



Edição 51

DESAFIO: Blog com MongoDB e Spark
Iniciando no Java por um novo caminho

Interpretando Diagramas
de Classes da UML
Desenvolvendo a camada
de modelo na prática



ANÁLISE ESTÁTICA

Simplifique a detecção
e a correção de bugs

ISSN 2179625-4



DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

FORMAÇÃO DESENVOLVEDOR JAVA

DOMINE O DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETO COM **UML** E APRENDA TÓPICOS AVANÇADOS DE PROGRAMAÇÃO COMO O USO DE **SERVLETS, JSP, FRAMEWORKS WEB DENTRO DO PADRÃO MVC.**

➤ A linguagem Java está presente em complexos sistemas como em aplicativos mobile e desktop, servindo a milhares de pessoas. Inicie sua carreira e certifique-se no instituto que valoriza seu currículo. Faça Infnet!

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer



Sumário

Artigo no estilo Solução Completa

06 - Aprenda a interpretar Diagramas de Classes da UML – Parte 2

[Everson Mauda]

Conteúdo sobre Boas Práticas

16 - Como adotar a análise estática de código

[Daniel Medeiros de Assis]

Artigo no estilo Solução Completa

26 - Criando um blog com MongoDB e Spark Framework

[Diego Travassos Balduini]

CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



CONHEÇA ALGUNS DOS CURSOS:

- Curso de noSQL (Redis) com Java
- Curso Básico de JDBC
- Java Básico: Aplicações Desktop
- JSF com Primefaces
- Conhecendo o Apache Struts

Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Edição 51 • 2015 • ISSN 2173625-4



Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Diogo Souza (diagosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse www.devmedia.com.br/central, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Easy Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

CURSOS ONLINE

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Feita para Desenvolvedores de Software e DBAs



CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :

<http://www.devmedia.com.br/cursos/banco-de-dados>
(21) 3382-5038



DEVMEDIA



CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

DESENVOLVA COM AS FERRAMENTAS QUE OS MELHORES DESENVOLVEDORES
USAM: GIT, REDMINE, JENKINS, SONAR, MAVEN E MUITO MAIS.

A suite ToolsCloud é um conjunto integrado das melhores ferramentas para desenvolvimento de software do mercado.

Um ambiente pronto para usar e com suporte, que os desenvolvedores curtem e os gerentes se surpreendem.

- Aumenta a produtividade;
- Facilita a colaboração e comunicação;
- Maior visibilidade e acompanhamento;
- Melhor planejamento e mais agilidade.

Na suite ToolsCloud você tem a sua disposição:

- As ferramentas mais usadas, integradas e de uso imediato;
- Repositório de código (git e svn), integração contínua (Jenkins/Hudson), Gerência de pendências (Redmine), e mais Maven, Nexus, Sonar;
- Infraestrutura robusta, em ambiente de nuvem, suporte e segurança;
- Facilita práticas de agilidade e lean startups;
- Ambiente isolado, customizável e flexível;
- Colaboração garantida: wiki, fóruns de discussão, chat (IRC).

Acompanhe a Toolscloud:

 toolscloud@toolscloud.com



twitter.com/toolscloud



ToolsCloud
toolscloud.com

Aprenda a interpretar Diagramas de Classes da UML – Parte 2

Como desenvolver a camada Model de um software para tickets aéreos a partir da interpretação de um diagrama de classes da UML

ESTE ARTIGO FAZ PARTE DE UM CURSO

Na primeira parte deste artigo foram apresentados os conceitos relacionados à interpretação de um diagrama de classes UML. Foi visto que uma associação entre classes contém quatro características básicas: **Navegabilidade**, que define a direção de uma associação, podendo ser unidirecional ou bidirecional; **Multiplicidade**, que indica quantos objetos de uma classe destino estão presentes em uma classe origem; **Conectividade**, que é a união dos dois sentidos da Navegabilidade, formando a representação da leitura da associação, que pode ser do tipo *OneToOne*, *OneToMany*, *ManyToOne* e *ManyToMany*; e **Obrigatoriedade**, que indica a necessidade básica de uma classe para ser instanciada como, por exemplo, para criar uma instância de uma Casa é necessário que exista um Terreno. Além dos aspectos básicos de uma associação, foram apresentados conceitos sobre as camadas Model, Business, View e DAO, que ajudam a organizar a arquitetura de um software.

Com base nisso, nesta segunda parte do artigo iremos realizar a codificação do diagrama de classes apresentado no próximo tópico. Deste modo, serão assimilados os conceitos abordados na parte um com a implementação de cada classe projetada no diagrama. Antes, porém, vale ressaltar que o mesmo possui várias situações que certamente serão encontradas em

Fique por dentro

Independentemente da área de atuação no desenvolvimento de software, o diagrama de classes possui muitas informações que podem estar ocultas sob seus elementos para pessoas que não estão treinadas em observá-las. E o problema principal é que muitas vezes essas informações podem passar despercebidas não só por desenvolvedores, mas também por analistas de negócio, que são os responsáveis pela construção desses diagramas e não sabem trabalhar mais profundamente sobre eles. Vale ressaltar que é muito importante a correta interpretação deste para gerar classes e atributos corretos e, principalmente, para a utilização dos construtores das classes na construção das regras de negócio de um software a partir das obrigatoriedades existentes nas associações entre as classes. Com base nisso, este artigo visa explorar a interpretação desses aspectos com um exemplo prático de um sistema de compras de bilhetes aéreos, cujo projeto Java será gerado a partir de um diagrama de classes.

outros diagramas, o ajudando a se preparar para os desafios enfrentados diariamente na carreira de um desenvolvedor profissional.

O diagrama de classes do sistema de bilhetes aéreos

Além do texto de requisitos do sistema de Bilhetes Aéreos, o título de melhor visualização dos requisitos, o analista de sistema do projeto deixou um diagrama de classes UML para ser codificado na linguagem Java, conforme a **Figura 1**. Esse diagrama contém 14 classes e três enums, que serão explicitados com mais detalhes durante a parte prática deste artigo. Antes disso, é interessante que você o observe e tente imaginar como seria a codificação das clas-

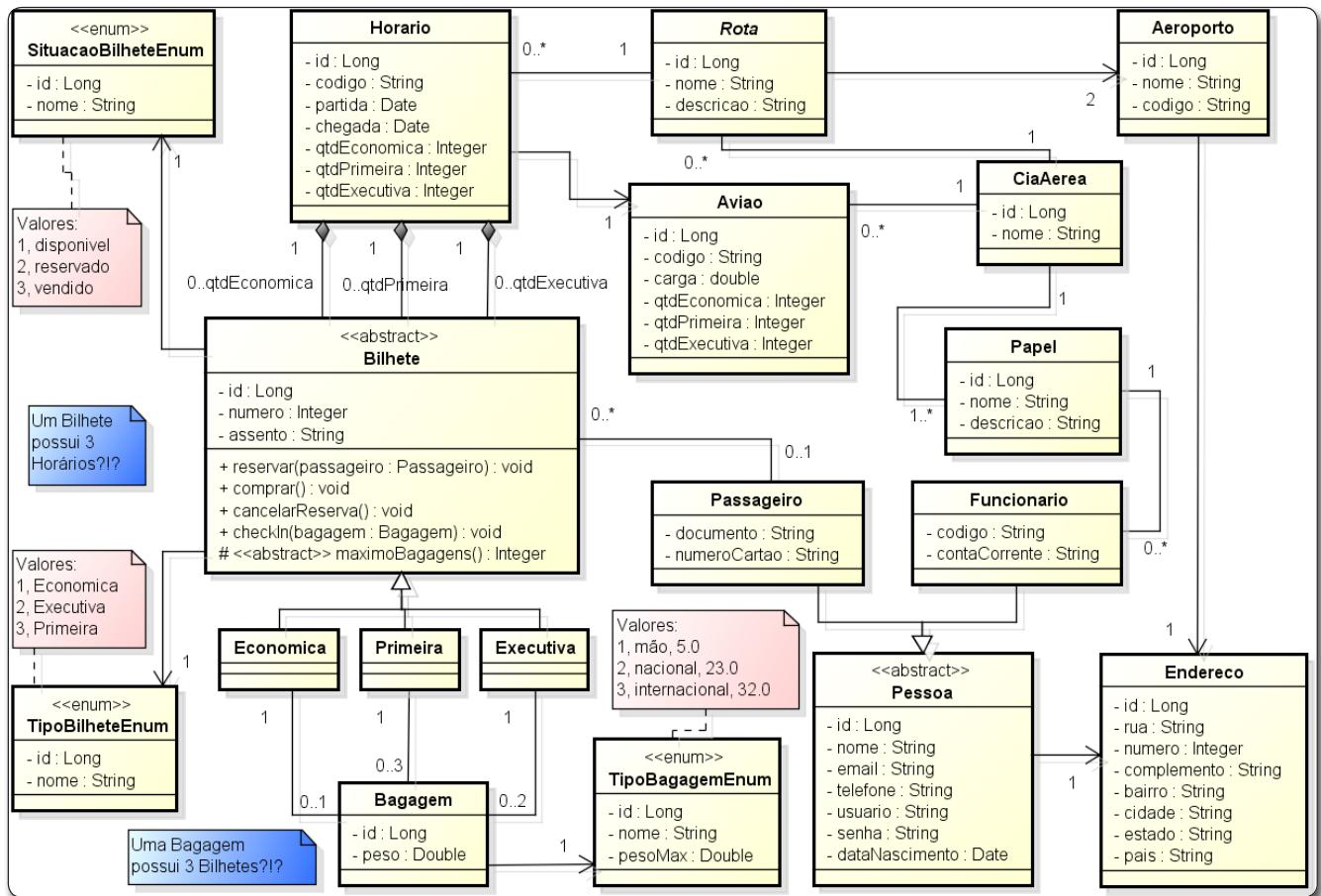


Figura 1. Diagrama de classes representando o sistema de bilhetes aéreos

ses, enums e suas associações. Além disso, também é interessante que você tente antecipar problemas que podem aparecer durante a codificação. Isto certamente lhe ajudará a entender melhor o conteúdo apresentado nos próximos tópicos.

Codificação da camada Model

Além da qualidade em um projeto de software, é válido destacar que a codificação de um diagrama de classes é uma tarefa que exige uma atenção muito grande. Nesta etapa é interessante lembrar de um ditado bastante conhecido na área da computação: Dividir para Conquistar. Esse “lema” sinaliza que quando existe um grande problema, por exemplo, codificar o diagrama de classes desse projeto, a melhor forma de resolvê-lo é através da sua divisão por partes. Sendo assim, vamos dividir o nosso diagrama de classes para tornar mais fácil a sua codificação.

Codificação da classe Endereco

A primeira atividade a ser realizada sobre um diagrama de classes UML é a busca por classes que sejam somente de destino, ou seja, que possuam somente associações unidirecionais com outras classes e que todas essas associações terminem nela (a classe destino). No caso do diagrama do projeto, a classe

Endereco, situada no canto inferior direito, satisfaz essa condição. Repare que existem duas associações com a classe **Endereco** (**Pessoa** e **Aeroporto**), mas em ambas, **Endereco** é a classe destino. Assim, essa classe é uma excelente candidata para iniciar a codificação da camada model, conforme a **Listagem 1**.

Em Java, a divisão por camadas é auxiliada pelos pacotes (*packages*). Com o intuito de manter essa padronização/organização, para o nosso projeto de tickets aéreos iremos criar um pacote para a camada model com o nome **br.com.devmedia.bilhetesAereos.model**.

Listagem 1. Código da classe Endereco.

```

01 package br.com.devmedia.bilhetesAereos.model;
02
03 public class Endereco {
04     private Long id;
05     private String rua, complemento, bairro, cidade, estado, pais;
06     private Integer numero;
07
08     //não é necessário construtor default
09
10    //Métodos getters e setters para todos os atributos
11
12 }
```

Como é possível perceber, todos os atributos do diagrama de classes foram representados utilizando a palavra-chave **private**. Isto se deve ao fato de no diagrama de classes esses atributos estarem representados com um sinal de menos ("‐"). Portanto, quando houver esse símbolo na frente de atributos ou métodos em um diagrama, ele sempre será **private**. Caso o símbolo seja um sinal de adição ("+") esse atributo ou método deve ser codificado como **public**. E caso o símbolo seja um sinal de Hash ("#"), esse atributo ou método deve ser codificado como **protected**.

Essa definição também tem relação com o conceito da orientação a objetos chamado encapsulamento. Tal conceito declara que todos os atributos de uma classe devem estar protegidos de outras classes. Uma forma de acesso a esses atributos seria através dos métodos getters e setters.

Como exposto na parte teórica, não existe na classe **Endereco** atributos do tipo **Pessoa** ou **Aeroporto**, pois as associações com a classe **Endereco** são unidirecionais, sendo **Endereco** a classe destino. Por fim, não implementamos um construtor, pois o Java cria automaticamente o construtor **default**, sem argumentos. Neste caso, a não codificação de um construtor (assim como a simples codificação de um construtor sem parâmetros) está correta, pois pela característica da Obrigatoriedade não existe nenhuma classe que obrigue **Endereco** a receber alguma instância dessa classe.

Implementada a classe **Endereco**, existe mais alguma que seja uma classe Destino em nosso diagrama? Não! Portanto, podemos avançar para a próxima atividade relacionada ao diagrama de classes.

Codificação do enum **TipoBagagemEnum**

Nesse momento é hora de verificar se existem mais classes “independentes”, ou seja, classes que não dependam de outras classes que ainda não foram codificadas para o serem. No caso do projeto de Bilhetes Aéreos, existem as Enums, **SituacaoBilheteEnum**, **TipoBilheteEnum** e **TipoBagagemEnum**.

É importante lembrar que o tipo **Enum** não é uma classe, mas sim um tipo de artefato, como as interfaces. Por isso a codificação desse tipo segue alguns padrões próprios. O primeiro é a sua declaração, que sempre será **public enum nomeEnum**, mas que não é necessário escrever, visto que a própria JVM insere estas palavras reservadas em tempo de compilação. O segundo ponto é a declaração das constantes do **enum**. Elas sempre serão **public static final** e são inicializadas quando ocorre a primeira invocação ao **enum**, durante a execução da JVM. Essas constantes se utilizam de construtores próprios, que recebem os valores no momento da sua invocação e os atribuem aos seus tipos internos correspondentes. Tais atributos, para serem acessados por outras classes, devem possuir métodos getters.

Como exemplo, vamos codificar o enum **TipoBagagemEnum**. O diagrama de classes apresenta três atributos para ele: **id**, **nome** e **pesoMax**, que são codificados conforme a **Listagem 2**, nas linhas 8 a 10. Além disso, devemos codificar os métodos getters desses atributos, o que ocorre nas linhas 18 a 28. Para auxiliar na criação das constantes do **enum**, linhas 4 a 6, o diagrama de

classes possui um comentário, indicando quais são os valores que os atributos devem possuir para cada constante.

A ordem desses valores ajuda na criação do construtor, pois define a ordem dos parâmetros. Seguindo esse pensamento e observando o comentário para o enum **TipobagagemEnum**, este terá um construtor que possui os parâmetros **Long**, **String** e **Double**.

Listagem 2. Código do enum **TipoBagagemEnum**.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 public enum TipoBagagemEnum {
04     MAO(1, "mao", 5.0),
05     NACIONAL(2, "nacional", 23.0),
06     INTERNACIONAL(3, "internacional", 32.0);
07
08     private Long id;
09     private String nome;
10     private Double pesoMax;
11
12     private TipoBagagemEnum(long id, String nome, double pesoMax) {
13         this.id = id;
14         this.nome = nome;
15         this.pesoMax = pesoMax;
16     }
17
18     public Long getId() {
19         return id;
20     }
21
22     public String getNome() {
23         return nome;
24     }
25
26     public double getPesoMax() {
27         return pesoMax;
28     }
29 }
```

Os outros enums deverão ser criados de modo semelhante ao **TipoBagagemEnum**, respeitando seus atributos internos e os comentários existentes no diagrama de classes.

Codificação das classes **Aeroporto** e **Pessoa**

A próxima atividade sobre o diagrama de classes é, a partir das classes destino codificadas (no caso a classe **Endereco** e os enums), observar a “vizinhança” dessas classes, a fim de procurarmos classes que possuam poucas associações, ou ainda, classes destino de outras associações. No diagrama da **Figura 1** existem duas classes em potencial para serem codificadas: **Aeroporto** e **Pessoa**. Ambas, assim como **Endereco**, são classes destino de outras associações, mas com a diferença que originam a associação com **Endereco**. Vamos começar pela classe **Aeroporto**.

Esta classe é a origem de uma associação unidirecional com **Endereco**. Assim, o atributo **Endereco** deverá ser representado na classe **Aeroporto** conforme a linha 7 da **Listagem 3**. Um ponto interessante aqui e já comentado na parte teórica é que temos uma associação unidirecional um para um com obrigatoriedade. Deste modo, **Aeroporto** deve receber uma instância de **Endereco** em seu

construtor, visto que o limite inferior da associação é o cardinal 1, como verificado na linha 9 da **Listagem 3**.

Listagem 3. Código da classe Aeroporto.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 public class Aeroporto {
04     private Long id;
05     private String nome, codigo;
06
07     private Endereco endereço;
08
09     public Aeroporto(Endereco endereço){
10         this.endereço = endereço;
11     }
12
13     //Métodos getters e setters para todos os atributos básicos da classe
14
15     public Endereco getEndereço(){
16         return endereço;
17     }
18 }
19 }
```

Neste momento você pode estar se perguntando: “Poderia existir mais de um construtor para a classe **Aeroporto**?” Até poderia, mas todos, independentemente da quantidade de construtores, devem ter como parâmetro uma instância de **Endereco**.

Outro questionamento é se o construtor deve verificar se a instância de **Endereco** está nula. Em nosso exemplo essa verificação não foi realizada porque estamos confiando que o software sempre irá passar objetos instanciados ao chamar o construtor da classe **Aeroporto**. Pode ser que no seu projeto seja necessário realizar esta validação, por talvez receber dados antigos que podem estar incompletos ou ainda ser um software que recebe requisições de outros softwares, o qual não foi codificado por você ou sua equipe. Nestes casos pode ser recomendado codificar uma validação de instâncias nulas no construtor.

Por fim, não é necessário criar o método **setEndereco(Endereco)**, já que este deve ser passado via construtor, evitando assim que a instância de **Endereco** possa ser alterada. Como ainda existe o método **getEndereco()**, somente os atributos internos de **Endereco** poderão ser alterados.

A outra classe que está na vizinhança de **Endereco** é a classe **Pessoa**. Essa classe possui uma associação unidirecional obrigatória semelhante a **Aeroporto**. Dessa forma é necessário que seja codificado um atributo que represente uma instância de **Endereco** dentro da classe **Pessoa**.

Do mesmo modo que na classe **Aeroporto**, o construtor da classe **Pessoa** deverá receber como parâmetro uma instância de **Endereco**. Além disso, não é necessário criar um método **setEndereco(Endereco)**, pelo mesmo motivo citado anteriormente. E os atributos da classe **Pessoa** devem ser criados conforme o encapsulamento descrito no diagrama. Por fim, podemos verificar que a classe **Pessoa** possui o estereótipo **abstract** sobre o seu nome. Sendo assim, ela deve ser abstrata (vide **Listagem 4**).

Listagem 4. Código da classe Pessoa.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 import java.util.Date;
04
05 public abstract class Pessoa {
06     private Long id;
07     private String nome, email, telefone, usuario, senha;
08     private Date dataNascimento;
09
10     private Endereco endereço;
11
12     public Pessoa(Endereco endereço){
13         this.endereço = endereço;
14     }
15
16     //Métodos getters e setters para todos os atributos básicos da classe
17
18     public Endereco getEndereço(){
19         return endereço;
20     }
21 }
```

Codificação das classes **CiaAerea** e **Passageiro**

Nesse momento não há mais nenhuma classe independente. Portanto, é necessário realizar uma análise sobre as classes restantes, procurando aquelas que não possuam associações obrigatórias com classes que ainda não foram codificadas. Neste quesito, existem duas classes que se enquadram: **CiaAerea** e **Passageiro**.

A classe **CiaAerea** possui três associações com multiplicidade de muitos objetos. As classes destino destas associações são **Rota**, **Aviao** e **Papel**, sendo a última obrigatória, ou seja, a classe **CiaAerea** necessita de, ao menos, uma instância da classe **Papel** para ser construída. Mas há algo estranho aqui. O leitor mais atento notará que foi mencionado que deveriam ser procuradas as classes que não possuíssem associações obrigatórias e a classe **CiaAerea** possui uma associação obrigatória com a classe **Papel**.

Mas, vamos olhar o outro lado dessa associação. A associação da classe **Papel** com a classe **CiaAerea** também é uma associação obrigatória que possui a necessidade de uma instância de **CiaAerea** para ser criada. Então, que implementação vem primeiro?

Caso sigamos à risca a regra da obrigatoriedade, o nosso código do construtor de **CiaAerea** deveria receber uma instância de **Papel** e o construtor de **Papel** deveria receber uma instância de **CiaAerea**. Mas, como seria a instanciação de uma classe **CiaAerea**? Vejamos o código a seguir:

```
CiaAerea ciaAerea = new CiaAerea();
```

Entretanto, está faltando uma instância de **Papel** ser passada para esse construtor. Como também não temos nenhuma instância de **Papel** construída, vamos chamar o **new** para criar uma instância de **Papel** antes da linha que instancia a **CiaAerea**:

```
Papel papel = new Papel();
CiaAerea ciaAerea = new CiaAerea(papel);
```

Agora a CiaAerea está resolvida. Contudo, está faltando uma instância de **CiaAerea** ser passada para a construção da instância de **Papel**. Assim, vamos codificar uma nova linha com um **new** para criar uma instância de **CiaAerea** antes da linha que instancia **Papel**:

```
CiaAerea ciaAerea = new CiaAerea();
Papel papel = new Papel(ciaAerea);
CiaAerea ciaAerea1 = new CiaAerea(papel);
```

E agora, funcionou? Não! Perceba que voltamos ao problema inicial. Está faltando uma instância de **Papel**. Por esse exemplo é possível perceber que nesses casos de dupla obrigatoriedade em uma associação bidirecional, é necessário que um lado “ceda” a obrigação.

Mas como é o processo de inferir qual lado poderia ceder? O processo consiste em analisar o escopo do projeto. Vejamos esse caso de **CiaAerea** e **Papel**. O que é mais importante existir primeiro: uma **CiaAerea**, que terá após a sua criação vários papéis, ou um **Papel**, que pertenceria a uma **CiaAerea**?

Fácil, não? Uma **CiaAerea** tem um peso muito maior que um **Papel** para a criação inicial do escopo do projeto. Assim, o lado do construtor da **CiaAerea** deverá ceder para que não receba uma instância de **Papel**.

Além dessa observação sobre o construtor da classe **CiaAerea**, lembre-se que ela deve representar no código as associações com as classes **Rota**, **Aviao** e **Papel**, o que foi feito através de listas, que por sua vez adotaram *Generics*, para indicar o tipo de objeto que elas aceitam. Lembre-se ainda que essas listas devem ser inicializadas no construtor de **CiaAerea**.

Após codificar essas associações, no entanto, irão ocorrer alguns erros de compilação na classe **CiaAerea**. Apesar disso, não se preocupe com esses erros, pois ao criarmos as outras classes eles serão resolvidos.

Uma última ressalva a respeito das listas é que normalmente os desenvolvedores criam *getters* e *setters* para elas. No entanto, isso não é muito interessante, pois as classes externas teriam acesso ao método setter de uma lista e poderiam modificar a sua instância, passando uma nova instância ou até mesmo um ponteiro nulo. Assim, normalmente costumamos criar apenas um método getter e um método **add()**, que adiciona uma instância do objeto na lista correspondente. A **Listagem 5** mostra o código da classe **CiaAerea**.

Repare na listagem que para o método relacionado à inserção de um avião na lista não foi especificado um nome mais completo, como **addAviao()**, mas sim apenas como **add()**, recebendo como parâmetro o tipo do objeto que queremos armazenar. Os métodos **add()** que implementamos apresentam uma importante característica da Orientação a Objetos: o polimorfismo. Essa característica define que o método que será chamado pelo compilador será decidido em tempo de execução. Note que mais uma vez não foram inseridas verificações de ponteiros nulos, mas isso poderá ser adicionado caso o desenvolvedor deseje.

Listagem 5. Código da classe **CiaAerea**.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 import java.util.List;
04 import java.util.ArrayList;
05
06 public class CiaAerea {
07     private Long id;
08     private String nome;
09
10     private List<Aviao> avioes;
11     private List<Papel> papeis;
12     private List<Rota> rotas;
13
14     //Lembrando que esse lado cedeu a obrigatoriedade
15     public CiaAerea(){
16         this.avioes = new ArrayList<Aviao>();
17         this.papeis = new ArrayList<Papel>();
18         this.rotas = new ArrayList<Rota>();
19     }
20
21     //Métodos getters e setters para todos os atributos básicos da classe
22
23     public List<Aviao> getAvioes() {
24         return this.avioes;
25     }
26
27     public void add(Aviao aviao) {
28         this.avioes.add(aviao);
29     }
30
31     // Os métodos get e add para os atributos papeis e rotas devem
32     // seguir o mesmo padrão.
33 }
```

Voltando à análise realizada no início desse tópico, devemos criar a classe **Passageiro**. Ao observar o diagrama de classes mais uma vez, constatamos que existe uma associação um pouco diferente entre **Passageiro** e **Pessoa**, com um triângulo branco na ponta. Esse símbolo representa uma Herança. Sendo assim, a classe **Passageiro** deve ser implementada como filha de **Pessoa**, utilizando para isso a palavra-chave **extends**.

O conceito de Herança da Orientação a Objetos indica que todos os métodos de uma classe que sejam do tipo **public** ou **protected** serão herdados pelas classes filhas desta. Mas a definição da linguagem Java é um pouco diferente para construtores, pois não permite que os construtores da classe pai sejam herdados pelas classes filhas. No caso da classe **Pessoa**, existe um construtor que implementa uma regra de negócio de obrigatoriedade. Assim, para que a obrigatoriedade continue a existir nas classes filhas de **Pessoa**, essa regra também deve ser implementada nestas classes. Portanto, o construtor que recebe uma instância de **Endereco**, da classe **Pessoa**, deve ser repetido na classe filha. Além disso, como não há um construtor default na classe pai, **Pessoa**, o construtor da classe pai deve ser chamado através do método **super(endereco)**, conforme a linha 12.

Outro item importante é a representação da associação zero ou muitos com a classe **Bilhete**, que ainda não foi codificada e que por isso provocará alguns erros de compilação. Lembre-se ainda que

não iremos criar o método set para a lista de instâncias da classe **Bilhete** e sim um método **add()**, seguindo os mesmos conceitos aplicados aos métodos das listas da classe **CiaAerea**. O código da classe **Passageiro** é apresentado na **Listagem 6**.

Listagem 6. Código da classe Passageiro.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 import java.util.List;
04 import java.util.ArrayList;
05
06 public class Passageiro extends Pessoa {
07     private String documento, numeroCartao;
08
09     private List<Bilhete> bilhetes;
10
11     public Passageiro(Endereco endereco){
12         super(endereco);
13         this.bilhetes = new ArrayList<Bilhete>();
14     }
15
16     //Métodos getters e setters para todos os atributos básicos da classe
17     //Lembrando que já existem métodos getters e setters na classe Pessoa
18
19     public List<Bilhete> getBilhetes() {
20         return this.bilhetes;
21     }
22
23     public void add(Bilhete bilhete) {
24         this.bilhetes.add(bilhete);
25     }
26 }
```

Codificação das classes Aviao e Rota

Com a classe **CiaAerea** codificada, é possível agora codificar as classes **Aviao** e **Rota**, pois as dependências para a classe **CiaAerea** foram preenchidas. Começando pela classe **Aviao**, o diagrama de classes mostra uma associação com a classe **CiaAerea**, sendo esta obrigatória, ou seja, um **Aviao** deve pertencer a uma **CiaAerea**. Assim é necessário que o construtor de **Aviao** receba uma **CiaAerea**.

Além disso, é uma associação bidirecional, logo é necessário que a instância de **CiaAerea** possua o conhecimento sobre a nova instância de **Aviao**, conforme a linha 13 da **Listagem 7**, que obtém a lista de aviões da instância de **CiaAerea** e adiciona a esta ela mesma, mantendo deste modo a bidirecionalidade entre as classes. Como já descrito em outras classes, não é interessante que **Aviao** tenha um método **setCiaAerea()**, já que essa informação é passada em seu construtor.

Com a classe **Aviao** pronta, a próxima classe a ser codificada será a classe **Rota**. Esta possui três associações, sendo duas obrigatorias (**Aeroporto** e **CiaAerea**) e outra não (**Horario**). Uma associação obrigatória exige que seja passada no construtor da classe a instância da classe destino. Neste caso, o que você acha que deve ser feito: ter dois construtores, um recebendo **Aeroporto** e outro recebendo **CiaAerea**, ou ter apenas um construtor que receba ambas as classes?

Listagem 7. Código da classe Aviao.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 public class Aviao {
04     private Long id;
05     private String codigo;
06     private Double carga;
07     private Integer qtdEconomica, qtdExecutiva, qtdPrimeira;
08
09     private CiaAerea ciaAerea;
10
11     public Aviao(CiaAerea ciaAerea){
12         this.ciaAerea = ciaAerea;
13         this.ciaAerea.getAvioes().add(this);
14     }
15
16     //Métodos getters e setters para todos os atributos básicos da classe
17
18     public CiaAerea getCiaAerea(){
19         return ciaAerea;
20     }
21 }
```

Para chegar a essa resposta, vamos relembrar o que a definição de obrigatoriedade nos diz: uma classe deve receber a instância de outra para realizar a operação de atribuição da instância recebida ao campo interno da classe recebedora. Está lembrado do exemplo da casa e do terreno? Se casa não receber uma instância de terreno, onde é que ela será construída? Dessa forma, não importa se existem duas associações ou 10 associações obrigatorias. Todas elas têm que estar no construtor da classe como parâmetros, pois esse é o único ponto comum para a instanciação da classe.

Dito isso, concluímos que a classe **Rota** terá um construtor no qual serão respeitadas todas as obrigações e cardinalidades das associações; nesse caso, dois aeroportos e uma **CiaAerea**. Como a associação com **CiaAerea** é bidirecional, **CiaAerea** deve ter conhecimento da nova **Rota**, conforme mostra a linha 19 da **Listagem 8**. Por fim, deve ser realizada a declaração dos atributos básicos e associações, reforçando que não há métodos setters para os atributos que representam as associações, evitando com isso a troca da instância.

Codificação das classes Papel e Funcionario

Além de permitir a codificação das classes **Aviao** e **Rota**, a codificação da classe **CiaAerea** nos permite criar a classe **Papel**. Pelo diagrama de classes, **Papel** possui duas associações: uma com **CiaAerea** e outra com **Funcionario**. Na associação com **CiaAerea**, foi necessário ceder o lado da **CiaAerea**, para que esta não receba uma instância de **Papel**. Assim, **Papel** deverá seguir a obrigatoriedade, recebendo no construtor uma instância de **CiaAerea**. No entanto, além de receber a instância, ela deverá também realizar a atribuição dela mesma para a instância de **CiaAerea**, para que a obrigatoriedade do lado da **CiaAerea** ocorra e a bidirecionalidade fique representada corretamente.

Dessa forma, dentro do construtor da classe **Papel** deve haver uma atribuição do valor da nova instância da classe **Papel** na lista de papéis da instância de **CiaAerea**, como demonstra a linha 15 da **Listagem 9**.

Aprenda a interpretar Diagramas de Classes da UML – Parte 2

Listagem 8. Código da classe Rota.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 import java.util.List;
04 import java.util.ArrayList;
05
06 public class Rota {
07     private Long id;
08     private String nome, descricao;
09
10    private CiaAerea ciaAerea;
11    private Aeroporto origem, destino;
12    private List<Horario> horarios;
13
14    public Rota(CiaAerea ciaAerea, Aeroporto origem, Aeroporto destino){
15        this.ciaAerea = ciaAerea;
16        this.origem = origem;
17        this.destino = destino;
18        this.horarios = new ArrayList<Horario>();
19        this.ciaAerea.getRotas().add(this);
20    }
21
22    //Métodos getters e setters para todos os atributos básicos da classe
23
24    public CiaAerea getCiaAerea() {
25        return ciaAerea;
26    }
27
28    public Aeroporto getOrigem() {
29        return origem;
30    }
31
32    public Aeroporto getDestino() {
33        return destino;
34    }
35
36    public List<Horario> getHorarios() {
37        return horarios;
38    }
39
40    public void add(Horario horario){
41        this.horarios.add(horario);
42    }
43 }
```

Ademais, **Papel** deverá ter representado os atributos básicos da classe e das associações a que esta referencia. Note, mais uma vez, que não há métodos setter para as associações.

Vamos implementar agora a classe **Funcionario**. Esta classe é filha da classe **Pessoa**, a qual deve respeitar a obrigatoriedade da associação com a classe **Endereco**. Além disso, existe uma associação com a classe **Papel** que é obrigatória, devendo o construtor de **Funcionario** receber também uma instância de **Papel**. Como a associação entre as classes **Funcionario** e **Papel** é bidirecional, a dupla referência entre as classes será realizada. E assim como ocorreu com as outras classes, o método **setPapel()** não será implementado. A classe completa está descrita na **Listagem 10**.

Codificação da classe Horario

A classe **Horario** possui cinco associações, mas apenas duas delas são obrigatórias. As associações entre **Rota** e **Aviao** deverão estar contempladas como parâmetros no construtor e para a

instância de **Rota** deve ser realizada a dupla referência. Outro ponto é que existe uma composição nas associações entre **Horario** e **Bilhete**. Essa composição indica que quando um **Horario** existe, ele é composto por vários bilhetes. E quando esse **Horario** deixa de existir, os bilhetes também o deixarão de existir. Por esse conceito, ela possui uma característica de obrigatoriedade.

Listagem 9. Código da classe Papel.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 import java.util.List;
04 import java.util.ArrayList;
05
06 public class Papel {
07     private Long id;
08     private String nome, descricao;
09
10    private CiaAerea ciaAerea;
11    private List<Funcionario> funcionarios;
12
13    public Papel(CiaAerea ciaAerea){
14        this.ciaAerea = ciaAerea;
15        this.funcionarios = new ArrayList<Funcionario>();
16        this.ciaAerea.getPaperis().add(this);
17    }
18
19    //Métodos getters e setters para todos os atributos básicos da classe
20
21    public CiaAerea getCiaAerea() {
22        return ciaAerea;
23    }
24
25    public List<Funcionario> getFuncionarios() {
26        return funcionarios;
27    }
28
29    public void add(Funcionario funcionario){
30        this.funcionarios.add(funcionario);
31    }
32 }
```

Listagem 10. Código da classe Funcionario.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 import java.util.List;
04 import java.util.ArrayList;
05
06 public class Funcionario extends Pessoa {
07     private String codigo, contaCorrente;
08
09     private Papel papel;
10
11     public Funcionario(Endereco endereco, Papel papel){
12         super(endereco);
13         this.papel = papel;
14         this.papel.add(this);
15     }
16
17     //Métodos getters e setters para todos os atributos básicos da classe
18     //Lembrando que já existem métodos getters e setters na classe Pessoa
19
20     public Papel getPapel() {
21         return this.papel;
22     }
23
24 }
```

Outro detalhe importante está relacionado ao limite superior das associações de composição. Repare que este limite superior vai até o valor de um atributo, por exemplo, **qtdPrimeira**. Como é uma associação com composição, esses valores devem ser passados para o construtor da classe **Horario**. Esses atributos servem para estabelecer a quantidade máxima de bilhetes por classe de voo que devem ser gerados.

Como a classe **Horario** possui uma associação de composição com a classe **Bilhete**, é necessário que durante a criação de uma instância de **Horario** sejam criadas as novas instâncias dos bilhetes.

Ainda falando sobre os atributos que representam as associações com bilhetes, repare que o nosso analista de negócio colocou um comentário ao lado da classe **Bilhete**: “Um Bilhete possui 3 horários?”, para alertar o desenvolvedor para tomar cuidado com essa associação.

Pensando na criação dos atributos da classe **Horario**, para representar essas associações devem ser criadas três listas de bilhetes? Ou devemos criar uma lista para cada instância que é filha da classe **Bilhete** (**Economica**, **Primeira** e **Executiva**)? Repare que na representação da associação nós temos no limite superior um elemento indicando as quantidades para cada instância das filhas da classe **Bilhete**. Logo, está implícito que cada associação representa exclusivamente uma ligação com uma classe filha, tornando a representação dessa forma a correta. A **Listagem 11** apresenta o código completo da classe **Horario**.

Codificação da hierarquia de Bilhete

Codificada a classe **Horario**, esta deve ter ficado com alguns erros de compilação referentes à hierarquia da classe de **Bilhete**, mas que agora serão corrigidos. A primeira classe a ser codificada dessa hierarquia deve ser a classe **Bilhete** e esta classe possui algumas particularidades. A primeira delas é que ela é uma classe abstrata, ou seja, não pode ser instanciada, da mesma forma que a classe **Pessoa**, mas isso não impede que sejam representadas as associações obrigatórias em seu construtor.

A classe **Bilhete** possui três associações obrigatórias com a classe **Horario**. Qual será o estado da venda de um bilhete que acabou de ser criado junto com a classe **Horario**? Provavelmente seja o estado **DISPONIVEL**, pois ao criar um horário sabemos que ele ainda não foi para a comercialização. Sendo assim, os bilhetes devem estar disponíveis para compra. Então, essa atribuição pode ser realizada dentro do construtor, pois o valor inicial sempre será esse, não havendo a necessidade de passar parâmetros.

A associação com o **enum TipoBilheteEnum** também é outro caso particular. Observe que no diagrama de classes existem três classes filhas e três valores no **enum** que representam os nomes de cada uma dessas classes. Assim, não é necessário informar o tipo do bilhete via parâmetro, pois é mais simples se dentro do construtor de cada classe filha atribuirmos o valor ao **enum** correspondente à classe. Portanto, no construtor da classe **Economica**, o atributo que representa o tipo de bilhete pode receber o valor **ECONOMICA**. Essa atribuição deve ser realizada da mesma forma nas outras classes filhas. Note que, como estamos em uma

hierarquia, não é interessante representar o atributo **tipoBilhete-Enum** em cada uma das filhas, podendo este ser criado dentro da classe **Bilhete** com o modificador de acesso **protected**.

A última associação obrigatória, entre a classe **Bilhete** e a classe **Horario**, deve ser passada como parâmetro ao construtor da classe **Bilhete**. No entanto, quantos horários devemos passar? Veja que no diagrama de classes existe a seguinte nota: “Um Bilhete possui 3 horários?”. Essa nota foi inserida pelo Analista de Negócio para que fique fácil a identificação de que existem três associações entre as classes **Bilhete** e **Horario**, mas não é necessário possuir três atributos que representem a classe **Horario** dentro da classe **Bilhete**.

Listagem 11. Código da classe Horario.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 import java.util.Date;
04 import java.util.List;
05 import java.util.ArrayList;
06
07 public class Horario {
08     private Long id;
09     private String codigo;
10     private Date partida, chegada;
11     private Integer qtdEconomica, qtdExecutiva, qtdPrimeira;
12
13     private Rota rota;
14     private Aviao aviao;
15     private List<Economica> economicas;
16     private List<Executiva> executivas;
17     private List<Primeira> primeiras;
18
19     public Horario(Rota rota, Aviao aviao, Integer qtdEconomica,
20                     Integer qtdExecutiva, Integer qtdPrimeira){
21         this.rota = rota;
22         this.rota.add(this);
23         this.aviao = aviao;
24         this.qtdEconomica = qtdEconomica;
25         this.qtdExecutiva = qtdExecutiva;
26         this.qtdPrimeira = qtdPrimeira;
27
28         this.economicas = new ArrayList<Economica>();
29         this.executivas = new ArrayList<Executiva>();
30         this.primeiras = new ArrayList<Primeira>();
31
32         for(int i = 0; i < qtdEconomica; i++){
33             this.economicas.add(new Economica(this));
34         }
35
36         for(int i = 0; i < qtdExecutiva; i++){
37             this.executivas.add(new Executiva(this));
38         }
39
40         for(int i = 0; i < qtdPrimeira; i++){
41             this.primeiras.add(new Primeira(this));
42         }
43     }
44
45     //Métodos getters e setters para todos os atributos básicos da classe
46     //Métodos getters para as associações
47     //Métodos add() para as associações que possuem atributos
        //representados por listas
48
49 }
```

Aprenda a interpretar Diagramas de Classes da UML – Parte 2

Listagem 12. Código da classe Bilhete.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 import java.util.List;
04 import java.util.ArrayList;
05
06 public abstract class Bilhete {
07     private Long id;
08     private Integer numero;
09     private String assento;
10
11     private SituacaoBilheteEnum situacao;
12     protected TipoBilheteEnum tipo;
13     private Horario horario;
14     private Passageiro passageiro;
15     private List<Bagagem> bagagens;
16
17     public Bilhete(Horario horario){
18         this.horario = horario;
19         this.situacao = SituacaoBilheteEnum.DISPONIVEL;
20         this.bagagens = new ArrayList<Bagagem>();
21     }
22
23     //Métodos getters e setters para todos os atributos básicos da classe
24
25     public void reservar(Passageiro passageiro){
26         this.passageiro = passageiro;
27         this.passageiro.getBilhetes().add(this);
28
29     }
30
31     public void comprar(){
32         this.situacao = SituacaoBilheteEnum.VENDIDO;
33     }
34
35     public void cancelarReserva(){
36         this.situacao = SituacaoBilheteEnum.DISPONIVEL;
37         this.passageiro.getBilhetes().remove(this);
38         this.passageiro = null;
39     }
40
41     protected abstract Integer maximoBagagens();
42
43     public void checkIn(Bagagem bagagem){
44         if(bagagens.size() < maximoBagagens()){
45             bagagens.add(bagagem);
46         } else {
47             throw new RuntimeException("Número Máximo de Bagagens alcançado");
48         }
49     }
50
51     //Métodos getters para as associações
52
53 }
```

Para tanto, apenas um atributo com o tipo **Horario** já é suficiente. Em outras palavras, pense em um exemplo real: um bilhete de voo possui três horários? Normalmente seu bilhete possui apenas um horário, mesmo que existam escalas, pois cada escala terá o seu próprio horário.

Outro ponto a ser observado é a associação entre as classes filhas de bilhete e a classe **Bagagem**. Repare que todas possuem ao menos uma bagagem. Neste caso você tem duas opções: criar um atributo **Bagagem** para cada filha e na quantidade necessária, ou criar uma lista de **Bagagem** na classe pai, a qual deixará para as filhas, através do método **checkIn(Bagagem)**, o controle da quantidade de bagagens aceitas por cada filha. Para realizar esse controle pode ser criado um método **protected abstract**, chamado **maximoBagagens()**, a ser implementado em cada filha, retornando a quantidade máxima de bagagens que um bilhete deve aceitar de acordo com o seu tipo.

Além do método **checkIn(Bagagem)**, existem mais três métodos na classe **Bilhete**, chamados de métodos de ação, pois modificam o estado de objetos dessa classe, como o valor de um ou mais atributos.

O primeiro deles, **reservar(Passageiro)**, indica que a situação do bilhete deverá mudar para o estado **RESERVADO** e deverá ser atribuído o passageiro ao atributo interno que reservou a instância do bilhete. Já o método **comprar()** deve concretizar a compra pelo **Passageiro** informado pelo método **reservar()** e mudar a situação do bilhete para **VENDIDO**. Por sua vez, o método **cancelarReserva()** deve atribuir um valor nulo ao passageiro, “removendo” este da reserva e voltando a situação do bilhete para o estado **DISPONIVEL**. O código da classe **Bilhete**

se encontra na **Listagem 12** e o código da classe **Economica** se encontra na **Listagem 13**. As classes **Primeira** e **Executiva** não serão apresentadas, pois são muito parecidas com a classe **Economica**. Dessa forma, utilize-a como base para a codificação delas.

Listagem 13. Código da classe Economica.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 import java.util.List;
04 import java.util.ArrayList;
05
06 public class Economica extends Bilhete {
07     public Economica(Horario horario){
08         super(horario);
09         this.tipo = TipoBilheteEnum.ECONOMICA;
10     }
11
12     protected Integer maximoBagagens(){
13         return 1;
14     }
15
16 }
```

Codificação da classe **Bagagem**

A última classe a ser criada na nossa camada Model é a classe **Bagagem**. Essa classe, diferente da classe **Bilhete**, possui uma associação com um enum, **TipoBagagemEnum**, a qual deverá ser informada como parâmetro do construtor da classe, pois não existem classes filhas especializadas para a **Bagagem**. Além dessa associação, existem outras três com as classes filhas de **Bilhete**.

Note que nesta classe existe mais uma anotação do analista de negócios: “Uma bagagem possui 3 bilhetes?”.

Se você, como desenvolvedor, seguir à risca o que está desenhado no diagrama UML, irá criar três atributos do tipo **Bilhete** na classe **Bagagem**. Agora, vamos pensar em um exemplo real. Suponha que você comprou um bilhete. Sua bagagem de viagem será dividida entre estes três bilhetes? Certamente não!

Portanto, a observação do analista e as associações estão ali para sinalizar que existem ligações com as classes filhas de **Bilhete**, mas cada associação possui um limite superior próprio de bagagens, identificando qual é o máximo permitido de bagagens para cada classe de bilhete. Então, a **Bagagem** pertence a apenas um **Bilhete**, o qual também deve ser passado como parâmetro ao construtor da classe **Bagagem**. A **Listagem 14** apresenta o código da classe **Bagagem**.

Listagem 14. Código da classe Bagagem.

```
01 package br.com.devmedia.bilhetesAereos.model;
02
03 public class Bagagem {
04     private Long id;
05     private Double peso;
06
07     private Bilhete bilhete;
08     private TipoBagagemEnum tipo;
09
10    public Bagagem(TipoBagagemEnum tipo, Bilhete bilhete){
11        this.tipo = tipo;
12        this.bilhete = bilhete;
13        this.bilhete.checkIn(this);
14    }
15
16    //Métodos getters e setters para todos os atributos básicos da classe
17    //Método getter para o atributo tipo
18    //Método getter para o atributo bilhete
19
20 }
```

O diagrama de classes UML do sistema de bilhetes aéreos apresentado não é único. Caso fosse modelado por outro analista de negócio, provavelmente existiriam variações/mudanças nas associações e quem sabe até em classes, sendo criadas novas classes e outras removidas. Sendo assim, mais importante que compreender o diagrama apresentado é ter o domínio dos seus recursos. Assim você será capaz de implementar códigos em conformidade com o que foi especificado.

Além desse tipo de diagrama, na UML existem outros que são muito importantes para o desenvolvimento de software e amplamente utilizados no mercado de desenvolvimento. Os mais básicos e considerados mais relevantes a todo desenvolvedor Java são: o diagrama de classes, o diagrama de casos de uso, o diagrama de sequência e o diagrama de estados. A maioria dos analistas de negócio faz uso destes diagramas para expressar as regras de negócio de um sistema.

Portanto, para um desenvolvedor Java crescer no mercado, não basta possuir conhecimentos sólidos em Orientação a Objetos e nas tecnologias da linguagem Java. Ter domínio sobre a UML é de suma importância para que o software seja produzido de acordo com as regras de negócio especificadas.

Autor



Everson Mauda

mauda@mauda.com.br - www.mauda.com.br

Bacharel em Ciência da Computação pela Universidade Federal do Paraná (UFPR), Mestre em Informática pela Pontifícia Universidade Católica do Paraná (PUC-PR). Analista e Desenvolvedor Java desde 2004, trabalhou em várias empresas multinacionais como GVT, HSBC, SERPRO e SITA. Baterista por hobby desde 2002 e entusiasta por ensinar aos outros desde que se conhece por gente.



Como adotar a análise estática de código

Como simplificar a detecção e a correção de bugs para obter um código-fonte de qualidade

Hoje, entende-se que quanto mais cedo um problema é encontrado no software, mais barata é sua correção (um problema encontrado em desenvolvimento será muito mais barato de corrigir do que se o mesmo for encontrado em produção). Da mesma forma, código-fonte bem escrito reduz drasticamente o tempo (e o custo) de manutenção, e o uso de boas práticas de orientação a objetos permite que o código seja flexível e suporte mudanças de forma mais tranquila, segura e natural.

Todos estes conceitos relacionados à qualidade são importantes para a vida das aplicações, mas a realidade de prazos cada vez mais curtos acaba fazendo com que, muitas vezes, a preocupação com a qualidade seja deixada de lado, de forma que detalhes importantes são ignorados, levando a aplicações mal construídas. O resultado é sempre o mesmo: alto custo de evoluções, demora em manutenções, alta taxa de bugs e insatisfação dos clientes.

Devido a isso, a qualidade ganha cada vez mais relevância, já existindo atualmente diversas práticas que podem ser utilizadas de forma a agregá-la ao desenvolvimento de software. E quando tais práticas são usadas corretamente não resultam necessariamente em impactos em prazo. Um bom exemplo é o par refatoração/testes automatizados, e paralela a este, existe a análise estática de código – tema deste artigo. A análise estática incorporada à IDE, assim como a prática de testes unitários, tornaram-se práticas tão naturais (pela sua praticidade e simplicidade) que passam a não ser mais dissociadas da própria escrita de código-fonte. Contudo, isto vale quando os conceitos estão bem entendidos, pois de outra forma, tais práticas podem surtir o efeito contrário, onerando a tarefa de desenvolvimento.

Além de explicitar o conceito de análise estática de código e apresentar ferramentas (de uso integrado às IDEs, para atuar de forma localizada nos arquivos, e de

Fique por dentro

Este artigo é útil para todos os desenvolvedores que primam pela melhoria de qualidade do código-fonte. Para isso, demonstrará a evolução de um código-fonte de má qualidade para um de melhor qualidade por meio de um método prático, que fará uso de conceitos e ferramentas para atingir este objetivo. Neste contexto, serão considerados problemas relacionados a estilo, más-práticas e bugs. Também será demonstrado o uso de ferramentas em dois níveis: na verificação da qualidade (por meio de plugins na IDE IntelliJ IDEA – com menção ao Eclipse) e na validação da qualidade (por meio da exibição de métricas de qualidade obtidas pela ferramenta SonarQube).

uso externo, mais geral, para cobrir todo o código-fonte do projeto de uma vez), este artigo irá demonstrar um método para aplicação da análise estática por meio de um estudo de caso hipotético, onde o código é verificado durante a construção e posteriormente validado por uma ferramenta específica (SonarQube), através de regras.

Análise estática de código

A análise estática de código é uma das práticas que verifica a qualidade do código-fonte. Esta verificação é realizada antes mesmo que haja execução do software (um conceito oposto ao dos testes unitários, que validam o software com base no resultado de sua execução – vide **BOX 1**).

BOX 1. Verificação x Validação (ou caixa branca x caixa preta)

Uma verificação considera “fazer certo alguma coisa”, ou seja, garantir que os passos para atingir um dado objetivo foram realizados corretamente; já a validação trata de “fazer a coisa certa”, ou seja, garantir que o objetivo seja atingido, independente da forma como os passos foram realizados. Verificação é eficiência; validação é eficácia. Também se entende que os testes de caixa branca (que consideram o código escrito) são aspectos de verificação, e os testes de caixa preta (que testam o output do software com base num dado input) são aspectos de validação. Considera-se, portanto, que a análise estática de código está relacionada à verificação, pois analisa como o código-fonte foi construído internamente.

A verificação é realizada com base num conjunto de regras pré-estabelecidas. O objetivo da análise é evidenciar problemas, para que possam ser corrigidos com o máximo de foco, resultando em maior eficiência no processo de melhoria de qualidade.

Além disso, assumir que a análise estática de código realiza verificação sem que o código-fonte seja executado não significa dizer que apenas arquivos texto (com extensão .java, .jsp, .js) são verificados. Cada ferramenta, dependendo de sua implementação, pode realizar a verificação em código-fonte ou bytecode. Embora algumas ferramentas sejam capazes de realizar a verificação estática em arquivos JSP e JavaScript, por exemplo, a análise aqui proposta foca-se especificamente em código-fonte Java (bem como EM bytecodes).

A análise estática pode ter sua verificação agrupada em três aspectos principais, a saber:

- **Verificação por estilo:** Considera elementos como identação, espaços e tabs, convenção de nomes, número de parâmetros, alinhamento na vertical, formato e presença de comentários, dentre outros. São todos os aspectos que contribuem para tornar o código mais padronizado, organizado e legível. A ferramenta mais utilizada para verificação por estilo é o Checkstyle;
- **Verificação por boas práticas:** Aplica uma gama de regras para verificar se práticas corretas estão sendo realizadas, como evitar duplicação de código, garantir o correto uso de encoding, implementação do método `clone()`, tamanho de métodos e classes, tamanho de parâmetros, uso do padrão Singleton, criação desnecessária de variáveis locais e muitas outras. O conjunto de regras é extenso e visa garantir que o código apresente as melhores práticas possíveis. A ferramenta de verificação mais utilizada para aplicar boas práticas é o PMD;
- **Verificação por bugs:** Trata de encontrar erros no sistema. Isto é importante para antecipar a identificação de problemas no software (até antes mesmo de sua execução pelo cliente). A ferramenta mais utilizada para identificação de bugs é o Firebug.

A verificação das ferramentas PMD e Checkstyle é realizada sobre código-fonte não compilado (arquivo com extensão .java), mas a verificação com Firebug analisa bytecodes (arquivos compilados do Java, com extensão .class). Desta forma, para análises envolvendo Firebug, uma compilação prévia é necessária. Isso é tratado pelo seu respectivo plugin, de forma que se faz transparente ao desenvolvedor.

0 que não é análise estática – testes e validação por contrato

Uma vez que é comum utilizar tanto a análise estática quanto a escrita de testes automatizados em termos de garantia e melhoria de qualidade em código-fonte, é importante frisar a diferença entre elas.

Testes automatizados são uma prática que trata da validação do código-fonte de produção através de sua execução. Isto funciona pela chamada do código de produção pelo código de teste, que faz o papel de cliente do código de produção. Os testes necessitam, portanto, executar o código para identificar problemas por meio

da quebra de asserções definidas. Isso difere essencialmente da análise estática, que valida código-fonte sem que o mesmo seja executado.

Apesar dessa diferença, é muito importante que as duas técnicas sejam usadas em parceria, pois suas características distintas não as tornam mutuamente excludentes, mas sim complementares. Saber a diferença conceitual entre tais técnicas ajuda a aplicá-las de forma correta, nos momentos adequados.

Ainda conceitualmente falando, é possível agrupar os tipos de testes automatizados em três categorias principais:

- **Testes Unitários:** Pode-se entender, inicialmente, que um teste unitário testa os métodos de objetos de forma independente. Contudo, são raros os objetos que não possuem dependências com outros objetos. Desta forma, uma melhor definição de testes unitários é a de que são aqueles que testam objetos que não possuem dependência externa em termos de componentes, ou seja, que não dependam de web services, banco de dados, sistema de arquivos, etc.;
- **Testes de Integração:** São aqueles testes que testam código cuja execução depende de componentes externos. Por exemplo, um teste que valida se um serviço traz a lista de clientes esperada é um teste que executa um serviço que, possivelmente, acessa um banco de dados ou um web service para trazer os clientes. A premissa é que o serviço sendo testado não funciona sem que haja a dependência com o banco ou o web service. Os testes de integração podem ocasionar algum atraso no desenvolvimento por demandarem mais tempo em sua execução e serem geralmente mais complexos de criar. O trade-off de seu uso deve ser considerado (diferentemente dos testes unitários, que podem ser mais utilizados, por serem rápidos de criar e executar). Geralmente são implementados com o uso de um framework de testes (como JUnit ou TestNG) atrelado a um ou mais frameworks que lidam com recursos externos de forma geral (como Spring, EJB, etc.);
- **Testes Funcionais:** Testes considerados funcionais são aqueles que testam a aplicação de um ponto de vista funcional, ou seja, a partir das telas. A automatização de testes neste nível considera a reprodução da navegação no sistema e verifica se as telas se comportam de acordo com o esperado (se um link direciona para uma tela específica, por exemplo). Quando feitos de forma a agregar em si a mesma validação realizada pelos testes de integração, tornam-se mais pesados que os testes de integração, pois repetem os mesmos testes ao mesmo tempo em que adicionam a testabilidade em nível funcional (ou seja, testam telas e integração). Desta forma, os testes funcionais são melhores empregados quando utilizados como complemento aos testes de integração (por exemplo, focando mais na validação de uma navegação do que nos resultados esperados – o que pode caber aos testes de integração). Pode-se realizar esta automatização por meio de ferramentas especializadas, como o Selenium.

Existem mais categorizações, mas pode-se considerar que as três mencionadas são as principais. Além disso, as nomenclaturas também podem variar, sem que haja alterações semânticas.

Também é importante considerar que há muita confusão no quesito dos tipos de testes, sendo comum tratar todos os tipos de testes automatizados como “testes unitários”. Isto é um problema, no sentido que testes unitários são leves e rápidos, enquanto testes de integração e funcionais são mais trabalhosos e podem levar mais tempo para sua efetiva construção e execução. O que ocorre é que, num cenário de desconhecimento dos tipos de testes, a simples menção ao uso de testes unitários leva os stakeholders do projeto a reagir de forma negativa, por imaginarem que a adição de tais testes atrasará o projeto (pois consideram que todos os testes automatizados são testes unitários, o que não é verdade pela categorização supracitada).

De fato, a adição de testes unitários, quando feita dentro do correto conceito ao qual se referem, tanto não atrasa o projeto quanto oferece ganhos na redução de bugs em potencial. Contudo, a adição, de forma simultânea, de testes unitários, de integração e funcionais pode sim ter impactos no prazo, dependendo da característica do projeto sendo testado. Tal impacto é diretamente proporcional ao nível de qualidade da arquitetura do projeto (em termos de design, boas práticas de orientação a objetos, etc.). Quanto melhor a arquitetura que suporta o código, menor o impacto na adição de qualquer tipo de teste automatizado.

Portanto, a forma como o desenvolvimento flui produtivamente num projeto (ou seja, sem desvios significativos em cronograma), considerando a adição de testes automatizados, é diretamente influenciada pelo entendimento dos tipos de testes. Os testes unitários devem ser aqueles que rodam rápidos, viabilizando práticas como o TDD (onde testes unitários andam lado-a-lado com o código-fonte de produção, oferecendo uma série de ganhos em qualidade e design), testes de integração podem ser mais lentos de criar e executar por depender de recursos externos e testes funcionais devem ser usados com critério, pois testam a interface do cliente, podem ser mais pesados para executar e tornar-se redundantes em relação aos testes de integração. Tal conhecimento das características de cada tipo de teste pode permitir que as tarefas relativas à automatização sejam melhor planejadas e estimadas, mitigando desvios no prazo do projeto.

Outra técnica utilizada que visa o aumento da melhoria de qualidade é o design por contrato. Este conceito, desenvolvido por Bertrand Meyer nos anos 80, sugere que classes e métodos possuam pré-condições, invariantes e pós-condições explicitamente declaradas, de forma que a implementação do que entra, do que sai e do que não pode ser alterado num dado objeto seja garantida. Isto apresenta ganhos significativos na melhoria do código-fonte como um todo (por exemplo, a pós-condição declarada de que um método não pode receber um parâmetro nulo impede que ocorram bugs por **NullPointerException** futuramente, quando a aplicação já estiver em produção e o custo de correção for bem mais caro). Design por contrato restringe o potencial de erros numa aplicação.

Da mesma forma que os testes automatizados, a técnica de design por contrato depende da execução do código para sua validação. Hoje já existem frameworks em Java, como o Contract4J, que fazem

uso de anotações e aspectos para realizar a validação. Além disto, a especificação Java “JSR-303: Bean Validation” define uma série de anotações (como `@NotNull`, `@Min`, `@Max`, etc.) que, atreladas à implementação do Hibernate Validator, por exemplo, podem ser vistas como uma implementação da validação de design por contrato.

Enfim, testes automatizados e design por contrato são técnicas de validação que devem ser complementares à análise estática de código, para melhoria da qualidade do software como um todo.

Quando realizar a análise estática de código

A análise estática de código pode ser feita em diferentes momentos no ciclo de vida de desenvolvimento. Geralmente, a análise não é feita de forma preditiva, ou seja, o código é analisado e corrigido apenas depois de já ter sido versionado (o que significa que já pode estar em homologação ou produção). Este processo é ilustrado na **Figura 1**.

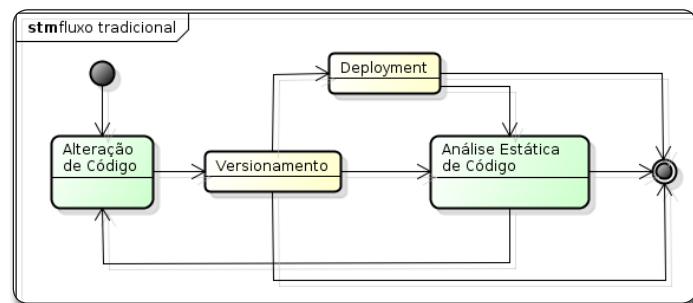


Figura 1. Análise estática de código a posteriori

Este fluxo apresenta algumas desvantagens, a saber:

- A análise ocorre somente depois do código já ter sido versionado, de forma que o mesmo precisa ser revisitado para ajustes e novo versionamento e/ou deployment;
- A pessoa que corrige uma análise pode não ser a mesma que implementou o código, o que pode aumentar a complexidade da correção, ocasionando atrasos;
- Não há garantias que uma análise estática de código acabe resultando em melhoria efetiva no código-fonte analisado, pois uma vez que a análise é executada posteriormente a um versionamento ou deployment, cabem às pessoas a ação de realizar a correção e novo versionamento/deployment – e este fluxo não é garantido.

Sabe-se que quanto mais cedo um problema é encontrado no código, mais barato é para corrigi-lo, em escala exponencial. Com base neste conceito, uma verificação antecipada da aderência do código-fonte (em relação a um conjunto de boas práticas e regras proposto pela empresa e/ou pela comunidade) no ciclo de vida do desenvolvimento traz benefícios ao custo da correção dos problemas (em forma de redução de tempo e recursos gastos na atividade de melhoria de código).

Desta forma, propõe-se uma inversão no fluxo, de forma que a análise estática de código possa acontecer juntamente com a alteração de código e antes do versionamento.

Um fluxo preditivo é ilustrado na **Figura 2**.

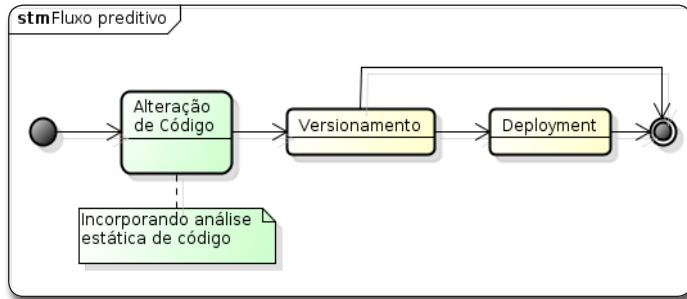


Figura 2. Fluxo com análise preditiva

Esta simplificação no fluxo sugere, na prática, que o desenvolvedor deve realizar a análise estática de código (e suas correções indicadas) antes de versionar a modificação, ou seja, juntamente com a modificação do código. O método é simples e consiste em: fazer o ajuste, verificar o ajuste com as ferramentas de análise estática e, se estiver tudo certo, então versionar; senão, repetir a verificação.

Versionamento: Um breve histórico

O conceito de versionamento declara que estados diferentes de artefatos devem ser controlados, permitindo que diferentes versões possam ser recuperadas. As ferramentas de Sistema de Controle de Versão (VCS – *Version Control System*) atendem a este fim, garantindo uma série de ações como, por exemplo, voltar o arquivo a um estado anterior, voltar todo o projeto a um estado anterior, comparar estados de um arquivo, verificar quem foi a pessoa que modificou um arquivo que está causando problema, dentre outros.

Existem três tipos de VCS: o Local, o Centralizado e o Distribuído. No VCS local, toda mudança em arquivo é gravada num banco de dados simples. Contudo, o problema reside na dificuldade de colaborar com outros desenvolvedores (além do risco de perder tudo se o HD for danificado). Um exemplo deste modelo é o RCS. Já no VCS centralizado, todas as mudanças realizadas são salvas num servidor centralizado, que possui clientes que baixam cópias dos arquivos, os modificam e os retornam ao servidor (resolvendo, portanto, o problema da colaboração). Contudo, talvez o maior problema do VCS centralizado seja justamente sua maior qualidade: ser centralizado. A centralização implica que, quando o servidor cair, ninguém conseguirá colaborar. Outros problemas são relativos àqueles encontrados na relação cliente/servidor (como latência de rede, proxy, permissões) e na questão de todo dado histórico ser perdido caso o servidor centralizado sofra um crash e não haja backup. Alguns exemplos de VCS centralizados são o CVS e o SVN.

Frente aos problemas do VCS centralizado, surgiu o VCS distribuído. O conceito básico é que todo o repositório esteja disponível em cada máquina de desenvolvimento. Assim sendo, para realizar qualquer ação (como, por exemplo, puxar um histórico de arquivos), não é preciso nem ao menos ter acesso à rede – o acesso à rede pode ser feito esporadicamente, para sincronização, se desejado. Talvez a maior desvantagem desse modelo seja, novamente, o risco de perder tudo no caso de um crash no HD.

Contudo, isto pode ser mitigado através de sincronismos contínuos (e, diferentemente do VCS centralizado, no distribuído cada desenvolvedor tem todo o repositório localmente, de forma que, no caso do crash do HD, basta copiar o repositório de outro desenvolvedor com dados mais recentes, e apenas aquilo que não foi previamente sincronizado é perdido). O exemplo mais famoso de um VCS distribuído é o Git, seguido pelas ferramentas Mercurial e Bazaar.

Agora que temos uma base conceitual, o entendimento da diferença entre validar código em execução e verificar código que não é executado e o conhecimento de ferramentas de verificação, vamos configurar a IDE para suportar as ferramentas.

Ferramentas de análise estática: Uma visão geral

A ferramenta PMD é um analisador de código que encontra falhas potenciais que sugerem pouco uso de boas práticas. As linguagens e frameworks atualmente suportados, além de Java, são PLSQL, JavaScript, XML, XSL e Velocity. Para a questão específica de código duplicado, consegue atuar num range maior de linguagens, como Ruby, Scala, Python, Go, dentre outras. O PMD possui plugins para diversas IDEs, como NetBeans, JDeveloper, JBuilder, Eclipse e IntelliJ, assim como possui um plugin para Maven. No momento da escrita deste artigo, a última release era a 5.3.1, de abril/2015, e a próxima versão será a 5.4.0, atualmente como Snapshot.

A ferramenta CheckStyle tem a função de verificar o código-fonte buscando problemas com estilo, em código que diverge de padrões de codificação diversos, como Sun Code Convention, Google Java Style, dentre outros. É uma ferramenta muito customizável e possui integração com o Maven, o que permite adicionar a verificação ao ciclo de build e gerar relatórios. Novas releases são liberadas com frequência (ao menos uma vez ao mês) e atualmente o projeto está hospedado no GitHub.

Por fim, o Findbugs é uma ferramenta de análise estática de código com a função de encontrar bugs. Foi criada por Bill Pugh (professor da Universidade de Maryland) e David Hovemeyer, nos Estados Unidos. No momento da escrita deste artigo, a última release era a 3.0.1, de março/2015. Por fim, vale frisar que o Findbugs é uma ferramenta bastante utilizada e que possui plugins para o Eclipse, IntelliJ, NetBeans, Gradle, Maven e Jenkins.

Realizando a análise e os ajustes com IntelliJ

Como verificado, estas ferramentas de verificação possuem plugins para o IntelliJ. Deste modo, visando também ampliar o leque de ferramentas de conhecimento do leitor, este artigo fará uso desta IDE, em sua versão 14.1.

Sabe-se que o IntelliJ é um ambiente de desenvolvimento pago e isto tem levado muitos desenvolvedores a permanecerem utilizando o Eclipse. Contudo, nem todos sabem que a versão Community Edition do IntelliJ, que é gratuita, em sua versão 14.1, possui uma quantidade tão grande de features da versão Ultimate (paga) que atende completamente ao desenvolvimento profissional de software. Aliado a isto, sua estabilidade, visual e um grande conjunto de atalhos tem levado muitos desenvolvedores a migrarem do Eclipse para o IntelliJ. Portanto, vale a pena experimentar.

Como adotar a análise estática de código

Após instalar o IntelliJ, é preciso instalar os plugins. Para fazer isso, acesse o menu *File > Settings > Plugins* e clique no botão *Browser Repositories*, o que gerará como resposta uma lista de plugins disponíveis (conceito similar ao do Eclipse MarketPlace). Nesta tela, realize um filtro buscando por “QAPlug”, sem aspas (o plugin de análise estática de código do IntelliJ). Como resultado, este plugin será exibido, bem como suas extensões. Desta lista, exibida na **Figura 3**, instale o QAPlug e suas extensões para Checkstyle, FindBugs e PMD.

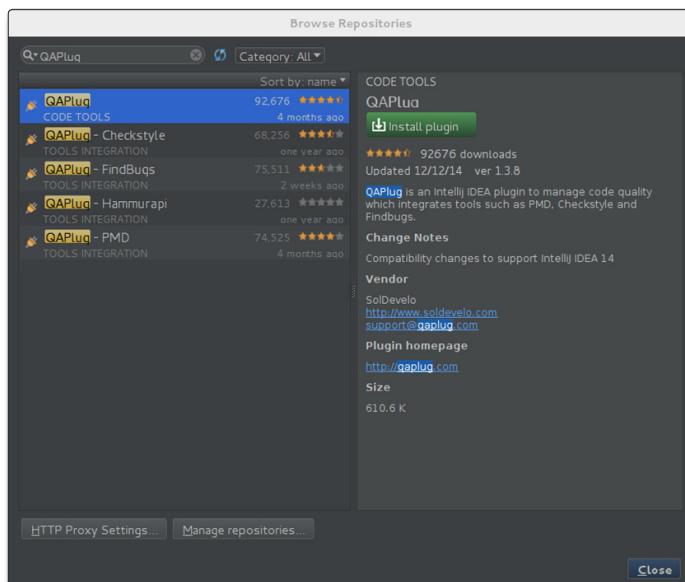


Figura 3. Instalação dos plugins no IntelliJ IDEA

Após reiniciar o IntelliJ, estamos prontos para uma demonstração da aplicação de análise estática de código sobre um código-fonte simples, mas que apresenta problemas relacionados à ausência de boas práticas, falta de aderência a regras de estilo e existência de bugs implícitos. Para isso, crie um novo projeto Java (menu *File > New Project*), e neste projeto, crie uma classe chamada **SomeCalculator** (clique com o botão direito sobre o diretório de código-fonte do projeto e selecione a opção *New > Java Class* do menu de contexto). A **Listagem 1** demonstra o código a ser inserido na classe, que pretende somar o valor 22 ao número 10 e retornar o resultado.

Perceba que o código-fonte, embora simples, possui uma série de problemas, como o uso de **System.out**, mal-uso de **BigDecimal** para adição de valor e números constantes que nada significam (não é possível entender por que estão lá). O que desejamos, em termos de verificação de código-fonte utilizando análise estática, é que o plugin QAPlug e suas extensões (instaladas de acordo com a **Figura 3**) sejam capazes de indicar quais são estes problemas.

Para realizar a análise, selecione a classe Java da **Listagem 1** e utilize a tecla de atalho **ALT+SHIFT+A** (ou o menu *Analyze > Analyze Code...*) para abrir a caixa de diálogo *Specify Analysis Scope* (**Figura 4**). Esta caixa permite selecionar opções de escopo (no caso, deixaremos no default para que o escopo de análise seja restrito

à classe selecionada) e de profile (que é o perfil de regras a ser aplicado; deixaremos no default para que seja utilizado o conjunto default de regras de verificação que vem com o plugin). Dito isso, clique em **OK** para fazer a análise do arquivo.

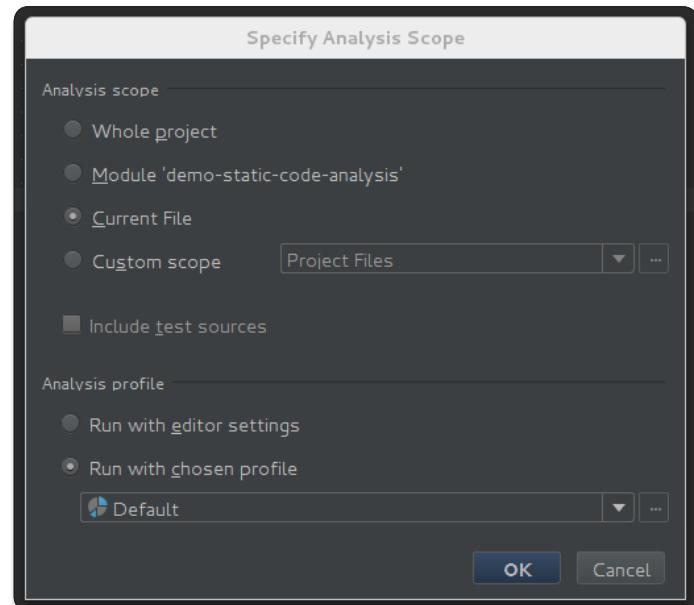
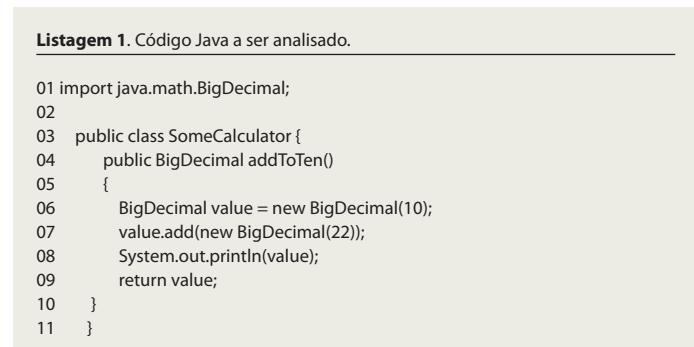


Figura 4. Executando a análise estática de código

Após clicar em **OK**, o plugin QAPlug irá verificar a classe utilizando suas regras de acordo com o PMD, Findbugs e Checkstyle (que são suas extensões). O conjunto de erros reportados, portanto, é relativo à soma das validações de cada uma destas extensões.

Neste caso, a execução da análise estática aponta para um total de sete erros, exibidos na **Figura 5**. Destes, um é bug (identificado pelo Findbugs), três são más-práticas (identificados pelo PMD) e três são relativos a estilo (identificados pelo Checkstyle). Uma forma simples de identificar qual extensão é responsável por qual regra é olhar o ícone de cada item na **Figura 5**. O Findbugs exibe um ícone de um bug, o PMD exibe o ícone de sua sigla e o Checkstyle exibe um ícone indefinido.

Neste caso simples, os erros poderiam ter sido verificados numa checagem manual. Contudo, a verificação manual sempre é sujeita a falhas, e quando códigos mais complexos passam a

ser considerados, os plugins se tornam indispensáveis para uma identificação rápida e prática dos problemas. Cada um dos sete problemas identificados é melhor descrito na **Tabela 1**.

Observe, pelos resultados, que:

- Plugins diferentes às vezes identificam os mesmos problemas (na linha 7, por FindBugs e PMD);
- Algumas das regras identificadas podem não ser consideradas úteis pelo desenvolvedor ou pela política da organização (o problema da linha 4, por exemplo, pode não fazer muito sentido);
- Algumas regras que podem ser consideradas importantes não foram aplicadas (por exemplo, o Checkstyle não considerou que as chaves de abertura do método devem estar na mesma linha da assinatura do método, na linha 4).

Os resultados demonstram que a análise estática indica problemas de acordo com regras. Cabe às pessoas a ação sobre os problemas identificados (incluindo a adição ou remoção de regras para refinar ou deixar de restringir aspectos do código-fonte). Sobre configuração de regras, ainda neste artigo falaremos do SonarQube e de como essas regras podem ser gerenciadas e aplicadas em nível corporativo.

Após as correções realizadas com base no resultado da análise estática, o código-fonte fica como o indicado na **Listagem 2**. Com exceção do **System.out**, que foi removido por considerar que o código não precisava mais dele e nem de log, as demais linhas foram ajustadas obedecendo aos problemas reportados.

Embora ainda não se trate do melhor código-fonte possível, os plugins permitiram um refinamento que exigiu pouco esforço de desenvolvimento, ao mesmo tempo em que resolveu um bug, melhorou a legibilidade do código e removeu código desnecessário.

Neste momento é importante ressaltar que em todo código-fonte modificado existe o risco de adição de novos bugs.

Desta forma, é sugerido como boa prática de engenharia de software que toda refatoração venha acompanhada do suporte de testes automatizados, de forma que o teste possa validar, por meio de regressão se, após a refatoração, o código continua funcionando como deveria.

Portanto, os testes devem ser utilizados como complemento à atividade da análise estática de código. Enquanto a segunda identifica os problemas, a primeira valida se a refatoração, visando a resolução destes problemas, não impactou no funcionamento da aplicação como um todo.

Listagem 2. Código Java ajustado após a análise problemas.

```
01 import java.math.BigDecimal;
02
03 public class SomeCalculator {
04     public BigDecimal addToTen()
05     {
06         BigDecimal value = BigDecimal.TEN;
07         value = value.add(new BigDecimal(valueToAdd));
08         return value;
09     }
10 }
```

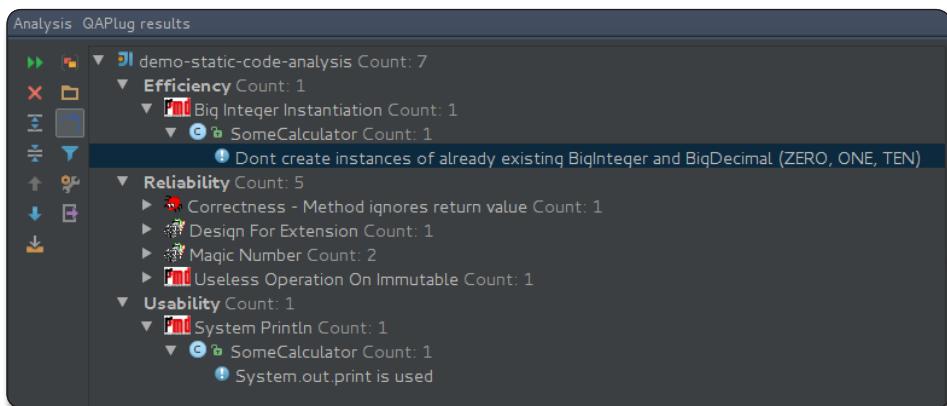


Figura 5. Resultado da verificação de um código simples

Plugin	Linha	Problema Identificado
Checkstyle	4	Como o método não é estendido, Checkstyle entende que seu design não considera extensão e que, portanto, o método precisa ser declarado como final para estar coerente com seu design (o que pode ou não ser verdade na prática).
PMD	6	Utilize uma constante para o valor 10 .
FindBugs	7	O método value.add() não altera o estado do objeto value na adição (como pode-se imaginar), e sim apenas retorna o resultado. O código está esperando um comportamento diferente do que é realizado.
Checkstyle	6, 7	Não use números mágicos. Números mágicos são constantes “soltas”, ou seja, que não declaram significado. O que 10 representa? Melhor seria utilizar uma constante, onde seu nome declararia seu significado.
PMD	7	O método value.add() não altera o estado do objeto na adição (como pode-se imaginar), e sim retorna a adição sem alterar o estado de value (conceito aplicado em objetos imutáveis como BigDecimal). O código está esperando um comportamento diferente do que é realizado.
PMD	8	Não use System.out . É uma má prática de desenvolvimento adicionar saídas para o console, pois este código pode ser útil em desenvolvimento, mas não será quando em produção. Se uma informação é relevante o suficiente para que precise ser escrita, utilize log para isto. Caso contrário, não mantenha no código.

Tabela 1. Problemas identificados pelos plugins de análise estática

Nota

Caso queira utilizar o projeto de exemplo deste artigo, o mesmo pode ser baixado do GitHub (use a opção Download Zip, ou, caso esteja adotando o Git, faça o clone do repositório com o comando: git clone https://github.com/meideiros/demo-static-code-analysis.git). O código também pode ser obtido na página de download desta edição da revista.

Tornando a análise estática uma ação corporativa com SonarQube

Até aqui, analisamos a aplicação de análise estática de código utilizando regras default de plugins das ferramentas PMD, Findbugs e Checkstyle. Contudo, é comum que empresas possuam a necessidade de customizar suas próprias regras de verificação (seja ativando, desativando, adicionando ou excluindo regras). Desta forma, veremos aqui como podemos incorporar regras específicas à nossa IDE, de forma a verificar o código-fonte de acordo com as exigências da corporação.

Validação e gerenciamento de regras: uso de ferramenta específica

Um conceito-chave por trás da análise corporativa de código é que as regras estejam centralizadas, de forma a serem utilizadas por todos os desenvolvedores. A implementação deste conceito pode ser realizada por meio de uma ferramenta específica, num servidor também específico (por exemplo, o SonarQube – ferramenta para centralização destas regras – pode ser instalado num servidor específico da rede, de onde os desenvolvedores tenham livre acesso). Além disto, é importante que a ferramenta permita exportar suas regras para que sejam importadas nas IDEs (afinal, espera-se que tais regras centralizadas possam ser utilizadas pelos desenvolvedores de forma prática). A IDE consegue, então, aplicar as mesmas regras que a ferramenta, garantindo certo nível de consistência na verificação do código-fonte.

Conforme dito anteriormente, a análise preditiva sugere que a verificação seja feita o quanto antes, e agora, ela pode ser feita usando as regras da empresa. Um exemplo de configuração (mas não o único) pode ser observado na **Figura 6**.

Nesta imagem, observa-se que os plugins da IDE (no caso, PMD, FindBugs e Checkstyle) e a ferramenta SonarQube (de análise estática de código, hospedada num servidor de livre acesso) possuem uma relação comum e direta com as regras da corporação: enquanto o SonarQube mantém e exporta as regras, os plugins da IDE dos desenvolvedores as importam. Desta forma, garante-se que as regras sendo utilizadas na IDE são as mesmas utilizadas pela ferramenta que as mantém.

Como em todo método, vale ressaltar que existem pontos positivos e negativos. O ponto negativo da abordagem de exportação/importação de regras é que se trata de um esforço manual. O problema é que, no momento em que houver uma alteração em regras na ferramenta centralizada de gerenciamento de análise estática de código, haverá uma diferença entre o declarado na ferramenta centralizada e o utilizado nas IDEs, e enquanto não houver um novo esforço de exportação/importação de regras, as regras entre esses dois pontos não estarão sincronizadas. Neste cenário, cabe ao time de desenvolvimento organizar-se de forma a manter as IDEs

atualizadas de acordo com as alterações na ferramenta de análise estática de código – e ao time responsável pelo gerenciamento das regras a tarefa de divulgar as atualizações.

Para resolver os problemas de sincronia entre as regras declaradas na IDE e as declaradas na ferramenta centralizada do servidor, uma abordagem válida (que está além do escopo deste artigo, mas que vale a pena ser mencionada) é de que a IDE faça a integração diretamente com a ferramenta do servidor, através de plugin. Atualmente, tanto o Eclipse quanto o IntelliJ possuem plugins para integração com o SonarQube (ferramenta muito utilizada para centralização de regras e verificação/validation de código-fonte). Contudo, deve-se considerar o trade-off: se por um lado a integração direta com o SonarQube apresenta o ganho óbvio de sincronia entre regras, pode também apresentar problemas comuns de dependência de aplicações externas acessíveis por rede, tais como latência, configuração de proxy, necessidade de gerenciamento e concessão de permissões de acesso.

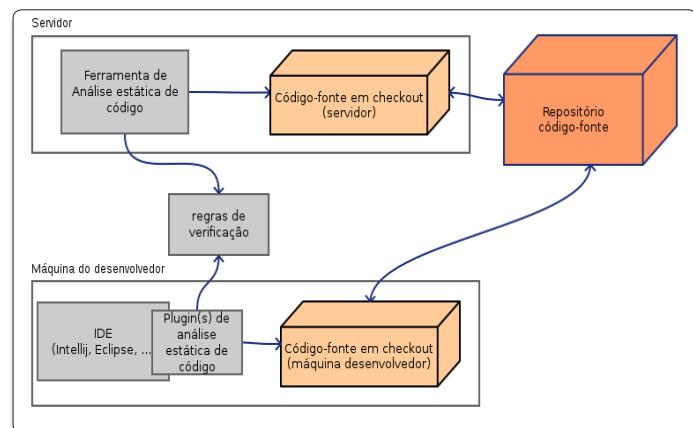


Figura 6. Exemplo de configuração do uso de regras corporativas.

Introduzindo o SonarQube

A ferramenta SonarQube (antigamente chamada apenas de "Sonar") é uma plataforma de apoio à qualidade do código-fonte das aplicações, possuindo muitas funções. Nela, sete eixos de qualidade são cobertos: Arquitetura e Design, Duplicações, Testes Unitários, Complexidade, Potenciais Bugs, Regras de Codificação e Comentários. Como um dos seus diferenciais, a ferramenta é extensível pelo uso de plugins (possui mais de 50 disponíveis) e cobre mais de 20 linguagens, incluindo Java, C, JavaScript, Groovy, dentre outras. Além disso, o SonarQube é uma Trata-se de uma aplicação web capaz de trabalhar com ferramentas de Integração Contínua, como Jenkins, Atlassian Bamboo, CruiseControl, dentre outras.

Embora também possa ser utilizada para verificação de métricas de testes automatizados e cobertura de código, no escopo deste artigo as funções relevantes do SonarQube que serão apresentadas são relacionadas ao gerenciamento de regras de análise estática de código para a corporação e à exportação de tais regras como arquivo XML, para serem utilizadas pelas IDEs de desenvolvimento (aspectos já analisados a partir da **Figura 6**).

Para que seja possível conhecer boa parte do seu funcionamento sem que seja preciso instalar a aplicação, o SonarQube mantém um site demo (o projeto Nemo). Este projeto nada mais é do que uma demonstração online das funcionalidades do SonarQube. Na seção **Links** pode-se encontrar a URL do projeto.

Aplicando a análise estática uso corporativo

Apoiando-se nos conceitos mencionados, será apresentado a seguir um método para aplicação do conceito de análise estática de código em âmbito corporativo (numa realidade onde a empresa faça uso da ferramenta SonarQube para gerenciamento de regras de qualidade). Tal método baseia-se num procedimento simples: a exportação das regras do SonarQube para que sejam importadas na IDE do desenvolvedor. Desta forma, este terá uma IDE configurada para realizar a análise estática de acordo com as regras da corporação. Este método faz uso do SonarQube como ferramenta de verificação (no sentido em que mantém as regras de verificação) e ferramenta de validação (no sentido em que interpreta o resultado de execução das regras, expondo-o como métricas de qualidade).

Exportando regras do SonarQube para o IntelliJ

Os passos a seguir pressupõem que haja acesso disponível à ferramenta SonarQube em sua empresa. Caso não seja o caso, os passos podem ser simulados com uma instalação local desta ferramenta. O download do SonarQube pode ser feito de forma gratuita e a instalação é realizada em poucos e simples passos. Feito isso, de dentro da própria ferramenta é possível instalar os plugins de análise estática (ela já vem com o FindBugs e permite a instalação do PMD e Checkstyle através de botões intuitivos em sua interface). Após a instalação e start do processo, a aplicação passará a responder na porta 9000.

Logo após, a IDE precisa ser configurada para ter as mesmas regras de análise estática existentes no SonarQube. Para isto, o primeiro passo é a exportação destas regras do SonarQube para um arquivo XML (para que possam ser importadas pela IDE, em

NAME	RULES	PROJECTS	DEFAULT
FindBugs	379	0	<input type="checkbox"/>
general	432	0	<input checked="" type="checkbox"/>
Sonar way	204	0	<input type="checkbox"/>

Figura 7. Tela de profile de regras

Link	URL
All rules	http://localhost:9000/profiles/export?language=java&name=general
Checkstyle	http://localhost:9000/profiles/export?format=checkstyle&language=java&name=general
FindBugs	http://localhost:9000/profiles/export?format=findbugs&language=java&name=general
PMD	http://localhost:9000/profiles/export?format=pmd&language=java&name=general

Figura 8. Regras dos plugins de análise estática no SonarQube

seguida). Para exportar as regras existentes, acesse o menu *Quality Profiles* e encontre o profile (um perfil de configuração, que permite agrupar regras de acordo com aspectos comuns – por exemplo, um profile somente para o Findbugs, outro para regras da Empresa, outro para regras da comunidade, etc.) que está definido como default (o profile default é aquele que efetivamente é utilizado pelo SonarQube nas validações). Na **Figura 7**, o profile definido como default é o *general*. Portanto, este é o profile cujas regras de análise estática serão efetivamente aplicadas – e que queremos, portanto, levar para a IDE. Saiba que todos os profiles, independentemente de estarem com o status *default* ou não, podem ter suas regras exportadas. Contudo, exportaremos o profile com status *default* porque as regras que estão sendo adotadas no momento são as dele.

Uma vez escolhido o profile, clique em seu nome (segundo a **Figura 7**, deve-se clicar em *general*) e, logo após, em *Permalinks*. Feito isso, os links para as regras dos plugins devem ser exibidos, como

demonstra a **Figura 8**. Cada um destes links referencia um XML no servidor, contendo as respectivas regras de análise estática. Então, para salvar estes arquivos XML, clique com o botão direito do mouse sobre cada link (de *Checkstyle*, *Findbugs* e *PMD*) e selecione a opção *Save Link As....*. Salve cada um destes arquivos para que possam ser usados depois.

Importando as regras no IntelliJ

O próximo passo será importar as regras dos arquivos XML para o IntelliJ. Para isso, abra a IDE e tecle **CTRL+ALT+S** (ou utilize o menu *File > Settings*) para acessar a caixa de diálogo de propriedades. Em seguida, na seção *Other Settings > QAPlug > Coding Rules*, clique no ícone “+” (*Add*) para criar um novo profile e selecione *IDE profile* (para que as regras sejam atreladas à IDE, de forma que estejam disponíveis a todos os projetos). Então, dê um nome relevante ao profile, marque o checkbox *Import profile* e selecione um dos arquivos XML de regras previamente exportado no Nemo. Depois, clique em *Ok*.

Como adotar a análise estática de código

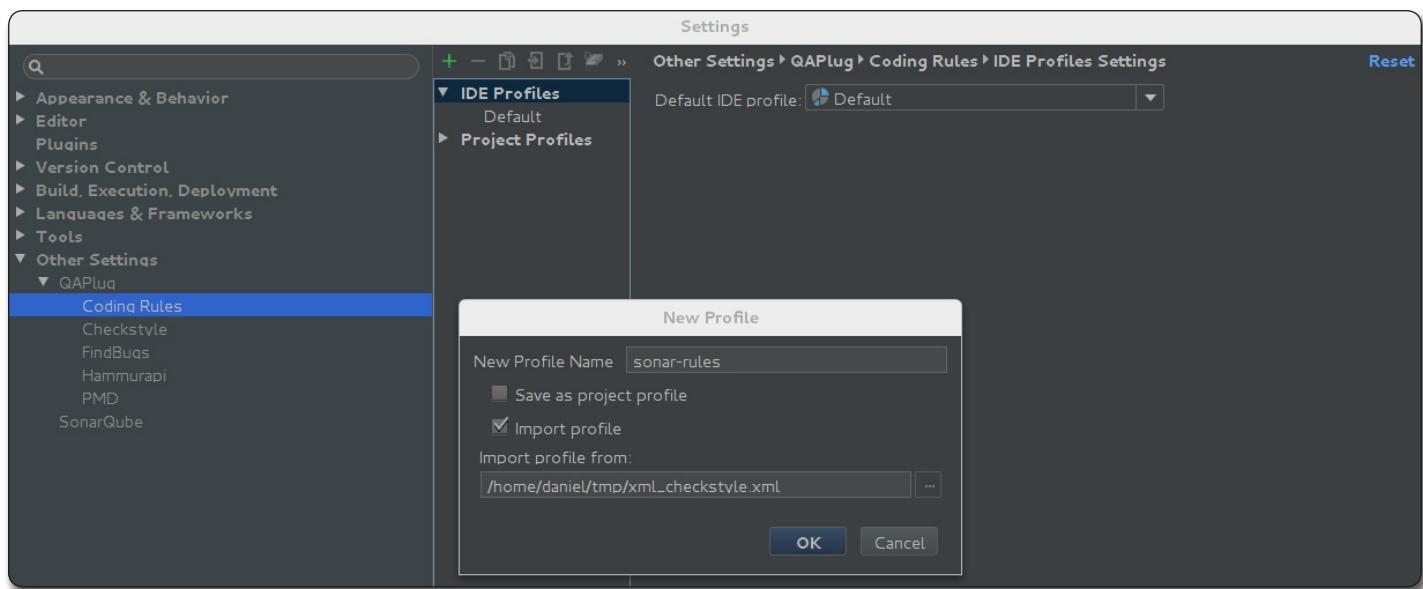


Figura 9. Importação de regras do Nemo para o IntelliJ

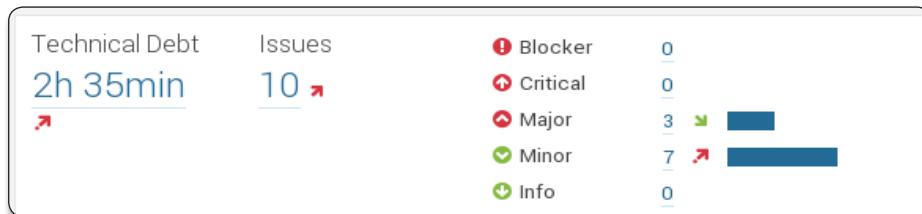


Figura 10. Consolidação de issues por severidade no SonarQube

Observe que foi possível importar apenas um arquivo, mas temos três arquivos a importar (para PMD, Findbugs e Checkstyle). Então, clique com o botão direito sobre o profile criado e selecione *Import* para importar o segundo arquivo XML faltante. Depois, repita o procedimento para que o último arquivo XML tenha sido importado. A tela deste procedimento é exibida na Figura 9.

Ao fim do procedimento, haverá um profile configurado na IDE com a soma das regras de PMD, FindBugs e Checkstyle. Com isto, sua IDE está preparada para verificar códigos-fonte.

Melhorando código mal escrito

Neste ponto, a IDE está configurada com os plugins e as regras da corporação. Então, para que se possa verificar a qualidade de código-fonte de um projeto, basta abri-lo na IDE e repetir o procedimento já descrito no tópico “Realizando a Análise e os Ajustes com IntelliJ”.

Assim, será possível identificar problemas no código-fonte, realizar sua devida correção e verificar, através de uma nova execução, se os problemas foram sanados. Lembre-se de escolher o profile criado, para que a verificação utilize as regras corporativas.

Gerando métricas

Após o devido ciclo de verificação/correção, o código-fonte está num nível de qualidade adequado. Mas existe alguma forma de validar as correções de forma consolidada?

Sim, existe. O SonarQube conta com uma funcionalidade que expõe as métricas do projeto em termos de número de *issues* (que são identificações de linhas de código que não atendem a regras) categorizados por severidade, conforme demonstra a Figura 10. A severidade indica o quanto o issue é crítico em relação a critérios definidos pelos próprios plugins de análise estática. Por exemplo, um bug que pode

causar um problema sério é categorizado pelo Findbugs como um blocker, enquanto um bug que tem pouco impacto pode ser categorizado pelo Findbugs de forma mais leve, como um minor.

É possível redefinir esta classificação no próprio SonarQube (por exemplo, um problema classificado por default como Critical pelo PMD pode ser reclassificado como Minor se a empresa considerar que isto é mais válido para seus padrões internos de qualidade).

A seta vermelha indica aumento do número de problemas (ou issues) daquela severidade em relação à geração de métricas anterior. Já a seta verde indica redução no número de issues. Contudo, para que estas métricas sejam exibidas, elas primeiramente necessitam ser geradas.

O primeiro passo para geração das métricas de um projeto é a adição do arquivo *sonar-project.properties* (que define propriedades gerais do projeto, necessárias para a geração de métricas que serão posteriormente exibidas pelo Sonar) na pasta raiz do projeto. Um exemplo deste é exibido na Listagem 3.

As quatro primeiras linhas são requeridas e falam de dados gerais do projeto e do subdiretório do projeto onde o código-fonte reside. Já a quinta linha especifica a linguagem sendo utilizada e a última linha declara o encoding adotado pelo projeto.

Listagem 3. Conteúdo do arquivo sonar-project.properties.

```
01 sonar.projectKey=demo-static-code-analysis
02 sonar.projectName=Demo Static Code Analysis
03 sonar.projectVersion=1.0
04 sonar.sources=src
05 sonar.language=java
06 sonar.sourceEncoding=UTF-8
```

É importante entender que o SonarQube não gera as métricas automaticamente. Ele apenas as exibe, cabendo a outras ferramentas sua geração. O arquivo *sonar-project.properties* será utilizado por uma ferramenta específica (como o plugin *sonar-maven-plugin* do Maven ou Sonar Runner, explicados a seguir) para gerar um arquivo de métricas do projeto. Este arquivo de métricas é então enviado ao SonarQube, para que exiba seus dados em termos de issues.

Uma das formas de gerar o arquivo de métricas é através do Maven (caso o projeto em questão o utilize). Para isso, basta adicionar o plugin *sonar-maven-plugin* no arquivo *pom.xml* do projeto, conforme a **Listagem 4**.

Listagem 4. Configuração do SonarQube Maven Plugin no pom.xml.

```
01 <project>
02 ...
03   <build>
04     ...
05       <plugins>
06         <plugin>
07           <groupId>org.codehaus.mojo</groupId>
08           <artifactId>sonar-maven-plugin</artifactId>
09           <version>2.6</version>
10         </plugin>
11       ...
12     </plugins>
13   </build>
14 ...
15 </project>
```

Note que nenhuma configuração adicional é requerida. Após isto, o plugin *sonar-maven-plugin* irá gerar o arquivo de métricas durante o build.

Caso o Maven não esteja disponível e a geração das métricas precise ser feita manualmente, existe a opção de executar o Sonar Runner. Esta é uma ferramenta que gera métricas e as exibe diretamente na interface do SonarQube. Para executá-la, após a instalação, basta rodar o script *sonar-runner* a partir do diretório raiz do projeto em análise. O código a seguir mostra um exemplo dos comandos necessários para executar o Sonar Runner, no Linux:

```
[/home/daniel]$ cd ~/dev/java/demo-static-code-analysis
[/home/daniel/dev/java/demo-static-code-analysis]$ ~/apps/sonar-runner/bin/sonar-runner
```

Com isto, as métricas serão geradas baseadas no arquivo *sonar-project.properties* e entregues diretamente ao Sonar. Em seguida,

basta fazer um refresh no navegador para verificar a atualização das métricas.

A análise estática é um importante recurso na busca pela melhoria da qualidade do código. A partir de sua adoção de forma antecipada, é possível obter um ganho expressivo de qualidade em termos de redução de bugs encontrados em homologação/produção. Neste contexto, o uso do SonarQube para inspeção é uma forma eficaz que pode ser aplicada para acompanhar a qualidade do produto. Ao utilizar o SonarQube como centralizador de regras e incorporador de regras à IDE, empresas ganham com uma padronização da qualidade, facilitando a fase de code review (ou mesmo viabilizando-a, caso não seja praticada).

É importante ainda considerar o uso de ferramentas de build e integração contínua (tais como Jenkins e Maven) na adoção da análise estática de código, pois desta forma é possível automatizar a geração de métricas para o SonarQube. Em conjunto, Jenkins e Maven podem ser configurados para, a cada build, realizar a geração automática de métricas e a entrega das métricas ao SonarQube da corporação. Com essa automatização, a organização pode ter um relatório de qualidade do status atual do projeto sem que esforço algum precise ser realizado para manter esses dados atualizados

Autor**Daniel Medeiros de Assis**

daniel.medeiros.assis@gmail.com

<https://br.linkedin.com/in/dmassis>

É mestrado em Engenharia de Software no IPT/USP. Possui mais de 15 anos de experiência em desenvolvimento de software, com especialização em tecnologias web, design e qualidade de código, e processos ágeis de desenvolvimento.

**Links:****Site do projeto FindBugs.**

<http://findbugs.sourceforge.net/>

Site do projeto PMD.

<http://pmd.sourceforge.net/>

Site da ferramenta CheckStyle.

<http://checkstyle.sourceforge.net/>

Site da ferramenta SonarQube.

<http://www.sonarqube.org/>

Nemo, demonstração online do SonarQube.

<http://nemo.sonarqube.org/>

Código-fonte utilizado no artigo.

<http://github.com/medeiros/demo-static-code-analysis>

Codehaus Sonar Maven Plugin.

<http://mojo.codehaus.org/sonar-maven-plugin/plugin-info.html>

Criando um blog com MongoDB e Spark Framework

Construa um blog com o mínimo esforço com Spark Framework, gastando mais tempo com o código e menos com a configuração do MongoDB

Não é sempre que temos a necessidade de construir uma aplicação de nível empresarial, que requer um maior esforço e mais tempo para se construir. Em alguns casos, queremos desenvolver algo simples como, por exemplo, um micro blog pessoal, sem precisar se preocupar em ter que escalar a aplicação em todas as direções.

Muitas vezes queremos apenas algo rodando em produção em poucos minutos. Entretanto, depender de um servidor web Java, como o Tomcat, requer um trabalho nas partes de configuração, manutenção e deployment do projeto. Nesses casos, usar um servidor Jetty embutido facilita bastante a vida do desenvolvedor, principalmente na hora da execução da aplicação.

Outra parte do projeto que também costumamos investir bastante tempo é na camada de acesso ao banco de dados. Neste contexto, o MongoDB é uma solução que se destaca quando buscamos produtividade e performance. Um dos seus principais objetivos é manter o foco do desenvolvedor nos objetos da aplicação. Outro diferencial do MongoDB é que sua estrutura segue o modelo *Schemaless*, ou seja, o banco de dados não tem controle sobre a estrutura dos dados como, por exemplo, o nome das colunas e/ou a quantidade de colunas que um documento pode ter.

Além disso, para permitir uma maior performance, principalmente em ambientes distribuídos, o MongoDB não oferece algumas das principais funcionalidades encontradas no mundo SQL, como *joins* e *transactions*. Sendo assim, quando necessário, é de responsabilidade da aplicação viabilizar e controlar essas funcionalidades.

Com base nestas informações, veremos neste artigo

Fique por dentro

Este artigo é útil por apresentar um tutorial sobre a construção de um micro blog com agilidade e facilidade. Para isso, veremos como utilizar o framework Spark, perfeito para criar aplicações web em Java que estarão rodando em minutos e que não precisam escalar horizontalmente e verticalmente. Além disso, como solução para persistência dos dados, adotaremos o MongoDB, também abordado neste artigo.

como utilizar o Spark Framework para construir um micro blog completo. Para isso, adotaremos, além do framework Spark, o MongoDB, que, juntos, possibilitam criar aplicações web com o menor esforço possível, trocando funcionalidades empresariais, como controle de transações, por facilidade e agilidade. Neste cenário, como um importante complemento, veremos também como adotar o FreeMarker para gerar templates HTML.

Conhecendo o MongoDB

O MongoDB é um banco de dados NoSQL orientado a documentos que armazena arquivos no formato BSON, que de forma simples pode ser definido como uma versão binária de um arquivo JSON.

No MongoDB chamamos de *collection* as estruturas que são parecidas com as tabelas dos bancos relacionais. Já os dados que são inseridos nestas coleções são chamados de documentos. Estes documentos, por sua vez, são formados por um mapa de chave/valor, onde cada chave seria como se fosse uma coluna de uma tabela.

Além disso, cada documento contido em uma coleção deve possuir obrigatoriamente uma chave única chamada `_id`, que funciona da mesma forma que uma *primary key* de um banco

relacional. Por padrão, a chave `_id` é gerada automaticamente pelo MongoDB. No banco de dados MySQL, uma situação similar ocorre quando selecionamos a opção `AUTO_INCREMENT` na criação de uma PK.

Assim como em uma tabela, uma coleção serve para guardar documentos que são comuns entre si, como uma tabela `Usuarios`, que possui colunas referentes à entidade `Usuario`, como `nome`, `email` e `senha`. Contudo, o MongoDB não possui um *Schema* para os dados que estão sendo inseridos nas coleções. Isso significa que é possível, por exemplo, guardar documentos referentes a posts de um blog em uma coleção de `Usuarios`. Porém, isso é uma péssima prática e apenas dificultará a recuperação dos dados do banco.

Deste modo, normalmente precisamos ter diversas coleções em uma aplicação. Este conjunto de coleções é mantido por um container físico denominado *database* que, por sua vez, é análogo a um também chamado de *database* do mundo relacional. Na maior parte dos casos, em um banco MongoDB temos um *database* para cada aplicação.

Contudo, as duas principais qualidades do MongoDB são a alta produtividade e a possibilidade de lidar com Big Data, pois assim como a maioria dos bancos NoSQL, este já foi criado com o intuito de ser facilmente configurado em um ambiente distribuído.

Introdução ao framework Spark

De acordo com a própria descrição do site do Spark, este framework foi fortemente inspirado no Sinatra, uma DSL (vide **BOX 1**) que permite criar aplicações web em Ruby com o mínimo esforço, sem ter que se preocupar muito com configurações e *boilerplate code* (vide **BOX 2**), como quando utilizamos o Spring MVC.

BOX 1. DSL

DSL, ou Domain-Specific-Language, é um tipo de linguagem de programação dedicada a um domínio de problema particular, específico para um cenário. O SQL, por exemplo, é uma DSL para consultas em bancos de dados relacionais. Neste caso, o problema seria a consulta e o domínio do banco de dados relacional.

BOX 2. Boilerplate code

São códigos que sempre temos que escrever quando desenvolvemos alguma coisa. Por exemplo: no Spring MVC, temos que escrever uma configuração padrão no arquivo `web.xml`, sendo que na maioria das vezes, é exatamente o mesmo código, exceto por algumas pequenas diferenças.

O Spark é um framework de código aberto que serve como uma alternativa aos diversos frameworks de desenvolvimento de aplicações web em Java, como JSF, Spring MVC, Play Framework, entre outros. Por padrão, ele roda em um servidor Jetty embutido, mas pode ser configurado para ser executado em qualquer servidor web como, por exemplo, no Tomcat.

A arquitetura de um software feito com o Spark Framework segue o padrão MVC, onde separamos a aplicação em no mínimo três camadas (Modelo, Visão e Controle). Seu principal objetivo é separar a informação e as regras de negócio da interface com a qual o usuário interage.

Um dos poucos códigos *boilerplate* de uma aplicação desenvolvida com Spark é a configuração das rotas. A rota é um componente responsável por transformar cada entrada HTTP, que chamamos de *request*, em uma ação, a qual pode ou não retornar uma resposta. Cada rota é constituída de três partes principais, a saber:

- **Verbo:** Um verbo definido pela especificação HTTP, como GET, POST, PUST, DELETE;
- **Callback:** Uma função que recebe uma *request* e uma *response* e retorna uma **String** que pode ser utilizada por engines de templates para renderizar páginas HTML;
- **Caminho:** O caminho se refere ao endereço de rede que, quando acessado, irá efetuar uma chamada ao *callback*, que irá executar a ação definida no callback.

Sua mais recente versão, usada na escrita deste artigo, tornou possível escrever as rotas com Java 8. Essa nova funcionalidade reduziu ainda mais a quantidade de código necessário. Assim, agora é possível ter uma aplicação *Hello World* rodando com apenas cinco linhas de código. Veja o exemplo na **Listagem 1**.

Listagem 1. Exemplo de Hello World com o Spark.

```
import static spark.Spark.*;
public class HelloWorld {
    public static void main(String[] args) {
        get("/hello", (req, res) -> "Hello World");
    }
}
```

No projeto de aplicação web que vamos construir no decorrer deste artigo, usaremos a engine de templates FreeMarker para gerar os códigos HTML. Entretanto, não faz parte do escopo do mesmo se aprofundar neste assunto. O código dos templates pode ser encontrado junto com o código fonte deste artigo, disponibilizado para download na página desta edição.

Criando o projeto

Para darmos início ao desenvolvimento do nosso micro blog, é necessário que tenhamos um ambiente previamente configurado com o Java 8, a IDE Eclipse, o banco de dados MongoDB e a ferramenta de automação de build Gradle.

Nota

Para conhecimento do leitor, é possível utilizar outras IDE, como o NetBeans, porém isto requer leves ajustes na configuração que analisaremos a seguir.

Apesar disso, a configuração e instalação destas ferramentas não fazem parte do escopo deste artigo, pois dessa forma não teríamos espaço suficiente para nos aprofundarmos no tutorial dos códigos do Spark e da persistência de dados com o driver do MongoDB.

Com tudo configurado, podemos iniciar a criação do projeto. Para isso, crie uma estrutura de pastas conforme a exibida na **Figura 1**, onde a pasta raiz é chamada de *MongoSpark*.

Criando um blog com MongoDB e Spark Framework

Agora, com o objetivo de utilizar a ferramenta de automação Gradle para gerenciar as dependências necessárias à construção do aplicativo, crie dentro da pasta *MongoSpark* o arquivo *build.gradle*, com o conteúdo igual ao exibido na **Listagem 2**.

```
/Users/dbalduini/Projects/DevMedia/prototipos/MongoSpark
|---src
|   |---main
|   |   |---java
|   |   |---resources
|   |---test
|   |   |---java
|   |   |---resources
```

Figura 1. Estrutura de pastas do projeto

Listagem 2. Conteúdo do arquivo build.gradle.

```
apply plugin:'java'
apply plugin:'eclipse'
apply plugin:'application'

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile "com.sparkjava:spark-core:2.1"
    compile "com.sparkjava:spark-template-freemarker:2.0.0"
    compile "org.mongodb:mongo-java-driver:2.12.5"
}

jar {
    baseName ='mongo-spark'
    version = '1.0'
}

mainClassName = "br.com.devmedia.mongospark.Main"
```

Nas primeiras linhas deste arquivo, definimos três *plugins*, sendo eles: *java*, *eclipse* e *application*. O primeiro informa que vamos usar o Java como linguagem de programação, o segundo permite utilizarmos a IDE Eclipse e o último serve apenas para que possamos utilizar o Gradle para, no final do desenvolvimento, executar a classe **Main** da aplicação, a qual informamos na propriedade **mainClassName**.

As dependências são especificadas na propriedade *dependencies*, onde definimos que vamos precisar do Spark Core, do suporte ao FreeMarker do Spark e do driver Java do MongoDB. A última propriedade, de nome *jar*, declara o nome do JAR quando fizermos o build do projeto.

Importando o projeto para o Eclipse

Com a configuração do Gradle concluída, acesse o terminal (Linux) ou o prompt de comando (Windows). Em seguida, execute o comando a seguir dentro da pasta *MongoSpark*:

```
gradle eclipse
```

Este comando irá utilizar o plugin do Eclipse para gerar os arquivos de configuração desta IDE, os quais permitem importarmos o projeto para dentro dela.

Para importar o código fonte para o Eclipse, abra a IDE, acesse o menu *File* e clique em *Import*. Na janela apresentada ao realizar esse comportamento, selecione o item *General > Existing Projects into Workspace* e, em seguida, clique em *Next*. Na próxima janela, em *Select root directory*, clique em *Browse*. Então procure e selecione a pasta *MongoSpark*, criada anteriormente, e clique em *Finish*. Se tudo ocorreu corretamente, seu workspace deve estar parecido com o exibido na **Figura 2**.

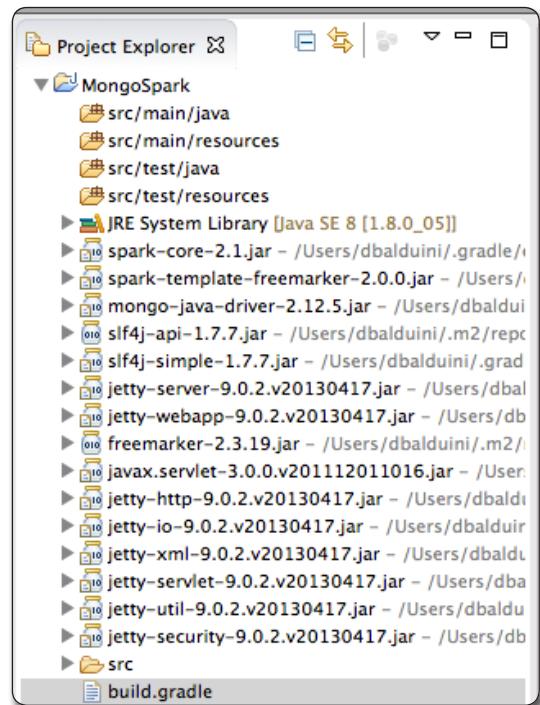


Figura 2. Projeto MongoSpark importado no Eclipse

Desenvolvendo a aplicação

Agora que já temos todo o ambiente configurado, podemos partir para o desenvolvimento da aplicação proposta. Feito isso, ao final deste tópico teremos um micro blog completamente funcional, além de conhecer o framework Spark e como utilizar o driver em Java do MongoDB para fazer as operações básicas de um CRUD.

Acesso ao banco de dados do MongoDB

As operações suportadas pelas coleções do MongoDB podem ser realizadas após obter uma referência à **database**, que iremos chamar de **blog**. Como este é um acesso ao serviço do MongoDB, e que será necessário em diversas partes da aplicação, vamos encapsular esta chamada na classe **ServiceLocator**, conforme o código da **Listagem 3**.

No construtor desta classe acessamos uma única vez o *client* do MongoDB, através da variável **mongoClient**, que nos permite conectar com o serviço de banco de dados.

Em seguida, usamos esta variável para instanciar o atributo **database** da classe **ServiceLocator**. Por ser do tipo **DB**, é através deste atributo que conseguimos acessar as coleções, possibilitando criar, buscar, remover e atualizar os documentos.

Vale ressaltar que a classe **ServiceLocator** foi desenvolvida de acordo com o padrão de projeto *Singleton*. Sendo assim, só existirá uma instância de **ServiceLocator** durante todo o ciclo de vida da aplicação.

Listagem 3. Código da classe ServiceLocator.

```
package br.com.devmedia.mongospark;

import java.io.IOException;

import com.mongodb.DB;
import com.mongodb.MongoClient;
import com.mongodb.MongoClientURI;

public class ServiceLocator {

    private static ServiceLocator instance;

    private DB database;

    private ServiceLocator() {
        try {
            final MongoClient mongoClient = new MongoClient(
                new MongoClientURI("mongodb://localhost"));
            database = mongoClient.getDB("blog");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public static ServiceLocator getInstance() {
        if (instance == null) {
            instance = new ServiceLocator();
        }
        return instance;
    }

    public DB getDB() {
        return this.database;
    }
}
```

Operações de CRUD com DAOs

O Objeto de Acesso a Dados, ou simplesmente DAO (*Data Access Object*), é um padrão de projeto que tem por objetivo centralizar as operações relacionadas à persistência de dados. Com o intuito de desacoplar a aplicação da fonte de dados, este padrão deve ser acompanhado de interfaces. Contudo, como não temos uma iminente mudança da fonte de dados em nossa aplicação exemplo, vamos pular a definição de interfaces e criar DAOs apenas para lidar com o MongoDB.

Algo comum em todos os DAOs são as operações de CRUD: criar, buscar, atualizar e deletar documentos. Com este conhecimento, vamos começar criando uma classe abstrata com apenas dois métodos, sendo um para criar documentos (**create()**) e outro para buscá-los por ID (**findById()**), conforme o código da Listagem 4.

Listagem 4. Código da classe abstrata DAO.

```
package br.com.devmedia.mongospark.dao;

import br.com.devmedia.mongospark.ServiceLocator;

import com.mongodb.BasicDBObject;
import com.mongodb.DBCollection;
import com.mongodbDBObject;
import com.mongodb.DuplicateKeyException;

public abstract class DAO {

    protected final DBCollection collection;

    public DAO(String collectionName) {
        this.collection = ServiceLocator
            .getInstance()
            .getDB()
            .getCollection(collectionName);
    }

    publicDBObject findById(String id) {
        return collection.findOne(new BasicDBObject("_id", id));
    }

    public boolean create(BasicDBObject doc) {
        try {
            collection.insert(doc);
            return true;
        } catch (DuplicateKeyException e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

Observe que no construtor da classe DAO pedimos que seja informado o nome da coleção que a classe filha deverá acessar. É com este nome que obtemos acesso às coleções do MongoDB. Com a coleção em mãos, a referência a ela é guardada na variável **collection**, a qual será utilizada para acessar todas as operações providas pelo driver do MongoDB.

O primeiro método definido nesta classe é o **findById()**, que recebe um **id** como parâmetro. Em seu corpo, construída com o auxílio da classe **BasicDBObject**, criamos uma query – a qual também é conhecida como seletor no universo do MongoDB – para buscar uma chave **_id** com o valor contido no parâmetro **id**. Essa query é então passada para o método **findOne()** da **collection**, que retorna um único documento, representado pela classe **DBObject**, ou então **null** caso nenhum resultado seja encontrado.

Com o driver em Java do MongoDB que adicionamos nas dependências do projeto, praticamente todas as operações são construídas a partir da classe **BasicDBObject**, que funciona de forma semelhante à interface **Map**, já que trabalhamos com chaves que apontam para valores. Esta classe permite enviar documentos para serem inseridos no banco de dados, construir queries para realizar buscas, efetuar atualizações, entre outros. Como **BasicDBObject** é mutável, assim como o **StringBuilder** do Java, podemos chamar seu método **append()** para passarmos mais chaves e valores. Veremos mais detalhes sobre como usar o **append()** a seguir.

Por fim, definimos o método **create()**, que recebe como parâmetro um **BasicDBObject**. Neste caso, entretanto, ao invés de ser um seletor, **BasicDBObject** representa o documento a ser inserido no banco de dados. Se tudo ocorrer com sucesso, retornamos **true**. Caso ocorra uma exceção do tipo **DuplicateKeyException**, retornamos **false**; algo que poderia ocorrer, por exemplo, na tentativa de inserir um *permalink* que já existe.

Nota

Em um blog, um *permalink* é como chamamos uma URL que aponta exatamente para a localização de um post. Por esta razão, todo *permalink* deve ser único na aplicação.

A primeira classe filha de DAO que vamos implementar tem como objetivo adicionar e autenticar os usuários da aplicação. Sendo assim, crie a classe **UserDAO** conforme exibido na **Listagem 5**. Como a classe pai, **DAO**, requer o nome da coleção em seu construtor, informamos que o nome da coleção que irá armazenar os dados da entidade **Usuario** será **users**.

Além dos métodos herdados de DAO, definimos em **UserDAO** mais dois métodos. O primeiro é o **addUser()**, responsável por criar os usuários. Com os dois parâmetros recebidos, criamos um **BasicDBObject**, o qual iremos passar para a operação de **insert()** do driver do MongoDB. Aqui utilizamos o método **append()** para popular o documento a ser inserido, passando a chave **_id** com o valor do parâmetro **username** e a chave **password** com o valor da senha.

Por fim, chamamos o método **create()** herdado de DAO para finalmente salvar este novo documento na coleção **users**. Como no MongoDB toda chave **_id** deve ser única, se tentarmos criar um **username** repetido, uma **DuplicateKeyException** será lançada, e então o valor **false** será retornado.

O segundo método, **authenticate()**, serve para autenticar os usuários. Nele tentamos consultar um usuário por **username**. Já que na coleção **users** o **_id** é o próprio **username** do usuário, então podemos utilizar o método **findById()** herdado de **DAO**.

Caso o retorno deste método seja **null**, ou a senha informada não bater com a recebida da consulta, então dizemos que o usuário não foi autenticado com sucesso. Caso contrário, um usuário autenticado é retornado.

O próximo DAO que iremos construir será responsável pelo gerenciamento de sessão da aplicação. Dessa forma, crie a classe **SessionDAO** de acordo com o código da **Listagem 6**. Assim como fizemos em **UserDAO**, informamos o nome da coleção que desejamos acessar com esta DAO em seu construtor, só que neste caso a coleção se chama **sessions**.

O último DAO que precisamos criar é o responsável pela parte mais importante de um blog, os posts. Portanto, crie a classe **PostDAO** com o código indicado na **Listagem 7**. E como verificado no construtor, eles serão armazenados em uma coleção denominada **posts**.

No primeiro método **findByPermalink()**, usamos o parâmetro **permalink** para buscar pelo post desejado.

Listagem 5. Código da classe UserDAO.

```
package br.com.devmedia.mongospark.dao;

import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;

public class UserDAO extends DAO {

    public UserDAO() {
        super("users");
    }

    public boolean addUser(String username, String password) {
        BasicDBObject user = new BasicDBObject();
        user.append("_id", username).append("password", password);
        return super.create(user);
    }

    public DBObject authenticate(String username, String password) {
        DBObject user = super.findById(username);

        if (user == null) {
            System.out.println("Usuário não existe");
            return null;
        }

        if (!password.equals(user.get("password").toString())) {
            System.out.println("Senha inválida");
            return null;
        }

        return user;
    }
}
```

Listagem 6. Código da classe abstrata SessionDAO.

```
package br.com.devmedia.mongospark.dao;

import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;

public class SessionDAO extends DAO {

    public SessionDAO() {
        super("sessions");
    }

    public void startSession(String username, String sessionID) {
        BasicDBObject session = new BasicDBObject();
        session.append("_id", sessionID);
        session.append("username", username);
        collection.insert(session);
    }

    public void endSession(String sessionID) {
        collection.remove(new BasicDBObject("_id", sessionID));
    }

    public String getUsernameBySessionId(String sessionID) {
        DBObject session = super.findById(sessionID);
        if (session == null) {
            return null;
        } else {
            return session.get("username").toString();
        }
    }
}
```

Listagem 7. Código da classe PostDAO.

```
package br.com.devmedia.mongospark.dao;

import java.util.Date;
import java.util.List;

import com.mongodb.*;

public class PostDAO extends DAO {

    public PostDAO() {
        super("posts");
    }

    public DBObject findByPermalink(String permalink) {
        BasicDBObject query = new BasicDBObject("permalink", permalink);
        DDBObject post = collection.findOne(query);
        return post;
    }

    public List<DBObject> findByDateDescending(int limit) {
        List<DBObject> posts = null;

        DBCursor cursor = collection.find()
            .sort(new BasicDBObject("date", -1))
            .limit(limit);

        try {
            posts = cursor.toArray();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            cursor.close();
        }
        return posts;
    }

    public String addPost(String title, String body, List<?> tags, String username) {
        String permalink = title.replaceAll("\\s+", "_");
        permalink = permalink.replaceAll("\\W","");
        permalink = permalink.toLowerCase();

        BasicDBObject post = new BasicDBObject();
        post.append("title", title);
        post.append("author", username);
        post.append("body", body);
        post.append("permalink", permalink);
        post.append("tags", tags);
        post.append("date", new Date());

        try {
            collection.insert(post);
        } catch (MongoException e) {
            e.printStackTrace();
        }
        return permalink;
    }
}
```

Em seguida, em `findByDateDescending()`, queremos retornar uma lista de posts ordenada pela data de criação, de maneira descendente. Note que em vez de retornar um `DBObject` como nas outras buscas que vimos, a função `find()` da `collection` retorna um objeto do tipo `DBCursor`, que pode ser usado para iterar nos documentos contidos na coleção. Ainda na mesma linha de código,

solicitamos para o driver do MongoDB que o cursor seja ordenado pelo campo `date`, de maneira descendente. Caso tivéssemos passado um número positivo, como 1, o resultado seria ordenado de forma crescente. Logo depois fazemos uma chamada ao método `limit()` para limitar a quantidade de documentos retornados de acordo com o parâmetro informado. Por fim, transformamos o cursor em uma lista de `DBObject` com a chamada da função `cursor.toArray()`.

Antes de prosseguir para a explicação do método `addPost()`, precisamos entender melhor o funcionamento do atributo `permalink` em um blog. O `permalink` é usado para recuperar os detalhes de um post, como autor, corpo e data de criação. Para isso, iremos passá-lo para a rota de um controlador como um parâmetro nomeado, o que significa que é possível passar parâmetros no próprio caminho da rota, ficando algo como, por exemplo, `/posts/Meu_Primeiro_Post`, onde “Meu_Primeiro_Post” é o `permalink`.

Dando continuidade, o próximo e último método definido em `PostDAO` é o `addPost()`. Aqui, quatro parâmetros são esperados para que possamos inserir um post no banco de dados, a saber: o título, o corpo, uma lista de tags e o nome do usuário autor do post. Note que nas primeiras três linhas deste método implementamos um tratamento para o parâmetro `title`, de forma que sejam removidos espaços e caracteres em branco, para que assim tenhamos um `permalink` válido.

Por fim, através da classe `BasicDBObject`, populamos as chaves e valores do documento referente ao post. Em seguida, ainda em `addPost()`, é feita uma chamada ao método `insert()` da `collection` para persistir o documento na base de dados. A **Figura 3** apresenta a página relacionada à criação de um post.

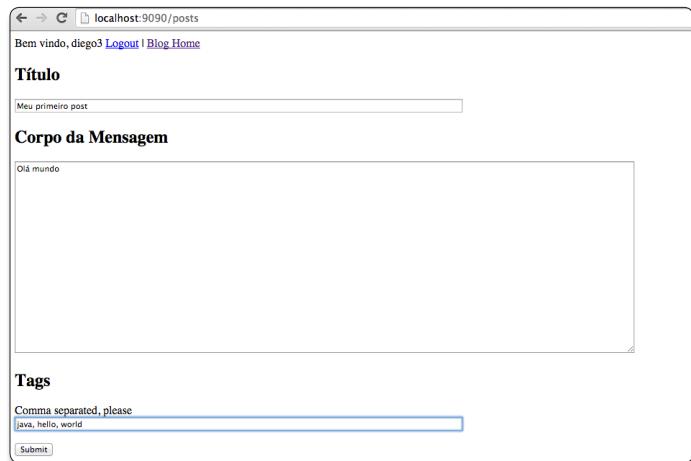


Figura 3. Página para criação de posts

Criando os controladores do blog

Com a camada de Modelo concluída, o próximo passo que iremos tomar é criar os controladores. O objetivo principal de um controlador é basicamente transformar uma *request* em uma *response*. Qualquer outro tipo de processamento, como acesso a dados, é passado para uma classe de alguma outra camada como, por exemplo, as DAOs que criamos anteriormente.

Criando um blog com MongoDB e Spark Framework

O primeiro controlador que vamos desenvolver é o **UserController**, apresentado na **Listagem 8**. Ele será o responsável pelas requisições de cadastro de usuários. Como o Spark Framework disponibiliza todos os seus recursos, como os métodos HTTP **get()** e **post()**, de forma estática, optamos por importar os membros da classe **Spark** também de forma estática, durante a chamada **import static spark.Spark.*;**.

No Spark, os métodos **get()** e **post()** recebem como parâmetros um caminho, uma função de *callback*, a qual é responsável por transformar a *request* em um *response* através de uma expressão lambda, e, opcionalmente, um *template engine*, que será explicado a seguir. A diferença entre esses dois métodos é bastante simples. Quando queremos recuperar algum recurso do servidor, como listar os usuários, por exemplo, utilizamos o método **get()**, e quando precisamos enviar um formulário HTTP para cadastrar algum recurso, usamos o **post()**. Esta situação é bastante parecida com a que ocorre quando desenvolvemos com Servlets e precisamos implementar os métodos **doGet()** e **doPost()**.

Depois de importar os métodos HTTP do Spark, podemos iniciar as rotas da aplicação referentes aos usuários através da chamada

ao método estático **initRoutes()**. Para isso, vamos definir duas rotas: GET */users* e POST */users*. Na primeira, desejamos retornar para o solicitante do recurso a página HTML contendo um formulário para cadastro de um usuário em branco, a ser preenchido e enviado posteriormente para cadastrar um usuário novo.

Contudo, em vez de escrevermos na mão uma **String** com código HTML como fazemos com os Servlets, faremos uso de um *template engine*. O Spark oferece suporte a três *template engines*, a saber: FreeMarker, Velocity e Mustache. Com a dependência do FreeMarker que adicionamos anteriormente, basta passar uma instância de **FreeMarkerEngine** como terceiro parâmetro do método **get()** para ativar esta *engine*.

Agora que informamos ao Spark que iremos utilizar o FreeMarker, basta retornar na função de *callback* uma instância de

Nota

Para que o template engine funcione é necessário que todos os templates estejam localizados na pasta `src/main/resources/spark/template/freemarker`. A implementação dos templates pode ser encontrada junto com o código fonte deste artigo.

Listagem 8. Código da classe UserController.

```
package br.com.devmedia.mongospark.controllers;

import br.com.devmedia.mongospark.dao.UserDAO;
import spark.ModelAndView;
import spark.template.freemarker.FreeMarkerEngine;
import freemarker.template.SimpleHash;

import static spark.Spark.*;

public class UserController {

    public static final String USER_TEMPLATE = "user_template.ftl";

    public static void initRoutes() {
        UserDAO userDAO = new UserDAO();

        UserController userController = new UserController();
        SessionController sessionController = new SessionController();

        // Exibe o form de cadastro de usuário
        get("/users", (request, response) -> {
            SimpleHash root = new SimpleHash();

            // Inicializa os valores do form
            root.put("username","");
            root.put("password","");
            root.put("password_error","");
            root.put("username_error","");
            root.put("verify_error","");

            return new ModelAndView(root, USER_TEMPLATE);
        }, new FreeMarkerEngine());

        // Controla o form de cadastro de usuário
        post("/users", (request, response) -> {
            SimpleHash root = new SimpleHash();

            String username = request.queryParams("username");
            String password = request.queryParams("password");
            String verify = request.queryParams("verify");

            root.put("username", username);
            root.put("password", password);
            root.put("verify", verify);

            if(userController.validaFormUsuario(username, password, verify)){
                System.out.println("Criando usuário: " + username + " " + password);
                boolean ok = userDAO.addUser(username, password);
                if(ok){
                    root.put("username_error", "Username já está sendo usado");
                    return new ModelAndView(root, USER_TEMPLATE);
                } else {
                    sessionController.iniciarSessao(username, request);
                    response.redirect("/welcome");
                    return null;
                }
            } else {
                halt(400, "Form com campos inválidos");
                return new ModelAndView(root, USER_TEMPLATE);
            }
        }, new FreeMarkerEngine());
    }

    public boolean validaFormUsuario(String username, String password, String verify) {
        if(preenchido(username) && preenchido(password) && preenchido(verify)){
            if(password.equals(verify))
                return true;
            else
                return false;
        } else {
            return false;
        }
    }

    public boolean preenchido(String s) {
        return s != null && !s.equals("");
    }
}
```

ModelAndView e passar como argumentos de seu construtor o nome do template a ser renderizado e um mapa de chave/valor. Como exemplo de uso desse recurso, no elemento HTML `<input type="text" name="username" value="${username}">`, o texto `${username}` será substituído pelo valor referente à chave `username` do mapa.

Concluímos o nosso primeiro controlador desenvolvendo o método `post()`, que irá receber os dados do formulário de cadastro de usuários contidos no elemento `<form>` do HTML. Com os valores recuperados através da chamada `request.queryParams("<parametro>")`, verificamos se os dados são válidos. Se tudo ocorrer com sucesso, o usuário é salvo no banco e redirecionado para a rota `GET /welcome` já com uma sessão. Caso contrário, o usuário será retornado para a página de cadastrado junto com o erro gerado.

O segundo controlador que iremos criar é o responsável pelas sessões. Dessa forma, crie a classe **SessionController** conforme a **Listagem 9**.

Listagem 9. Código da classe SessionController.

```
package br.com.devmedia.mongospark.controllers;

import java.util.HashMap;
import java.util.Map;
import br.com.devmedia.mongospark.dao.SessionDAO;
import br.com.devmedia.mongospark.dao.UserDAO;
import spark.ModelAndView;
import spark.Request;
import spark.Session;
import spark.template.freemarker.FreeMarkerEngine;
import com.mongodb.DBObject;
import static spark.Spark.*;

public class SessionController {

    public static final String LOGIN_TEMPLATE = "login.ftl";

    private UserDAO userDAO = new UserDAO();
    private SessionDAO sessionDAO = new SessionDAO();

    public static void initRoutes() {
        SessionController controller = new SessionController();

        // Exibe a página de login
        get("/login", (request, response) -> {
            Map<String, Object> attributes = new HashMap<>();
            attributes.put("username","");
            attributes.put("login_error","");
            return new ModelAndView(attributes, LOGIN_TEMPLATE);
        }, new FreeMarkerEngine());

        // Tenta autenticar o usuário
        post("/login", (request, response) -> {
            Map<String, Object> attributes = new HashMap<>();

            String username = request.queryParams("username");
            String password = request.queryParams("password");

            if (controller.autenticar(username, password)) {
                controller.iniciarSessao(username, request);
                response.redirect("/welcome");
            } else {
                attributes.put("username", username);
                attributes.put("password", password);
                attributes.put("login_error", "Login inválido");
            }
        }, new FreeMarkerEngine());
    }

    public String getUser(Request request) {
        Session session = request.session();
        if(session == null){
            return null;
        } else {
            String username = sessionDAO.getUsernameBySessionId(session.id());
            return username;
        }
    }

    public boolean autenticar(String username, String password) {
        DBObject user = userDAO.authenticate(username, password);
        return user != null;
    }

    public void iniciarSessao(String username, Request request) {
        Session session = request.session(true);
        String sessionId = session.id();
        sessionDAO.startSession(username, sessionId);
    }
}
```

Através deste controlador seremos capazes de efetuar autenticação, login e logout dos usuários.

Seguindo o mesmo estilo de desenvolvimento do **UserController**, inicializamos as rotas dentro do método estático `initRoutes()`. Ainda nele, definimos primeiramente a rota `GET /login`, a qual retorna a página onde o usuário deve preencher os dados de usuário e senha do formulário de login.

A segunda rota, `POST /login`, recebe o formulário preenchido para poder validar se o usuário realmente existe no sistema.

Nota

As expressões lambda são uma funcionalidade introduzida no Java 8. Baseada em linguagens funcionais, ela torna possível passar funções como parâmetros de outras funções. Com isso, métodos que esperavam implementações de classes anônimas como, por exemplo, a interface `ActionListener`, começam a poder receber lambdas, reduzindo em muitos casos a quantidade de linhas de código a ser implementada.

Criando um blog com MongoDB e Spark Framework

Listagem 10. Código da classe BlogController - Parte 1.

```
package br.com.devmedia.mongospark.controllers;

//imports omitidos...

public class BlogController {

    public static final String BLOG_TEMPLATE = "blog_template.ftl";
    public static final String ENTRY_TEMPLATE = "entry_template.ftl";
    public static final String POST_TEMPLATE = "post_template.ftl";
    public static final String WELCOME_TEMPLATE = "welcome.ftl";

    public static void initRoutes() {
        TemplateEngine engine = new FreeMarkerEngine();
        SessionDAO sessionDAO = new SessionDAO();
        PostDAO postDAO = new PostDAO();
        SessionController sessionController = new SessionController();

        // Home page do Blog
        get("/", (request, response) -> {
            SimpleHash root = new SimpleHash();
            List<DBObject> posts = postDAO.findByDateDescending(10);
            root.put("myposts", posts);
            Session session = request.session();
            String username = sessionDAO.getUsernameBySessionId(session.id());
            if (username == null) {
                root.put("username", username);
            }
            return new ModelAndView(root, BLOG_TEMPLATE);
        }, engine);

        get("/welcome", (request, response) -> {
            String username = sessionController.getUser(request);
            if (username == null) {
                response.redirect("/users");
                return null;
            } else {
                SimpleHash root = new SimpleHash();
                root.put("username", username);
                return new ModelAndView(root, WELCOME_TEMPLATE);
            }
        }, engine);
    }

    // Exibe os detalhes de um post
    get("/posts/:permalink", (request, response) -> {
        String permalink = request.params(":permalink");

        DBObject post = postDAO.findByPermalink(permalink);
        if (post == null) {
            halt(404, "Post não encontrado");
            return null;
        } else {
            SimpleHash root = new SimpleHash();
            root.put("post", post);
            return new ModelAndView(root, ENTRY_TEMPLATE);
        }
    }, engine);

    // Exibe o form de cadastro de posts
    get("/posts", (request, response) -> {
        String username = sessionController.getUser(request);
        if (username == null) {
            response.redirect("/login");
            return null;
        } else {
            SimpleHash root = new SimpleHash();
            root.put("username", username);
            return new ModelAndView(root, POST_TEMPLATE);
        }
    }, engine);
}
```

Se o método **autenticar()** retornar **true**, a sessão é criada e o usuário é redirecionado para a rota */welcome*. Caso contrário, o erro *Login invalido* é apresentado na página com o formulário de login.

A terceira e última rota, *GET /logout*, quando invocada, remove a sessão do usuário do banco e depois a invalida, impossibilitando que o usuário acesse as páginas protegidas até que entre com o usuário e senha no sistema novamente.

Para concluir a parte dos controladores, iremos criar aquele que será responsável pelas rotas relacionadas ao blog em si. Além das rotas referentes aos posts, teremos também uma para acessar a *home page* e outra para acessar a página *welcome*, para a qual o usuário será redirecionado assim que se logar no sistema. Sendo assim, crie a classe **BlogController** com o código apresentado nas **Listagens 10 e 11**.

A primeira rota, *GET /*, irá mostrar a *home page* do blog para o usuário. Nesta rota listamos os dez últimos posts ordenados pela data de criação. Em seguida, definimos a rota *GET /welcome*. Nesta página o usuário logado terá acesso ao link de criação de posts e também ao de logout do sistema.

A próxima rota, *GET /posts/:permalink*, serve para exibir os detalhes do post, como autor e texto. Observe que nesta rota o recurso **permalink** possui o sinal de pontuação dois pontos (“：“)

em seu prefixo. Isso faz com que, em vez de ser a continuação de um caminho da URL, ele se torne um parâmetro. Através deste parâmetro podemos recuperar informações do post para que possamos exibir seus detalhes na página HTML renderizada pelo FreeMarker. Caso nenhum post seja encontrado, com a chamada do método **halt()** retornamos o código HTTP 404 (*Not Found*), além da mensagem do ocorrido.

Na próxima rota, *GET /posts*, será retornado o formulário para cadastro de posts. Este, por sua vez, será usado na última rota iniciada por este controlador, a *POST /posts*, que recupera os dados do form para inserir o post no banco de dados.

Ambas requerem que o usuário esteja logado, pois se o **username** retornado na chamada da função **sessionController.getUser(request)** for **null**, a ação é cancelada e a página de login é apresentada para o usuário. Finalmente, se o post for cadastrado com sucesso, o usuário será redirecionado para a página que exibe os detalhes do post.

Executando a aplicação

Com o objetivo de executar a aplicação é necessário implementar a classe **Main** que informamos no arquivo de configuração do Gradle. Para isso, crie uma classe conforme a apresentada na **Listagem 12**.

Listagem 11. Código da classe BlogController - Parte 2.

```
// Controle o form de cadastro de post
post"/posts", (request, response) -> {
    String title = request.queryParams("subject");
    String post = request.queryParams("body");
    String tags = request.queryParams("tags");
    String username = sessionController.getUser(request);

    if (username == null) {
        response.redirect("/login");
        return null;
    } else {
        if (title.equals("") || post.equals("")) {
            HashMap<String, String> root = new HashMap<String, String>();
            root.put("errors", "post must contain a title and blog entry.");
            root.put("subject", title);
            root.put("username", username);
            root.put("tags", tags);
            root.put("body", post);
            return new ModelAndView(root, POST_TEMPLATE);
        } else {
            ArrayList<String> tagsArray = extractTags(tags);
            post = post.replaceAll("\\r?\\n", "<p>");
            String permalink = postDAO.addPost(title, post, tagsArray, username);
            response.redirect("/posts/" + permalink);
            return null;
        }
    }
}

private static ArrayList<String> extractTags(String tags) {
    tags = tags.replaceAll("\\s","");
    String tagArray[] = tags.split(",");
    ArrayList<String> cleaned = new ArrayList<String>();

    for (String tag : tagArray) {
        if (!tag.equals("") && !cleaned.contains(tag)) {
            cleaned.add(tag);
        }
    }
    return cleaned;
}
```

Listagem 12. Código da classe principal da aplicação.

```
package br.com.devmedia.mongospark;

import static spark.SparkBase.*;
import br.com.devmedia.mongospark.controllers.BlogController;
import br.com.devmedia.mongospark.controllers.SessionController;
import br.com.devmedia.mongospark.controllers.UserController;

public class Main {

    public static void main(String[] args) {
        // O Spark vai rodar na porta 9090
        port(9090);

        // Inicializa as rotas
        SessionController.initRoutes();
        BlogController.initRoutes();
        UserController.initRoutes();
    }
}
```

Assim como toda classe **Main** Java, precisamos definir o método **main()** com sua assinatura padrão. Feito isso, o primeiro código que inserimos nele é uma chamada à função estática **port()** do Spark, onde declaramos que o servidor embutido deverá rodar na porta 9090. Em seguida, chamamos o método **initRoutes()** de cada controlador que desenvolvemos para que os mesmos sejam iniciados.

Como curiosidade, a declaração da porta através da função **port()** é opcional, e se não informada, o servidor executa a aplicação na porta padrão 4567. Sabendo disso, podemos notar que a única coisa necessária para desenvolver uma aplicação web com Spark é

inicializar as rotas, já que nenhuma outra configuração ou código *boilerplate* é obrigatório.

Com a última classe pronta, podemos rodar nosso projeto acessando a sua pasta raiz (*MongoSpark*) em um terminal e executando o comando:

gradle run

Caso tudo ocorra como esperado, deve ser possível visualizar no console a informação *[Thread-2] INFO spark.webserver.SparkServer - >> Listening on 0.0.0.0:9090*. Com essa confirmação, para acessar a aplicação, abra um navegador web e acesse a URL *http://localhost:9090/login*, para iniciar assim o processo de criação do primeiro usuário do blog (veja a **Figura 4**).

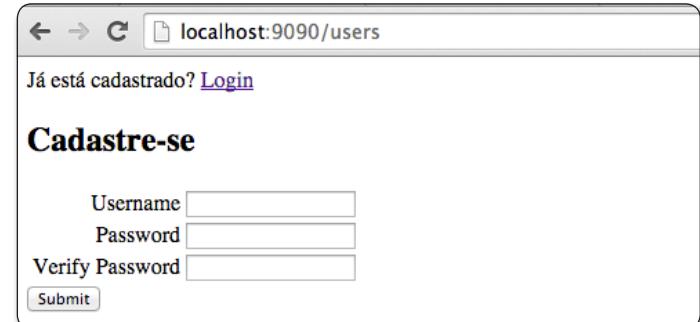


Figura 4. Tela de login do blog

Agora que chegamos ao fim do desenvolvimento do nosso micro blog, podemos observar a facilidade que é construir um projeto web com o framework Spark. Note que não tivemos a necessidade de configurar arquivos XML e nem instalar um

Criando um blog com MongoDB e Spark Framework

servidor de aplicação, para depois ainda ter que deployar um arquivo WAR.

Não é sempre que precisamos produzir sistemas de nível empresarial, com EJB e JPA, por exemplo, rodando em servidores de aplicação complexos de se configurar. Muitas vezes queremos apenas criar algo simples, onde não usariamos nem 50% das funcionalidades providas por outros frameworks focados em aplicações mais robustas, como o JavaServer Faces, as quais requerem muito mais configurações e códigos *boilerplate*.

O Spark é excepcional para um desenvolvimento ágil e produtivo de aplicações web. Entretanto, vale ressaltar que não estamos limitados à criação de micro blogs. Através do framework Spark podemos desenvolver qualquer tipo de aplicação web que não necessite de recursos que são somente providos pelos *Application Servers*.

Outro ponto a se destacar é que se tivermos a necessidade de executar uma aplicação desenvolvida com Spark em um servidor web, como o Tomcat, também é possível. Mas neste caso, seríamos obrigados a configurar um arquivo XML e todo o resto que faríamos com qualquer outra aplicação WAR que rode neste servidor.

A partir de agora o leitor está apto não só a desenvolver aplicações web com agilidade, produtividade e o menor esforço possível, mas também terá os conhecimentos necessários para adotar o MongoDB como solução de banco de dados.

Autor



Diego Travassos Balduini

*diego.balduini@gmail.com - http://blog.dbalduini.com/
MBA em Engenharia de Software pela FIAP, trabalha com Java
há mais de cinco anos e possui a certificação SCJP.*



Links:

Página do projeto Freemarker

<http://freemarker.org/>

Página do projeto Spark Framework

Página de projeto 3

Endereço para a especificação HTTP.

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

Endereço para download do MongoDB.

<http://www.mongodb.org/downloads>

Conhecimento faz diferença!

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**



DEVMEDIA

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia. Somos uma equipe composta de gente que entende e gosta do que faz, assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
[Conheça!](#)



Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

“Estudando Sistemas de Informação na Metodista eu me sinto desafiado a desenvolver minha capacidade dentro da sala de aula e no mercado de trabalho.”

Com os cursos da área de Exatas e Tecnologia da Metodista, o Sérgio tem contato com laboratórios e ferramentas de última geração. Além disso, desenvolve seu lado profissional por meio de Projetos de Ação Profissional, simulando na prática a resolução de problemas reais.

Sérgio
Aluno de Sistemas de Informação



CONHEÇA OS CURSOS DA ÁREA DE EXATAS E TECNOLOGIA DA METODISTA:

- Análise e Desenvolvimento de Sistemas (Presencial e EAD)
- Automação Industrial (Presencial)
- Engenharia Ambiental e Sanitária (Presencial)
- Engenharia de Produção (Presencial)
- Gestão da Tecnologia da Informação (Presencial)
- Sistemas de Informação (Presencial)



Transforme sua vida. Transforme a realidade.
processoseletivo.metodista.br



QUALIDADE | INOVAÇÃO | DESENVOLVIMENTO REGIONAL | INTERNACIONALIZAÇÃO

INOVAÇÃO DESDE 1938