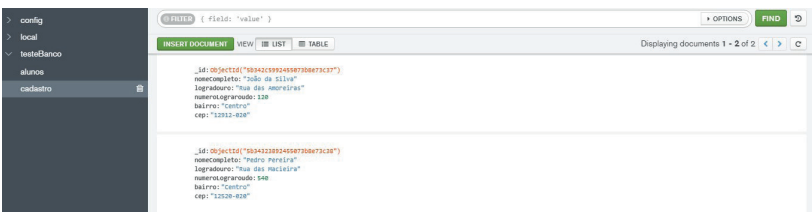
Figura 4.7 | Interface gráfica para administração do MongoDB



Fonte: captura de tela do MongoDB, elaborada pelo autor.

Para usar essa ferramenta também é necessário carregar o servidor "*mongod*" antes de iniciar a interface gráfica. Uma opção muito interessante que o MongoDB Compass oferece é a visualização dos dados. Na Figura 4.7 estão sendo exibidos todos os bancos de dados disponíveis, ao selecionar um deles é possível ver quais *collections* ele possui. A Figura 4.8 apresenta, em detalhe, como os dados são apresentados.

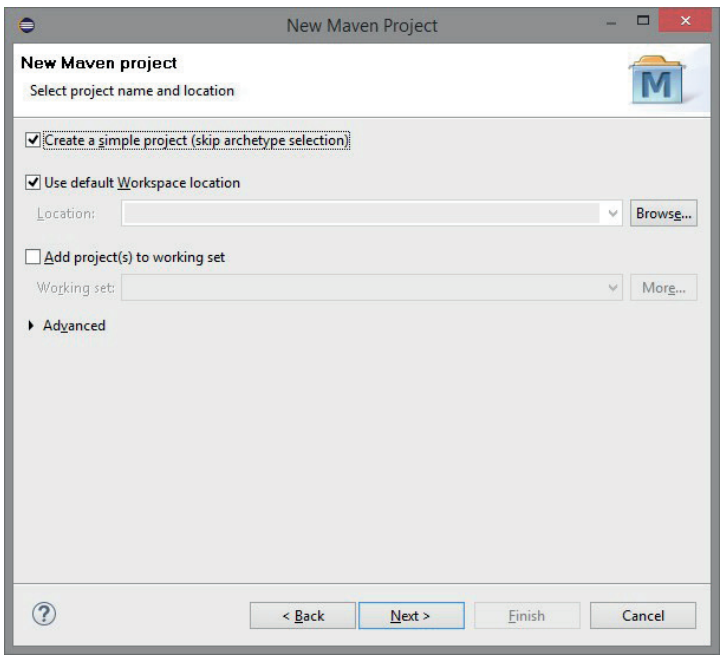
Figura 4.8 | Detalhe da interface do MongoDB Compass apresentado os dados



Fonte: captura do MongoDB Compass Community, elaborada pelo autor.

Após a instalação do MongoDB e dos testes iniciais, é necessário instalar as dependências necessárias para utilizar esse SGBD junto ao Java e ao Eclipse. Uma das opções é baixar o arquivo *Jar* e incluí-lo diretamente no projeto, o que está cada vez menos aconselhado, pois todo o tratamento de dependência é feito pelo programador. Uma forma mais dinâmica é utilizar o *Maven*, que faz o tratamento e controle das dependências das bibliotecas em uma aplicação Java de maneira documentada. Com esse componente é possível apenas indicar o software que será necessário e ele trata todas as dependências. No Eclipse é possível criar um projeto já com o suporte ao *Maven* seguindo o menu *File* >> *New* >> *Other...*. No menu para criação de novo item, buscar *Maven* e selecionar o *Maven Project*. A Figura 4.9 apresenta a primeira interface para a criação do projeto com suporte ao *Maven*. Nessa etapa é importante selecionar a opção "*Create a simple project (skip archetype selection)*" e com isso não será necessário escolher entre as arquiteturas que o projeto se encaixa na formatação do *Maven*.

Figura 4.9 | Criação do projeto com o *Maven*



Fonte: captura de tela do Eclipse IDE, elaborada pelo autor.

Na segunda interface é preciso definir o *Group Id* e *Artifact Id*, onde o primeiro representa um nome único dos projetos e o segundo definirá o nome para a distribuição do projeto (*jar*). Por exemplo, podemos definir o *Group Id* como *exemplo.mongodb* e o *Artifact Id* como *JavaMongoDBTeste*. As outras opções podem ser deixadas como padrão.

Repare que no projeto criado, existe um arquivo chamado *pom.xml*. Ele controla diversos aspectos no projeto, tais como as dependências, e ao abri-lo é possível ver um editor específico no Eclipse. Na parte inferior dessa tela, procure por uma aba escrito "pom.xml" e selecione essa opção para editar diretamente no arquivo, adicionando o conteúdo XML fornecido pelo projeto do MongoDB em <https://mongodb.github.io/mongo-java-driver/> (acesso em: 02 jul. 2018). O arquivo deve ficar semelhante ao Quadro 4.6, no qual você pode notar que a parte de "*dependencies*" foi inserida conforme recomendações.

Quadro 4.7 | Configuração do projeto para utilização do MongoDB

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>exemplo.mongodb</groupId>
  <artifactId>JavaMongoDBTeste</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongodb-driver</artifactId>
        <version>3.6.4</version>
    </dependency>
</dependencies>

</project>
```

Fonte: elaborado pelo autor.

Com isso o projeto está pronto para fazer acesso ao MongoDB pelo Java; dessa forma não é necessário tratar diretamente as

dependências e o programador pode focar no desenvolvimento. A utilização do MongDB é mais simples e direta que de um SGDB relacional, sendo necessário menos tempo para configurações e ajustes na estrutura do banco. Essa utilização vai de acordo com as metodologias ágeis, pois a ideia é gastar mais tempo no desenvolvimento do sistema do que planejar algo que pode nem ser aceito pelo cliente.

Pesquise mais

Existem diversas fontes disponíveis para fazer a configuração e utilização do MongoDB, tais como:

1. Documentação do MongoDB: esse deve ser o repositório inicial de consulta sobre esse NoSQL. Nele é possível encontrar todos os comandos, opções e descrições sobre o MongoBD. Disponível em: <https://docs.mongodb.com/>. Acesso em: 27 jun. 2018;

2. A parte entre as páginas 47 a 60 apresentam as formas de acesso aos dados do MongoDB.
PLUGGE, E.; MEMBREY, P.; HAWKINS, T. **The Definitive Guide to MongoDB**. New York: Apress, 2010. 328 p.

3. O site nosql-database dispõem de diversos artigos sobre sistema NoSQL, sendo uma fonte de dados atuais e caso de uso sobre os NoSQL. Disponível em: <http://nosql-database.org/>. Acesso em: 29 jun. 2018.

4. Documentação e como instalar o Maven para Eclipse. Disponível em: <http://www.eclipse.org/m2e/>. Acesso em: 02 jul. 2018.

Sem medo de errar

A empresa que você trabalha está com um projeto para produzir um sistema com recursos que propiciem a utilização de inteligência artificial aplicado a evitar situações que levem a evasão de alunos. Como o volume de dados é grande, está sendo considerado a utilização de um banco de dados NoSQL.

Seu líder pediu para que você montasse o ambiente de testes e definisse os dados iniciais. Para isso, é necessário seguir os passos para:

- Instalação dos componentes;
- Definir como é feito o acesso básico;
- Definição e configuração inicial da forma de armazenamento definindo os seguintes dados:
    - Nome;
    - Curso;
    - Nota média de todas as disciplinas;
    - Data de matricula;
    - Data de evasão.

O primeiro passo consiste em instalar o MongoDB usando o endereço <https://www.mongodb.com/download-center?jmp=nav#community> (acesso em: 09 ago. 2018), selecionando a opção "Communinty Server", opção "Windows" e "Windows 64-bit x64". Durante a instalação, selecionar a versão "*Custom*", e se certificar que todas os componentes serão instalados. Após essa etapa, deve-se selecionar a opção "Install MongoD as a Service", e definir para:

- "**Data Directory**": deve indicar C:\data\db;
- "**Log Directory**": deve indicar C:\data\log.

Os próximos passos consistem em construir os arquivos onde ficarão os logs e banco (caso eles não tenham sido criados durante a instalação):

- Para abrir Command Prompt (CMD) se deve clicar com o botão direito do Mouse sobre ícone do Menu Iniciar e selecionar Prompt de Comando (Adminstrator);
- Já no cmd se deve entrar com o comando: md "\data\db" "\data\log"

Criadas as pastas obrigatórias, agora é preciso carregar o servidor, e depois o MongoDB. Para isso:

- Abra o CMD do Microsoft Windows como Administrador;
- Use o comando: "C:\Program Files\MongoDB\Server\4.0\bin\mongod.exe".
- Abrir outra janela de CMD do Microsoft Windows como Administrador.
- Conectar via linha de comando no MongoDB: "C:\Program Files\MongoDB\Server\4.0\bin\mongo.exe".

Com esses passos já está disponível o *MongoDB shell* (modo texto). Com ele é possível definir qual é o banco ativo, inserir um registro em uma *collection* e mostrar quais bancos estão disponíveis e quais *collections* estão disponíveis no banco corrente. Os passos a seguir representam essa sequência:

- use bancoalunos
- db.evasao.insertOne({ nome: "João", curso: "Ciência da Computação", notamedia: 3.0, dataevasao: "21/10/2017", datamatricula: "01/02/2015" })
- show dbs
- show collections

Com isso o ambiente de desenvolvimento está pronto e é possível planejar como o software utilizará esse sistema de armazenamento.

Avançando na prática

Armazenamento de dados de clientes para *data mining*

Descrição da situação-problema

A empresa que você está atuando quer utilizar um banco de dados NoSQL para armazenar os dados dos clientes ativos e não ativos, com o objetivo de avaliar quais são os indicativos para os clientes pararem de comprar. Para isso, é necessário criar dois bancos no MongoDB (um para os ativos e outro para os inativos), com os seguintes dados:

- Nome;
- CNPJ;
- Data da última compra;
- Reclamações.

Como pode ser feito a criação de bancos de dados no MongoDB, considerando que o sistema já está instalado?

Resolução da situação-problema

Para resolver essa questão é necessário utilizar o *MongoDB shell* e executar os comandos para definir o banco de dados correte e inserir um registro como exemplo, tal como:

1. use clienteativo
2. db.ativos.insertOne({ nome: "Loja XYZ", cnpj: "25.769.445/0001-61", dataultimacompra: "05/05/2018", reclacamoes: 0 })

E para criar o banco de dados dos clientes inativos:

1. use clienteinativo
2. db.inativos.insertOne({ nome: "Loja TYU", cnpj: "35.869.445/0001-61", dataultimacompra: "01/01/2010", reclacamoes: 5 })

Com esses passos foi possível criar os dados de forma rápida e intuitiva, sem a necessidade de criar toda uma modelagem para que o sistema pudesse ser desenvolvido. Com isso, a produção do software pode gerar entregáveis de forma mais periódica relacionado com as diretrizes do desenvolvimento ágil de software.

Faça valer a pena

1. Lema de gerenciamento de bancos de dados (SGDB) relacional difere da estrutura dos sistemas NoSQL. As diferenças são evidentes antes mesmo da implementação, ou seja, logo na fase de modelagem. No primeiro caso é preciso ter um projeto somente para o banco de dados, contendo as tabelas, campos, tipos e relacionamentos, dentre outras características.

Analise as afirmações a seguir considerando um cenário que utiliza o MongoDB:

I. Os bancos de dados são estruturas que armazenam as *collections*.
II. As *collections* são um conjunto de bancos de dados onde os dados são armazenados.
III. As *collections* são armazenadas dentro dos bancos de dados e uma collection representa um grupo de dados.
IV. Não é possível ter mais um banco de dados por sistema MongoDB.
V. Ao se utilizar o comando use <*nome BD*> no MongoDB, e ele não existir, o sistema apresentará um erro.

a) Apenas as afirmações I, III, V são verdadeiras.
b) Apenas as afirmações II, IV são verdadeiras.
c) Apenas as afirmações I, III são verdadeiras.
d) Apenas as afirmações I, IV e V são verdadeiras.
e) Apenas as afirmações I, II e III são verdadeiras.

2. O MongoDB é um ótimo exemplo de um sistema NoSQL, no qual é possível fazer a instalação e controle dos dados de maneira intuitiva. Existem duas formas de fazer essa administração, uma dela é feita com uma interface gráfica e outra utilizando a linha de comando.

Qual das opções apresenta apenas comandos de linha do MongoDB?

a) use, drop, select, set.
b) show, db, use, db.dropDatabase(), view.
c) select, view, set, create.
d) show, db, use, show collections.
e) set, create, show, release.

3. A administração de um sistema de gerenciamento de banco de dados (SGDB) do tipo NoSQL tem menos opções que um sistema de banco de dados relacional. Essa característica vem diretamente da própria filosofia que os sistemas NoSQL apresentam.

Dentre os comandos abaixo que podem ser usados no *MongoDB shell*, escolha a opção que descreve a sequência correta para fazer a criação de um banco, a criação de uma *collection*, e suas respectivas visualizações (banco e collection).

1 – show collections.
2 - db.pessoas.insertOne({ info: "dados", numero: 70 }).
3 - show dbs.
4 - use nomeBanco.

a) 1 – 2 – 3 – 4.
b) 3 – 1 – 2 – 4.
c) 1 – 2 – 2 – 4.
d) 2 – 4 – 3 – 1.
e) 4 – 2 – 3 – 1.

Seção 4.3

Desenvolvimento em Java usando MongoDB

Diálogo aberto

Para utilizar qualquer elemento novo em nosso cotidiano, é necessário um tempo de adaptação. Como exemplo, podemos pensar quando temos um novo carro. Todos os automóveis possuem certos elementos básicos, como acelerador, freio ou luzes. Porém, as formas de utilização dos elementos podem variar para cada marca ou modelo, o que demanda uma certa adaptação do usuário a um novo carro. No caso da utilização de um sistema de gerenciamento de banco de dados novo, o processo funciona de forma semelhante, pois o acesso para leitura, escrita, alteração e remoção de dados é semelhante ao sistema de gerenciamento de banco de dados relacional, porém é necessário atentar a alguns detalhes. Portanto, ao se trabalhar com um banco de dados não relacional, algumas novas habilidades devem ser desenvolvidas.

Chegou o momento de finalizar o projeto de migração do sistema legado da universidade. Seu líder está interessado cada vez mais na versatilidade dos sistemas NoSQL, pois as características de desempenho e escalabilidade estão se mostrando de grande valia. Todavia, como é um processo diferente do adotado em sistemas com SQL clássico, diversos pesquisadores estão relutando para iniciar o projeto com essa tecnologia. Seu papel é garantir que todos adotem um padrão para fazer o acesso ao banco e produzir um conjunto de testes para mostrar as vantagens desse novo modelo. Para isso, você deve implementar um conjunto de códigos que possa fazer:

- Acesso ao banco de dados, passando por parâmetro o endereço do computador em que está o banco, o nome do banco e a coleção a ser utilizada.
- Inserção dos dados contendo nome completo do aluno, idade do aluno, curso que evadiu e data da evasão.
- Seleção para exibição no terminal dos dados.

- Alteração dos dados utilizando como chave o item id.
- Remoção dos dados utilizando como chave o item id.

Nesta seção, você vai aprender os comandos básicos para fazer acesso ao MongoDB e a manipulação dos dados. Com esse conhecimento será possível utilizar o banco de NoSQL no seu dia a dia. Para isso será utilizada a API fornecida pelos desenvolvedores do MongoDB e as formas de sua utilização.

Bons estudos!

Não pode faltar

A utilização do MongoDB é feita de forma diferente quando comparada aos sistemas de gerenciamento de banco de dados (SGBD) relacionais, pois sua utilização é baseada em uma Interface de Programação de Aplicativos (API, do inglês *Application Programming Interface*) específica, fornecida pela empresa que desenvolve o MongoDB (PLUGGE; MEMBREY; HAWKINS, 2010).

Assimile

No momento da produção deste livro, a versão do MongoDB é a 4.0.0. Essa informação é relevante, pois os comandos podem variar de versão para versão. A documentação oficial é sempre a melhor opção de consulta.

Para a utilização de banco de dados NoSQL, devem ser consideradas, além de sua estrutura de armazenamento, as formas de acesso (GESSERT et al., 2016). A primeira etapa para a utilização é especificar a conexão com o MongoDB. Nesse momento tanto o banco de dados quanto a *collection* podem ser selecionados. O Quadro 4.8 apresenta como é feito esse processo de conexão, lembrando que é necessário configurar o eclipse e o projeto via Maven para que as bibliotecas estejam disponíveis – essa classe deve ser criada na pasta src.

No Quadro 4.8, as linhas de 1 a 5 são relacionadas à inclusão das bibliotecas que serão utilizadas por toda a classe. Os atributos entre as linhas 7 a 9 são os tipos necessários para fazer a

conexão. O primeiro deles (*MongoClient*) é utilizado para realizar as conexões com o MongoDB e é responsável por prover o método que faz a seleção do banco de dados que é o próximo atributo (*MongoDatabase*). Por fim, na linha 9 temos o tipo *MongoCollection*, que representa as coleções nas quais os dados são armazenados. No construtor, que inicia na linha 10, temos a criação das instâncias. Já na linha 11 é criada uma instância da classe *MongoClient*, e para isso é passado o endereço do banco (no nosso caso, *localhost*), e a porta de conexão, que por padrão para o MongoDB é a 27017. Na linha 12, utilizando a instância de *MongoClient*, é possível selecionar qual base de dados será utilizada com o método *getDatabase*. Nesse método é passada uma *String* com o nome do banco a ser utilizado, e ele retorna uma instância de *MongoDatabase*. Por fim, na linha 13, utilizando a instância de *MongoDatabase* (database), é possível obter uma instância de *MongoCollection* com o método *getCollection*, passando uma *String* com nome da coleção selecionada ("alunos").

Quadro 4.8 | Conexão com o MongoDB

```
1.  import com.mongodb.MongoClient;
2.  import com.mongodb.client.*;
3.  import org.bson.Document;
4.  import static com.mongodb.client.model.Filters.*;
5.  import org.bson.types.ObjectId;

6.  public class ConexaoMongoDB {

7.          private MongoClient mongoClient;
8.          private MongoDatabase database;
9.          private MongoCollection<Document> collection;

10.         public ConexaoMongoDB() {
11.                 mongoClient = new MongoClient
    ("localhost", 27017);
12.                 database =
    mongoClient.getDatabase("baseuniversidade");
13.                 collection =
    database.getCollection("alunos");
14.         }
15. }
```

Fonte: elaborado pelo autor.

Assimile

Existem três classes básicas para fazer o acesso ao banco de dados MongoDB:

- *MongoClient*: responsável pela conexão, definindo endereço e porta do servidor MongoDB.
- *MongoDatabase*: define o banco de dados utilizado e fornece acesso à coleção de dados.
- *MongoCollection*: fornece acesso às coleções vindas do banco de dados.

Com a conexão feita, a base de dados selecionada e a coleção definida, é possível executar algumas operações básicas, também conhecidas como CRUD. São elas:

1. Inserção.
2. Seleção.
3. Alteração.
4. Remoção.

A primeira etapa é a inserção. Em um processo utilizando um SGDB relacional, seria necessário definir a estrutura, os tipos de dados e os relacionamentos entre as tabelas. Todavia, utilizando um sistema NoSQL, basta apenas fazer a inserção dos dados, e a estrutura será criada automaticamente, mesmo que depois novos campos sejam criados, pois esses dados podem ter uma estrutura heterogênea (REDMOND; WILSON, 2012). Como exemplo de modelagem de dados, utilizaremos as informações do Quadro 4.9. Nele são definidos dados como o nome do aluno, o curso e o ano de início.

Quadro 4.9 | Dados exemplo para operações com o MongoDB

```
{"nome": "Joao Silva",
"curso": "Ciência da Computação",
 "anoInicio": 2018
```

Fonte: elaborado pelo autor.

O método *insereAluno()*, no Quadro 4.10, apresenta como é feita a inserção no MongoDB. Esse método faz parte da classe

ConexaoMongoDB do Quadro 4.8. Para esse caso, o método de inserção recebe três parâmetros (nome, curso e ano de ingresso). Para fazer a inserção é preciso criar o registro utilizando a classe *Document*, que possui um construtor que recebe uma *String* referenciando a chave e o valor. Esse segundo parâmetro (valor) é um *Object* (classe superior do Java, sendo que todos os objetos são filhos dessa classe), com isso é possível passar qualquer objeto como valor. Na linha 1 é definido o método que recebe nome, curso e ano de início para ser inserido no banco. A linha 2 apresenta a classe *Document*, recebendo a relação chave/valor, onde são passados o nome do campo e o valor a ser inserido. Com o método *append()* são adicionados novos elementos dento da instância *doc* de *Document*. Na linha 3, utilizando o método *inserOne()* da instância de *MongoCollection* chamada *collection*, é passada a instância de *Document* para fazer a inserção. A classe *Document* pode receber qualquer quantidade de campos, e com ela é possível inserir qualquer combinação de dados.

Quadro 4.10 | Inserção de dados utilizando o MongoDB

```
1.  public void insereAluno(String nome, String curso, int
    anoInicio)
    {
2.     Document doc = new Document("nome", nome).append("curso",
                                    curso).append("anoInicio",
    anoInicio);
3.        collection.insertOne(doc);
    }
```

Fonte: elaborado pelo autor.

Ao inserir um registro em uma coleção, o próprio SGBD cria um campo extra chamado ***_id***, com tamanho de 12 bytes, que é usado pelo software para gerenciamento. Esse campo possui um valor único, ou seja, seu funcionamento pode ser comparado ao da chave primária em um SGBD relacional.

Reflita

Ao fazer a inserção em um banco NoSQL, como o MongoDB, a estrutura de armazenamento se modela com as inserções. Isso pode ser um problema com desenvolvedores sem organização ou sem sincronismo? As metodologias ágeis facilitam esse processo?

O próximo passo é a seleção de dados no banco. No caso do MongoDB é possível utilizar um método para buscar as informações. O Quadro 4.11 apresenta como fazer a seleção de todos os dados da coleção selecionada. A linha 1 do Quadro 4.11 apresenta a definição do método. Na linha 2 é definido um objeto chamado *"cursor"* da classe *MongoCursor*, que é criado a partir da instância de *MongoCollection* (*collection*) utilizando o método *find()* e o método *iterator()*. Se não for passado nenhum parâmetro no método *find()*, são retornados todos os elementos da *collection*. Na linha 3 foi utilizado o método *hasNext()* como uma forma de verificar se existem novos registros dentro do objeto da classe *MongoCursor*. Dentro do **while**, entre as linhas 4 a 8, é utilizado novamente o objeto *cursor* para recuperar o *Document* atual com o método *next()*.

Para cada documento armazenado, é possível acessar seus campos com o método *get()*. Isso é feito utilizando o objeto chamado *"atual"* (da classe *Document*), que busca os itens da *collection* com o método *get()* passando o nome da chave como parâmetro. No final do método em **finally**, o objeto cursor é fechado pelo método *close()*.

Quadro 4.11 | Busca de dados no MongoDB

```
1.  public void exibeAlunos()
    {
2.          MongoCursor<Document> cursor =
    collection.find().iterator();
            try {
3.              while (cursor.hasNext()) {
4.                  Document atual = cursor.next();
5.
    System.out.println(atual.get("_id"));
6.                  System.out.println(atual.get("nome"));
7.                  System.out.println(atual.get("curso"));
8.
    System.out.println(atual.get("anoInicio"));
9.              }
10.         } finally {
11.               cursor.close ();
12.               }
13. }
```

Fonte: elaborado pelo autor.

Em diversos cenários é necessário fazer uma seleção mais específica de dados, e para isso é necessário passar algum parâmetro no momento de executar o método *find()* da classe *MongoCollection*. O Quadro 4.12 apresenta um exemplo de como selecionar um item que deve ser igual a um parâmetro. Repare que no método *find()*, da linha 2, é utilizado um outro método chamado *eq()* para verificar se a chave e o valor são encontrados. O primeiro parâmetro é a chave, e o segundo o valor a ser buscado. Esse elemento é um método estático vindo pelo **import static** com.mongodb.client.model.Filters.*. Note que as inclusões *static* garantem que você possa utilizar os métodos estáticos sem a necessidade de colocar o caminho completo do método. Entre as linhas 4 a 10 são exibidos os dados do registro, e a linha 13 fecha o objeto da classe *MongoCursor*.

Quadro 4.12 | Seleção específica de dados

```
1.  public void exibeAluno(String nome){
2.     MongoCursor<Document> cursor = collection.
    find(eq("nome", nome)).iterator();
3.        try {
4.          while (cursor.hasNext()) {
5.              Document atual = cursor.next();
6.              System.out.println(atual.get("_id"));
7.              System.out.println(atual.get("nome"));
8.              System.out.println(atual.get("curso"));
9.              System.out.println(atual.get("curso"));
10.             System.out.println(atual.get("anoInicio"));
11.         }
12.       } finally {
13.         cursor.close();
14.       }
15. }
```

Fonte: elaborado pelo autor.

No pacote *com.mongodb.client.model.Filters* existem diversos elementos estáticos que são utilizados para fazer a filtragem de dados. O Quadro 4.13 apresenta alguns desses métodos.

Quadro 4.13 | Métodos estáticos para filtragem de dados

Método	Descrição
`eq(java.lang.String fieldName, TItem value)`	Seleciona os dados que forem iguais ao *value* da chave *fieldname*.
`ne(java.lang.String fieldName, TItem value)`	Seleciona os dados que forem diferentes do *value* da chave *fieldname*.
`lt(java.lang.String fieldName, TItem value)`	Seleciona os dados que forem menores que o *value* da chave *fieldname*.
`lte(java.lang.String fieldName, TItem value)`	Seleciona os dados que forem menores ou iguais ao *value* da chave *fieldname*.
`gte(java.lang.String fieldName, TItem value)`	Seleciona os dados que forem maiores ou iguais ao *value* da chave *fieldname*.
`gt(java.lang.String fieldName, TItem value)`	Seleciona os dados que forem maiores que o *value* da chave *fieldname*.

Fonte: elaborado pelo autor.

Exemplificando

O Quadro 4.14 apresenta como selecionar todos os alunos que têm data de início de curso maior ou igual a 2011, e a linha 2 apresenta essa utilização.

Quadro 4.14 | Selecionar os dados utilizando o método gte

```
1.  public void exibeAlunosData(String nome)
    {
2.          MongoCursor<Document> cursor =
    collection.find(gte("dataInicio", 2011)).iterator();
3.          try {
4.                  while (cursor.hasNext()) {
5.                          Document atual = cursor.
    next();
6.
            System.out.println(atual.get("nome"));
7.
            System.out.println(atual.get("curso"));
8.
            System.out.println(atual.get("anoInicio"));
9.                  }
10.         } finally {
11.                  cursor.close ();
12.         }
13. }
```

Fonte: elaborado pelo autor.

Para fazer a alteração dos dados é possível utilizar o método *updateOne()* da classe *MongoCollection*. O Quadro 4.15 apresenta como fazer essa operação. O método *updateOne()* na linha 2, precisa de dois parâmetros: o primeiro diz respeito ao registro que se deseja alterar e o segundo, aos dados que serão usados para alterar. Para filtrar o registro a ser alterado (primeiro parâmetro) é usado o método *eq()*, passando a chave e o valor a ser localizado. Já no segundo parâmetro, um novo objeto *Document* é instanciado, com o argumento *$set*, que informa para alterar o campo atual, e o respectivo valor a ser usado para a alteração.

Quadro 4.15 | Atualizar um dado do MongoDB

```
1.  public void alteraAluno(String nomeAntigo, String
    nomeNovo)
    {
2       collection.updateOne(eq("nome",  nomeAntigo),  new
    Document("$set",
    new             Document("nome", nomeNovo)));
    }
```

Fonte: elaborado pelo autor.

O processo para remoção dos dados é feito de maneira semelhante à da atualização, todavia é passado apenas um parâmetro para buscar e selecionar qual registro remover. O Quadro 4.16 apresenta o código necessário para efetuar a remoção. Na linha 1 é apresentado o método que recebe o nome do aluno e na linha 2, o método *deleteOne()*, que recebe como parâmetro um filtro utilizando o método *eq()*, comparando o parâmetro *nome* com a chave do banco de dados.

Quadro 4.16 | Remoção de dados do MongoDB

```
1.  public void removeAluno(String nome) {
2.          collection.deleteOne(eq("nome", nome));
3.  }
```

Fonte: elaborado pelo autor.

A alteração ou remoção também pode ser feita utilizando o parâmetro "*_id*", que funciona como uma chave primária (valor que não se repete em uma coleção). O Quadro 4.17 apresenta como é feito o processo. Repare que é igual quando utilizado apenas o nome, todavia é passado como parâmetro uma instância da classe *ObjectId*.

Quadro 4.17 | Alteração e remoção utilizando o campo "_id "

```
1.  public void alteraAluno(ObjectId id, String nomeNovo)
    {
2.      collection.updateOne(eq("_id", id), new
    Document("$set", new Document("nome", nomeNovo)));
    }
3.  public void removeAluno(ObjectId id)
    {
4.      collection.deleteOne(eq("_id", id));
    }
```

Fonte: elaborado pelo autor.

Para finalizar esta seção, você vai aprender como usar os métodos para o CRUD, implementados na classe *ConexaoMongoDB*. Primeiro, lembre-se de abrir uma janela de *prompt* e carregar o servidor *mongod* com o comando *"C:\Program Files\MongoDB\Server\4.0\bin\mongod.exe"*. Como um exemplo geral, o Quadro 4.18 apresenta como utilizar a classe de acesso ao MongoDB. A linha 2 faz a conexão com o banco de dados, a linha 32 faz inserção de um aluno, a linha 4 exibe os dados dos alunos (onde é possível ver qual é o "id" de cada objeto), a linha 5 altera um registro e a linha 6 faz a remoção de um registro. Fique atento: quando você inserir um registro, o valor do _id será diferente, e você poderá consultar pelo comando da linha 4.

Quadro 4.18 | Utilização da classe para operações no MongoDB

```
1.  public static void main(String[] args) {
2.      ConexaoMongoDB c = new ConexaoMongoDB();
3.      c.insereAluno("Pedro Silva", "Ciência da
    Computação", 2016);
4.      c.exibeAlunos();
5.      c.alteraAluno(new ObjectId("5b3ec4603af20f20a8e-
    dcb29"), "Pedro Da Silva");
6.      c.removeAluno(new ObjectId("5b3ec4603af20f20a8e-
    dcb29"));
7.  }
```

Fonte: elaborado pelo autor.

Os comandos para manipulação dos dados utilizando MongoDB são mais simples de serem utilizados e não necessitam de uma formalização como os SGDB relacionais. Com eles a forma de acesso se torna mais simples e não são necessários grandes estudos da estrutura da informação.

Pesquise mais

Como outras fontes de informação é possível consultar diversos sites que possuem as informações sobre o *driver* do MongoDB para Java:

- MongoDB driver. Disponível em: <https://mongodb.github.io/mongo-java-driver/>. Acesso em: 6 jul. 2018.
- Documentação do driver 3.8. Disponível em: <http://mongodb.github.io/mongo-java-driver/3.8/>. Acesso em: 6 jul. 2018.
- Descrição da API do driver. Disponível em: <http://mongodb.github.io/mongo-java-driver/3.8/javadoc/overview-summary.html>. Acesso em: 6 jul. 2018.

Sem medo de errar

A empresa em que você trabalha é especializada no desenvolvimento de sistemas que atualizam softwares legados. Na migração de um sistema para universidade, surgiu a necessidade de criar um módulo que analise os dados, com o objetivo de detectar possíveis desistências dos cursos. Como é um sistema que utilizará um grande volume de dados, seu líder achou interessante aplicar um banco de dados NoSQL. Dessa forma sua tarefa é criar uma classe que forneça o acesso a esse novo modelo de banco de dados, contendo os seguintes métodos:

- Acesso ao banco de dados, passando por parâmetro o endereço do computador em que está o banco, nome do banco e coleção a ser utilizada.
- Inserção dos dados contendo nome completo do aluno, idade do aluno, curso que evadiu e data da evasão.
- Seleção e exibição no terminal dos dados.
- Alteração dos dados utilizando como chave o item id.
- Remoção dos dados utilizando como chave o item id.

O Quadro 4.19 apresenta a classe que implementa a forma de acesso ao NoSQL MongoDB. Repare que os dados da conexão foram passados no construtor, e cada ação necessária foi feita em um método em separado.

Quadro 4.19 | Classe de acesso ao NoSQL

```
import static com.mongodb.client.model.Filters.eq;
import org.bson.Document;
import org.bson.types.ObjectId;
import com.mongodb.client.*;
import com.mongodb.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import com.mongodb.client.MongoDatabase;

public class AcessoNoSQL {

    private MongoClient mongoClient;
    private MongoDatabase database;
    private MongoCollection<Document> collection;

    public AcessoNoSQL(String endereBanco, String
nomeBanco, String nomeCollection)
    {
        mongoClient = new MongoClient(endereBanco, 27017);
        database = mongoClient.getDatabase(nomeBanco);
        collection =
database.getCollection(nomeCollection);
    }
    public void insereAluno(String nome, int idade,
String curso,String dataEvasao)
    {
        Document doc = new Document("nome",
nome).append("idade", idade).append("curso",
curso).append("dataEvasao",dataEvasao);
        collection.insertOne(doc);
    }
    public void exibeAlunos()
    {
        MongoCursor<Document> cursor =
collection.find().iterator();
        try {
```

```
            while(cursor.hasNext()) {
                  Document atual = cursor.next();
                  System.out.println(atual.get("_id"));
                  System.out.println(atual.get("nome"));
                  System.out.println(atual.get("idade"));
                  System.out.println(atual.get("curso"));

    System.out.println(atual.get("dataEvasao"));
            }

        } finally {
            cursor.close ();
        }
    }
   public void alteraAluno(ObjectId id, String nome,int
idade, String curso,String dataEvasao)
    {
        collection.updateOne(eq("_id", id), new
Document("$set", new Document("nome", nome).append("idade",
idade).append("curso", curso).append ("dataEvasao",
dataEvasao) ));
    }
```

Fonte: elaborado pelo autor.

A utilização da classe AcessoNoSQL pode ser feita como mostrado no Quadro 4.20.

Quadro 4.20 | Método principal

```
import org.bson.types.ObjectId;

public class Principal {

       public static void main(String[] args) {
            AcessoNoSQL acesso = new AcessoNoSQL
("localhost", "bancoSP", "alunos");
            acesso.insereAluno("Ruy", 30, "computação",
"20/03/2018");
            acesso.exibeAlunos();
       }
}
```

Fonte: elaborado pelo autor.

Com essa implementação, finalizamos a parte de migração da persistência do sistema legado, antes feita por um SGBD relacional, e agora usando um banco de dados NoSQL. Com isso o sistema está apto a utilizar mecanismos mais modernos de tratamento de dados, por exemplo, a partir de inteligência artificial.

Avançando na prática

Busca em banco NoSQL

Descrição da situação-problema

A empresa em que você presta serviço de consultoria implantou um banco de dados NoSQL em um sistema bibliotecário. Nesse sistema, diversas buscas são baseadas em parâmetros do banco de dados, sendo um deles a data de lançamento dos livros. Sua tarefa é construir três métodos para o NoSQL MongoDB que sejam capazes de:

- Exibir os livros com ano de lançamento maior que um valor passado por parâmetro.
- Exibir os livros com ano de lançamento menor ou igual ao valor passado por parâmetro.

Esse banco de dados possui os seguintes campos:

- Nome do livro.
- Autores.
- Editora.
- Ano de lançamento.

Resolução da situação-problema

Para resolver essa questão é necessário utilizar filtros nos métodos de busca para definir quais dados serão exibidos. O Quadro 4.21 apresenta como devem ser os métodos de busca. O método *exibeLivros()* recebe como parâmetro o ano e uma variável para indicar qual operação de busca deve ser utilizada.

Quadro 4.21 | Métodos de busca por data de lançamento

```
import static com.mongodb.client.model.Filters.*;
import org.bson.Document;
import org.bson.types.ObjectId;
import com.mongodb.client.*;
import com.mongodb.MongoClient;

public class AcessoBancoBiblioteca {

// Aqui devem ter os atributos e construtor para
configurar a conexão

     public void exibiLivros(int ano, int op)
     {
          MongoCursor<Document> cursor = null;

          if(op == 1)
                    cursor =
collection.find(gt("anoLancamento",ano)).iterator();
          else
                    cursor =
collection.find(lte("anoLancamento",ano)).iterator();

          try {
              while (cursor.hasNext()) {
                    Document atual = cursor.next();
                    System.out.println(atual.get("_id"));
     System.out.println(atual.get("autores"));
     System.out.println(atual.get("editora"));
     System.out.println(atual.get("anoLancamento"));
               }

          } finally {
              cursor.close ();
          }
     }
}
```

Fonte: elaborado pelo autor.

A partir dos filtros utilizados no Quadro 4.21, o método para exibir os livros nas condições especificadas pode ser utilizado, melhorando a experiência do usuário que utiliza o sistema.

Faça valer a pena

1. Para fazer a conexão com o sistema de gerenciamento de banco de dados NoSQL MongoDB é necessário utilizar uma API fornecida pela empresa que desenvolve esse sistema. Nesse processo de conexão deve-se definir o endereço, o banco e a coleção que será utilizada, considerado os passos:

1. Utilizar a classe *MongoCollection* para definir qual coleção seria utilizada.
2. Com o objeto da classe *MongoCollection*, executar as operações de manipulação dos dados.
3. Definir o nome do banco a ser utilizado com a classe *MongoDatabase*.
4. Escolher o endereço e a porta de conexão com o Classe *MongoClient*.

Selecione a opção que define a correta sequência de passos para fazer uso do MongoDB em Java:

a) 4 – 3 – 1 – 2.
b) 3 – 4 – 1 – 2.
c) 4 – 3 – 2 – 1.
d) 1 – 2 – 3 – 4.
e) 1 – 4 – 2 – 3.

2. O trabalho com um banco de dados NoSQL difere de um sistema de gerenciamento de banco de dados relacional no que diz respeito à forma de manipulação dos dados. No NoSQL não é necessário um formalismo como a linguagem SQL, pois é possível fazer o acesso de forma mais simples e fornecido pela API do NoSQL.

Sobre as formas de seleção de dados utilizando o MongoDB, é correto afirmar:

a) Em banco de dados NoSQL não é possível selecionar os dados.
b) O método *find()*, da classe *MongoCollection*, recebe um parâmetro que determina o máximo de registro a retornar.
c) A busca de dados é feita pelo método *find()* da classe *MongoBD*, e como parâmetro são passados os resultados dos métodos do pacote *com.mongodb.client.model.Filters*.
d) A busca de dados é feita pelo método find(), da classe *MongoCollection*, e como parâmetro são passados os resultados dos métodos do pacote com.mongodb.client.model.Filters.
e) O método *find()* da classe *MongoCollection* não recebe parâmetros, dessa forma busca sempre retornar todos os dados.

3. Ao se utilizar a API do MongoDB para manipular os dados é possível fazer alteração, inserção, seleção e remoção dos dados. Essas operações devem ser utilizadas considerando a documentação do projeto. Nesse contexto, avalie as afirmações a seguir:

I. Ao fazer a inserção de dados, deve ser seguida a mesma estrutura definida na configuração inicial, pois os dados devem ter uma estrutura homogênia.
II. O método *insertOne()* da classe MongoCollection recebe como parâmetro uma instância da classe Document, com isso é possível enviar os dados.
III. Para executar a alteração dos dados utilizando o método *updateOne()*, da classe MongoCollection, é necessário definir um filtro para informar qual registro alterar.
IV. Ao remover um dado do MongoDB, todos os registros devem receber a atualização em seus índices.
V. O campo id dos registros do MongoDB são úteis para escolher, de forma única, os dados.

Quais das afirmações são corretas?

a) Apenas as afirmações I, II e V.
b) Apenas as afirmações I, II, III e V.
c) Apenas as afirmações II, III e V.
d) Apenas as afirmações II e III.
e) Apenas as afirmações II e IV.

Referências

GESSERT, F. et al. NoSQL database systems: a survey and decision guidance. **Computer Science - Research And Development**, [s.l.], v. 32, n. 3-4, p. 353-365, 3 nov. 2016. Springer Nature. Disponível em: <http://dx.doi.org/10.1007/s00450-016-0334-3>. Acesso em: 8 ago. 2018.

HAN, J.; E, H.; LE, G. Survey on NoSQL database. In: INTERNATIONAL CONFERENCE ON PERVASIVE COMPUTING AND APPLICATIONS, 6., 2011, Port Elizabeth, v. 1, n. 1, p. 363-366, out. 2011.

HEUSER, C. A. **Projeto de banco de dados**. 4. ed. Porto Alegre: Bookman, 2011. 282 p.

LEAVITT, N. Will NoSQL Databases Live Up to Their Promise? **Computer**, Washington, v. 43, n. 2, p. 12-14, fev. 2010.

MONIRUZZAMAN, A. B. M.; HOSSAN, S. A. NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. **International Journal Of Database Theory And Application**, Sandy Bay, v. 6, n. 4, p.1-13, 2013.

PLUGGE, E.; MEMBREY, P.; HAWKINS, T. **The Definitive Guide to MongoDB**. New York: Apress, 2010. 328 p.

REDMOND, E.; WILSON, J. **Seven Databases in Seven Weeks:** A Guide to Modern Databases and the NoSQL Movement. Strickland: Pragmatic Bookshelf, 2012. 354 p.

ISBN 978-85-522-1166-2