



Alta produtividade com Spring Data, REST e MongoDB

Aprenda a criar uma API REST
sem implementar uma classe DAO

Relatórios com Hibernate e JasperReports

Construindo templates e integrando o
JSF à arquitetura do simulador

Edição 144 :: R\$ 14,90

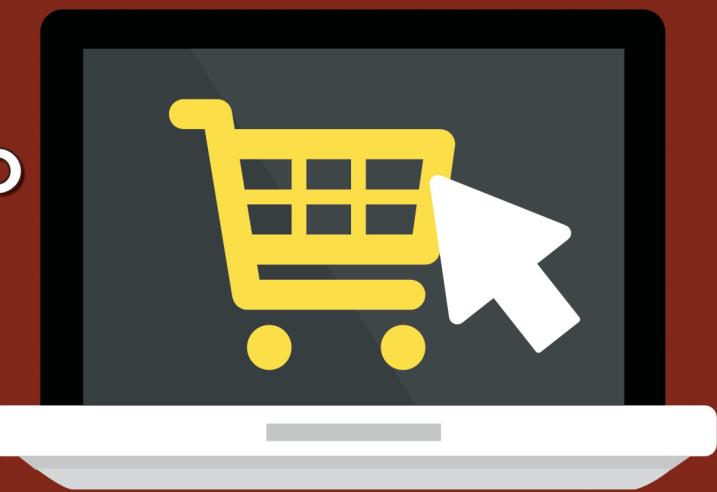
 DEVMEDIA

Introdução ao Spring Batch
Processando itens em lotes na prática



TURBINE SEU E-COMMERCE

Como criar
um mecanismo
de sugestão
de produtos



**Requisitos e Casos de Uso
versus User Story**

Comparando técnicas de
levantamento de requisitos

**A Gestão de Mudanças e
o Teste de Software**

Como a gestão de mudanças
pode auxiliar os testes

**Modelagem e implementação
com TDD e DDD**

Saiba evoluir o código-fonte do
negócio de forma clara e eficiente

MVP

R\$ 1.000.000,00
INVESTIDOS EM CONTEÚDO
NOS ÚLTIMOS 12 MESES.

APlique esse investimento
na sua carreira...

E MOSTRE AO MERCADO
QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

 **DEVMEDIA**



Edição 144 • 2015 • ISSN 1676-8361



MVP

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 (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araújo

Distribuição

FC Comercial e Distribuidora S.A.

Rua Teodoro da Silva, 907, Grajaú - RJ

CEP 20563-900, (21) 3879-7766 - (21) 2577-6362

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.

Java, o logotipo da xícara de café Java e todas as marcas e logotipos baseados em/ou referentes a Java são marcas comerciais ou marcas registradas da Sun Microsystems, Inc. nos Estados Unidos e em outros países.

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 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 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



Sumário

Conteúdo sobre Boas Práticas, Artigo no estilo Solução Completa

06 – Como criar uma API REST com Spring Data, REST e MongoDB

[Gabriel Novais Amorim e Kleber Andrade]

Conteúdo sobre Boas Práticas

14 – Modelagem e implementação com TDD e DDD

[Daniel Medeiros de Assis]

Conteúdo sobre Novidades, Artigo no estilo Solução Completa

28 – Apache Spark: Como criar um mecanismo de sugestão de produtos

[Luiz Henrique Zambom Santana e Eduardo Felipe Zambom Santana]

Artigo no estilo Curso

38 – Introdução ao Spring Batch – Parte 2

[Pedro E. Cunha Brigatto]

Artigo no estilo Curso

46 – Relatórios avançados com Hibernate, JasperReports e PrimeFaces – Parte 3

[Marcos Vinicios Turisco Dória]

Conteúdo sobre Boas Práticas

60 – Requisitos e Casos de Uso x User Story

[José Compadre Junior]

Conteúdo sobre Eng. de Software

68 – A Gestão de Mudanças e o Teste de Software

[Renata Eliza]



Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/javamagazine/feedback

CURSOS ONLINE



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

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>

Como criar uma API REST com Spring Data, REST e MongoDB

Aprenda como criar uma API REST em Java sem implementar uma classe DAO

A produtividade no desenvolvimento de software sempre foi motivo de preocupação e motivação para constantes pesquisas sobre como melhorar o rendimento por meio de ambientes tecnológicos, técnicas e frameworks de desenvolvimento. Devido a essas preocupações, a evolução dos frameworks e ferramentas de auxílio à programação de softwares tornaram-se constantes, tanto em relação à tecnologia, como a linguagem e a plataforma de desenvolvimento, quanto a técnicas e conceitos de programação, como orientação a objetos, orientação a serviços, etc.

O número de ferramentas cujo objetivo é aumentar a produtividade do desenvolvedor é enorme e este, por si só, pode melhorar ainda mais seu desempenho por meio da combinação de diversas técnicas e soluções disponíveis.

Com base nisso, o objetivo deste artigo é mostrar uma abordagem bastante produtiva ao lidar com o acesso a dados e configuração de ambiente. A ideia é criar uma API REST de acesso a dados que dispense a necessidade de programar classes DAO ou escrever comandos em SQL utilizando a abordagem de “convenção ao invés de configuração”, bastando definir uma interface para que as classes concretas e os comandos SQL sejam gerados em tempo de execução.

A princípio, ao se pensar em uma API REST para acesso a dados, a arquitetura que vem à cabeça seria a de uma aplicação que segue o padrão MVC, roda em um container configurado exclusivamente para a aplicação e conta com uma camada de acesso a dados que implementa o padrão DAO repleta de classes de mapeamento objeto-relacional e comandos SQL.

Fique por dentro

Formas de tornar o trabalho de desenvolvimento mais produtivo, seja no desenvolvimento de novas soluções ou na manutenção, sempre foram motivos de constantes pesquisas no mercado. Atualmente, a quantidade de ferramentas à disposição dos desenvolvedores é enorme, entretanto, nem sempre consegue-se fazer uso efetivo das mesmas, especialmente quando se trata de integrar diferentes soluções para o mesmo propósito.

Nesse artigo será mostrado como a utilização de diferentes ferramentas e conceitos em conjunto pode melhorar tanto o ambiente de desenvolvimento quanto a produtividade. Para isso, será mostrado como criar uma API REST para acesso e manipulação de dados que, com o uso do Spring Boot, torna o ambiente de execução mais rápido e facilmente replicado em diferentes máquinas para quantos desenvolvedores forem necessários. Além disso, o Spring Data será utilizado em conjunto com o MongoDB para facilitar a criação da API REST e proporcionar um ambiente livre de queries em SQL e de schemas de banco de dados, graças às características não relacionais do MongoDB.

Entretanto, é possível ter uma aplicação deste tipo de forma rápida e, o mais importante, implementada de uma maneira que torne a evolução muito produtiva, sem a necessidade de implementar classes concretas, comandos SQL, configurar servidores ou configurar URLs para o padrão REST.

Nesse artigo, serão apresentados uma arquitetura e um ambiente de desenvolvimento que possibilitam, juntos, um elevado nível de produtividade com a união do Spring Data, Spring Boot e MongoDB, cada qual sendo responsável por eliminar tarefas mais complexas e improdutivas do desenvolvimento de aplicações.

Spring Framework

O Spring se tornou, ao longo dos últimos anos, uma plataforma de desenvolvimento de aplicações Java que conta com uma série de ferramentas para as mais diversas situações. No caso deste artigo, a solução empregada faz uso de duas ferramentas provenientes desta plataforma: Spring Boot e Spring Data. Os próximos tópicos detalharão o que essas ferramentas fazem e como foram utilizadas na arquitetura da aplicação criada.

Spring Boot

O Spring Boot é uma solução que torna mais fácil a criação de aplicações stand-alone, deixando o maçante trabalho de configuração por conta da própria ferramenta. Graças às características desta ferramenta, é possível a criação de aplicativos Java que podem ser iniciados usando o comando `java -jar` ou mesmo aplicações web tradicionais com WAR. É importante frisar que ele não é uma ferramenta de geração automática de código. Pelo contrário, é, essencialmente, um plugin para o seu sistema de compilação (Maven, Gradle, entre outros).

O principal benefício do Spring Boot é a criação de recursos com base no que ele encontra no classpath. Se o `pom.xml` do projeto inclui dependências de um driver MySQL, por exemplo, então o Spring Boot irá criar uma unidade de persistência baseada em MySQL. Se for adicionada uma dependência web, então o projeto passa a seguir os padrões do Spring MVC. Se o projeto requer persistência, mas não for especificada qualquer informação sobre sua configuração, então o Spring Boot configura o Hibernate como um provedor JPA com um banco de dados HSQLDB por padrão.

Essas características vêm do conceito de convenção ao invés de configuração (*convention over configuration*), implementadas pelo Spring Boot, que facilita o desenvolvimento ao eliminar a necessidade de configurações extenuantes por intermédio de arquivos XML ou no próprio código. Na aplicação desenvolvida neste artigo, o Spring Boot é utilizado para prover um ambiente de desenvolvimento padronizado por meio da inicialização de um container web com todos os recursos necessários para a execução da mesma, como os módulos de acesso a dados e REST.

Spring Data

O Spring Data é um projeto do Spring cujo propósito é facilitar o acesso a dados, sejam bancos de dados não-relacionais, serviços de dados baseados em nuvem ou bancos de dados relacionais. Dentro do projeto Spring Data há uma série de subprojetos específicos, tais como Spring Data JPA, Spring Data JDBC Extension, Spring Data REST, entre outros. Todos com o objetivo de facilitar a manipulação de dados sob uma plataforma ou tecnologia, viabilizando uma interface de alto nível entre a aplicação e a fonte de dados.

O Spring Data MongoDB é um dos subprojetos do Spring Data e fornece integração com o MongoDB ao prover uma série de facilidades, entre elas, a criação de queries em tempo de execução, um dos pilares da aplicação desenvolvida neste artigo. Graças ao Spring Data MongoDB é possível tirar proveito de um banco de dados não relacional sem precisar conhecer profundamente

detalhes de como realizar as operações sobre o mesmo. O Spring Data é utilizado nessa aplicação para integrar o MongoDB sem a necessidade de se conhecer a sintaxe da DML implementada pelo mesmo, tornando mais simples e rápido o desenvolvimento, além de possibilitar o mapeamento direto entre operações REST com métodos de acesso a dados.

MongoDB

O MongoDB é um banco de dados orientado a documentos que provê, entre outras características, a flexibilidade do *schema*. Essa particularidade torna muito simples e fácil a manipulação de dados, pois elimina a necessidade de se ter um *schema*, o que possibilita a inserção sem precisar declarar e determinar campos e tabelas previamente. Em aplicações onde as entidades seguem os princípios da orientação a objetos, essa característica facilita o mapeamento de documentos para objetos e vice-versa.

A aplicação implementada neste artigo faz uso do MongoDB para armazenar e recuperar informações, entretanto, não é necessário nenhum conhecimento a respeito de como realizar a manipulação de dados no MongoDB, pois o Spring Data abstrai essa etapa e se responsabiliza por realizar as operações e servir como uma interface de acesso a dados para o restante da aplicação.

Os próximos tópicos mostram como criar uma API REST que dispensa a definição do modelo de dados e a criação de queries, solução esta a ser implementada em um ambiente autoconfigurável fácil de replicar entre os desenvolvedores.

Arquitetura para produtividade

A utilização do Spring em conjunto com o MongoDB possibilita a criação de uma arquitetura robusta e flexível graças às



Como criar uma API REST com Spring Data, REST e MongoDB

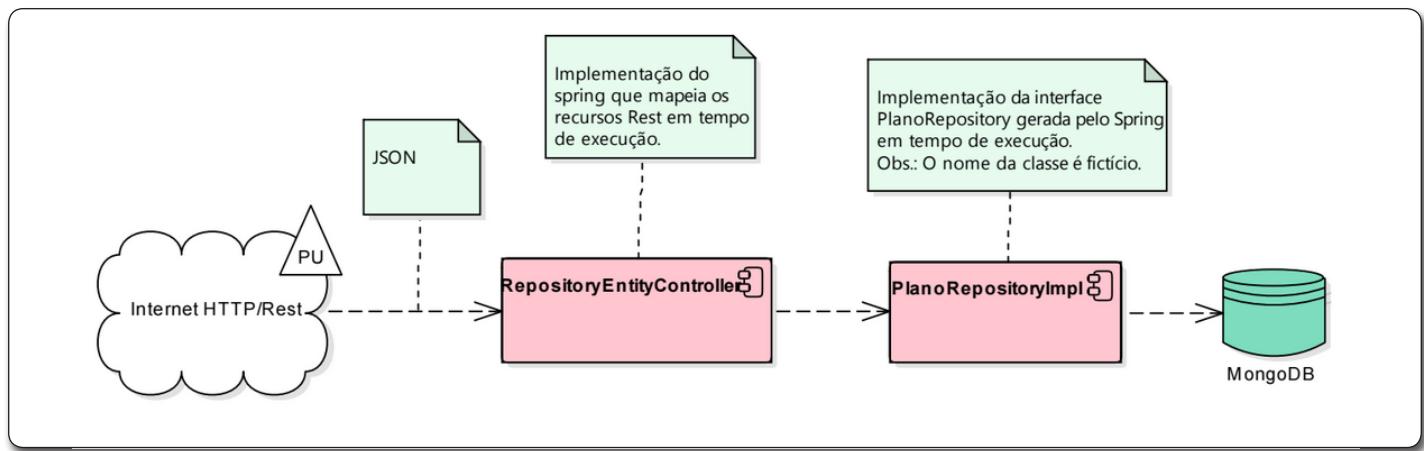


Figura 1. Arquitetura da solução com Spring Data e MongoDB

susas características que, em conjunto, contribui muito com o desenvolvimento para torná-lo mais simples. Do Spring, tem-se o Spring Data, que facilita a exposição de uma API REST baseada em métodos declarados em uma interface, e do MongoDB tem-se a vantagem da adoção de uma abordagem *schemaless*, podendo-se alterar a aplicação a qualquer momento sem a necessidade de alterar o banco de dados.

A seguir será mostrada a arquitetura da aplicação de exemplo e como a mesma foi implementada utilizando-se as ferramentas comentadas anteriormente.

Aplicação de exemplo

A aplicação para ilustrar os conceitos e tecnologias deste artigo é um cadastro de planos de telefonia. Basicamente, será possível a inclusão, alteração, remoção e consulta de planos, e todas essas operações serão expostas em uma API REST de forma que todos os dados sejam persistidos em uma base de dados não relacional.

A Figura 1 mostra um overview e os principais componentes da arquitetura dessa solução. Como pode ser notado, toda a troca de informações entre os clientes e a API REST é feita com JSON. Essas requisições (GET e POST) são mapeadas pelo Spring Data (por meio da classe interna do Spring **RepositoryEntityController**) e são direcionadas para os métodos específicos da implementação da interface **PlanoRepository**, que é uma extensão da interface **MongoRepository**, fornecida pelo Spring Data e que determina as operações padrão de acesso a dados.

A interface **PlanoRepository** define as operações que deverão ser expostas, sendo que cada método declarado na interface será mapeado como um recurso REST, além de se tornar uma operação de manipulação de dados no MongoDB. Essa interface, que define os métodos de acesso a dados, receberá, em tempo de execução, uma implementação provida pelo Spring Data. Dessa forma é possível adicionar novas operações de forma simples e rápida, sem o conhecimento de linguagens como SQL ou outras utilizadas por bancos não relacionais. Essa implementação é a responsável pelo acesso ao MongoDB.

Listagem 1. Classe responsável por iniciar uma aplicação Spring a partir de um método main().

```
package br.com.minhaoperadora.service.config;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.data.mongodb.repository.config.EnableMongoRepositories;
import org.springframework.data.rest.webmvc.config.RepositoryRestMvcConfiguration;

@Configuration
@EnableMongoRepositories
@Import(RepositoryRestMvcConfiguration.class)
@EnableAutoConfiguration
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

A seguir será mostrado em mais detalhes como configurar cada um dos componentes arquiteturais dessa aplicação.

Configurando a aplicação com o Spring Boot

A aplicação implementada neste artigo faz uso do Spring Boot como meio de fornecer um ambiente padrão de desenvolvimento, no qual as convenções mais comuns de configuração são utilizadas. Essas convenções têm por objetivo aumentar a produtividade e manter a consistência entre ambientes quando muitos desenvolvedores estão trabalhando no mesmo projeto.

A Listagem 1 mostra a classe principal, que é responsável por iniciar o ambiente de execução e a aplicação com base em suas anotações diretas (aqueles anotadas diretamente na classe **Application**) e indiretas (anotadas em classes utilizadas pela classe **Application**). Os componentes do Spring Boot utilizados nessa classe de inicialização são a classe **org.springframework**

.boot.SpringApplication e a anotação @org.springframework.boot.autoconfigure.EnableAutoConfiguration.

A classe **SpringApplication** é responsável por iniciar uma aplicação Spring, que será o background da API respondendo a todas as requisições realizadas sobre a mesma. A aplicação é iniciada a partir do método **main()**, que, por sua vez, invoca **SpringApplication.run()**. Com a invocação desse método, **Application** será carregada e suas anotações serão utilizadas para configurar uma aplicação Spring por intermédio da criação de um **ApplicationContext**, que é uma interface responsável por fornecer as configurações de um sistema de forma centralizada.

Ainda sobre como configurar o background da API, comentado anteriormente, a anotação **@EnableAutoConfiguration** fornece características padrão para todos os beans que o sistema possa precisar. Isso sem a necessidade de informar, por intermédio de arquivos XML ou anotações extras, propriedades para o funcionamento do sistema. Essa anotação ainda configura a aplicação Spring com base nas dependências adicionadas ao projeto por meio do arquivo *pom.xml*.

Como exemplo da autoconfiguração, consideremos as dependências do projeto apresentadas na **Listagem 2**. A dependência **spring-boot-starter-web** acrescenta o Tomcat e o Spring MVC à aplicação. Neste cenário, o mecanismo de autoconfiguração irá assumir que se trata de uma aplicação web e vai configurar para a mesma, além do Spring MVC, um servidor de aplicação (que por padrão é o Tomcat) que será iniciado com a execução do método **SpringApplication.run()** da **Listagem 1**. Há a possibilidade de autoconfiguração também pela inclusão de arquivos JAR no classpath como, por exemplo, se o HSQLDB estiver no classpath e não houver especificações para o mesmo, então o Spring Boot irá configurar um banco de dados em memória e prover, por meio das classes específicas para bancos de dados, uma interface para acesso e manipulação dos dados no banco.

Listagem 2. Dependências do projeto no arquivo pom.xml.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-rest-webmvc</artifactId>
  </dependency>
</dependencies>
```

Com o Spring Boot configurado de acordo com os passos anteriores, o background da API passa a ter o seu start a partir da classe **Application**. Ao executar essa classe, o Tomcat será iniciado e a aplicação implantada no mesmo. Além disso, o Spring MVC também será iniciado, fazendo o mapeamento de todas as URLs

da API REST. Nesse momento tem-se a estrutura de uma aplicação Spring MVC sendo executada no Tomcat, entretanto, como nenhum código foi implementado ainda, esta aplicação não tem nenhuma utilidade prática. Sendo assim, a seguir será mostrado o que é necessário para tornar essa aplicação uma API REST e conectar ao banco de dados.

Configurando a API REST

Note que a classe **Application**, da **Listagem 1**, também foi anotada com **@org.springframework.context.annotation.Configuration**. Esta anotação indica que a classe possui um ou mais métodos que podem ser executados pelo container do Spring com o objetivo de gerar definições para beans em tempo de execução, ou seja, indica que a classe anotada é uma classe de configuração.

Além disso, é possível compor a configuração da classe atual com dados em outras classes. Isso é feito por meio da anotação **@org.springframework.context.annotation.Import**, que recebe como parâmetro uma classe de configuração para ser adicionada durante a inicialização.

Nesse caso, a classe recebida por parâmetro foi **org.springframework.data.rest.webmvc.config.RepositoryRestMvcConfiguration**. Esta classe possui os dados para o Spring Data REST e para o suporte ao MongoDB. Ela possibilita a operações CRUD serem expostas usando a semântica HTTP REST, que são baseadas nos métodos HTTP como GET, POST e UPDATE.

A seguir será mostrado o que é necessário para ter o MongoDB integrado ao Spring Data REST, de forma que todas as operações do CRUD sejam expostas como uma API REST.



Configurando o uso do MongoDB com Spring Data

A configuração do MongoDB para integração com o Spring Data se dá através da classe **Application**, apresentada na [Listagem 1](#). Como pode ser notado, essa classe possui a anotação `@org.springframework.data.mongodb.repository.config.EnableMongoRepositories`, recurso com o propósito de ativar os repositórios do MongoDB para fazerem com que, na inicialização da aplicação, sejam buscadas quaisquer interfaces implementadas pelo desenvolvedor que estendam a interface `org.springframework.data.mongodb.repository.MongoRepository`, utilizada para definir métodos padrão para inserção e busca. Assim, fica a cargo do Spring Data criar os DAOs em tempo de execução de acordo com a interface definidora das operações implementadas pelo desenvolvedor.

A [Listagem 3](#) mostra a interface `PlanoRepository`, que estende `MongoRepository` e define métodos de busca com diferentes parâmetros. Quando a aplicação for executada, serão criados DAOs que implementam `PlanoRepository` e contenham, além dos métodos padrão especificados na interface `MongoRepository`, a implementação para os métodos definidos. O tipo de query a ser gerada dependerá do nome do método especificado na interface e do padrão de nomenclatura especificados pelo Spring Data. Note que os métodos descritos na interface da [Listagem 3](#) começam com “`findBy`”, significando que este método deve ser implementado como uma busca.

Nota

Veja o tópico “Construindo queries com Spring Data” para mais detalhes sobre a construção de queries com o Spring Data.

Listagem 3. Interface PlanoRepository. Define os métodos que serão expostos na API REST.

```
package br.com.minhaoperadora.service.config;

import java.math.BigDecimal;
import java.util.List;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(collectionResourceRel = "plano", path = "plano")
public interface PlanoRepository extends MongoRepository<Plano, String> {

    List<Plano> findByTipo(@Param("tipo") TipoPlano tipo);
    List<Plano> findByValorLessThan(@Param("valor") BigDecimal valor);
}
```

Além disso, cada operação da interface `PlanoRepository` é mapeada como um recurso para ser acessada por meio da API REST. Esse mapeamento é feito pelo Spring Data a partir do conceito de convenção ao invés de configuração. Sendo assim, não são necessárias configurações extras. O nome da operação definido pelo desenvolvedor na interface será a URL gerada pelo Spring para que o método seja invocado em uma requisição HTTP.

Para utilizar os recursos do Spring Data em conjunto com o MongoDB é necessário implementar uma interface que estenda `MongoRepository`. Essa interface possui métodos comuns de acesso a dados, é genérica e recebe como tipos genéricos o objeto de domínio que o repositório deverá gerenciar e o tipo do identificador da entidade em questão. Na [Listagem 3](#), o tipo do domínio é uma classe que especifica objetos do tipo `Plano` e o tipo do identificador é uma `String`. A [Listagem 4](#) mostra a definição do objeto de domínio. A classe é um POJO e precisa apenas especificar os getters e setters para todos os atributos e anotar o identificador com a anotação `@org.springframework.data.annotation.Id`.

As URLs que serão utilizadas para invocar os métodos da interface `PlanoRepository` por meio de REST são especificadas com o uso da anotação `@org.springframework.data.rest.core.annotation.RepositoryRestResource`. Essa anotação recebe dois parâmetros: `collectionResourceRel`, que especifica um nome de recurso utilizado para que o Spring Data possa criar os links de acesso às operações expostas; e `path`, que indica o caminho a partir da raiz onde o recurso será exposto. Dessa forma, a busca por todos os planos pode ser realizada, por exemplo, por meio da URL `http://localhost:8080/plano`.

Além da busca por todos os planos, a interface `PlanoRepository` define ainda duas buscas com parâmetros, uma por tipo e outra para planos cujo valor seja menor que o informado. Esses parâmetros devem ser configurados pelo uso da anotação `@org.springframework.data.repository.query.Param`, que identifica o nome do parâmetro na requisição HTTP para fazer o mapeamento para o método.



Os passos descritos até o momento configuram uma aplicação e um ambiente completo de desenvolvimento fazendo uso de recursos do Spring integrados ao MongoDB, tornando assim possível a rápida inicialização de um container web contendo uma aplicação pronta para receber requisições no padrão REST (por intermédio do Spring Data). Além disso, a produtividade no desenvolvimento aumenta bastante devido a não ser mais necessária a implementação de DAOs para cada objeto de domínio nem a alteração do banco de dados a cada mudança do modelo de dados, pois o Spring Data gera os DAOs em tempo de execução enquanto o MongoDB fornece um meio de persistência flexível.

Construindo queries com Spring Data

O Spring Data tem uma característica que aumenta bastante a produtividade do desenvolvedor: a geração de queries de acordo com o nome dos métodos. Dessa forma, basta criar uma interface e definir os nomes dos métodos de acordo com o padrão do Spring Data (baseado na sintaxe e operações da SQL) para que o mesmo gere o DAO que implementa cada método definido na interface, além da query que será executada no banco de dados. A **Listagem 3** é um exemplo de interface para um DAO que tem a implementação gerada em tempo de execução pelo Spring Data.

As queries geradas pelo Spring Data baseiam-se no nome do método, que deve ser construído como uma expressão. A construção de queries implementada pelo Spring Data procura pelos prefixos `findBy`, `readBy`, `queryBy`, `countBy` e `getBy` em um método e, então, começa a analisar o restante do nome, buscando por outras palavras-chaves que podem ser utilizadas para a construção das expressões, tais como `Or`, `And` ou `Distinct`.

O primeiro `By` encontrado na expressão é um delimitador e indica o início de um critério de busca, como mostram os exemplos a seguir:

```
List<Plano> findByNome(String nome);
List<Plano> findByTipo(String tipo);
List<Plano> findByDescricao(String descricao);
```

Note que, após a palavra-chave `By`, tem-se definido um atributo da entidade que será utilizado como modelo de dados para a busca. Esse atributo é o critério de busca. Deste modo, as queries geradas para os exemplos supracitados seriam, respectivamente:

```
SELECT * FROM Plano WHERE Nome = 'Nome';
SELECT * FROM Plano WHERE Tipo = 'Tipo';
SELECT * FROM Plano WHERE Descricao = 'Descricao';
```

Listagem 4. Classe Plano como objeto de domínio no repositório do MongoDB.

```
package br.com.minhaoperadora.service.config;

import java.math.BigDecimal;

import org.springframework.data.annotation.Id;

public class Plano {

    @Id
    private String id;
    private String nome;
    private String descricao;
    private String vantagens;
    private BigDecimal valor;
    private TipoPlano tipo;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public String getVantagens() {
        return vantagens;
    }

    public void setVantagens(String vantagens) {
        this.vantagens = vantagens;
    }

    public BigDecimal getValor() {
        return valor;
    }

    public void setValor(BigDecimal valor) {
        this.valor = valor;
    }

    public TipoPlano getTipo() {
        return tipo;
    }

    public void setTipo(TipoPlano tipo) {
        this.tipo = tipo;
    }
}
```

Os parâmetros do método se tornam o critério de busca e são mapeados para a query gerada de acordo com o próprio nome do atributo. Além disso, cada um desses métodos retorna uma lista da entidade pesquisada. Esse é o padrão de retorno implementado pelo Spring Data.

Vale ressaltar ainda que o acesso a dados e suas respectivas expressões podem tomar formas mais complexas, a partir da combinação de diferentes operadores como *Between*, *LessThan*, *GreaterThan* e *Like*, que fazem parte da linguagem SQL. Seguem alguns exemplos de expressões mais complexas:

```
List<Plano> findByNomeAndTipo(String nome, TipoPlano tipo);  
List<Plano> findByNomeIgnoreCase(String nome);  
List<Plano> findByNomeAndTipoAllIgnoreCase(String nome, TipoPlano tipo);  
List<Plano> findByNomeOrderByDescricaoAsc(String nome);  
List<Plano> findByNomeOrderByDescricaoDesc(String nome);
```

Observe como o nome do método se torna intuitivo e como é fácil imaginar como será a query gerada em SQL. Além desses recursos, esse mecanismo possibilita a realização de qualquer operação que seria realizada por intermédio de uma query nativa em SQL e até alguns recursos avançados como paginação. Para entendimento completo sobre esse mecanismo de geração de queries por meio de métodos definidos em interface, provido pelo Spring Data, veja a seção **Links** para a documentação oficial.

Usando a aplicação construída

O desenvolvimento da solução proposta na **Figura 1** envolveu basicamente três passos: a configuração do Spring Boot, a configuração do Spring Data implementando uma interface de métodos de queries junto com os requisitos necessários para expor os métodos como API REST e a integração com o MongoDB.

Essa parte do artigo se concentra em mostrar como a aplicação e o ambiente de desenvolvimento funcionam após todos os passos necessários para a implementação e configuração dos mesmos.

Iniciando a aplicação

O boot da aplicação é facilitado com o uso do Spring Boot. A classe **Application**, apresentada na **Listagem 1**, é a responsável por iniciar a API REST em conjunto com o Spring e MongoDB em um container web, que neste caso, como não foi alterado o container padrão, será o Tomcat. Durante a inicialização, o Spring Boot varre todo o classpath em busca de elementos de configuração; no caso da API REST implementada neste artigo, não há elementos a serem lidos do classpath. Ao executar a classe **Application**, seja diretamente a partir da IDE ou da linha de comando, o Tomcat será iniciado e a aplicação será implantada no container. Logo após, as URLs REST de acesso a dados serão mapeadas de acordo com os métodos definidos na interface **PlanoRepository** e os repositórios MongoDB serão ativados.

Consultando, alterando e removendo dados

Após a inicialização, é possível verificar o funcionamento da aplicação acessando-a de um navegador pela URL <http://localhost:8080/plano>. Quando uma requisição GET é enviada para essa URL, a aplicação realiza uma busca por todos os planos e retorna o resultado no formato JSON, como o demonstrado na **Listagem 5**.

A busca por registros específicos pode ser feita pelo envio de uma requisição GET à mesma URL, <http://localhost:8080/plano/>, mas adicionando o identificador do registro no final: <http://localhost:8080/plano/{id}/>. Note como o padrão REST é implementado perfeitamente e como a primeira requisição GET enviada retorna todos os planos. Ao realizar uma segunda requisição GET sobre um identificador, a aplicação retorna apenas o registro daquele identificador.

Além das operações de busca, tem-se ainda as operações de inserção, atualização e remoção de registros. Para fazer uma inserção, deve-se enviar uma requisição POST com os dados a serem inseridos em formato JSON para a URL <http://localhost:8080/plano>. O formato JSON a ser enviado pelo cliente deve ser enviado junto a requisição conforme o exemplo:

```
{"nome": "Plano Controle 500MB", "tipo": "CONTROLE", "descricao": "Plano Controle",  
"vantagens": "desc vantagens", "valor": "50.0"}
```

A operação de update é parecida com a operação de inserção. Para fazer um update, deve-se enviar uma requisição POST com os dados a serem atualizados no formato JSON para a



URL <http://localhost:8080/plano/{identificador}>. O identificador especifica o registro que será atualizado e o formato dos dados em JSON é igual ao utilizado na operação de inserção.

Listagem 5. Resultado em JSON de uma consulta por todos os planos.

```
{  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/plano?page,size,sort",  
      "templated": true  
    },  
    "search": {  
      "href": "http://localhost:8080/plano/search"  
    }  
  },  
  "_embedded": {  
    "plano": [  
      {  
        "id": "5596cca7dc1a45f2a997d29",  
        "nome": "Plano Pos 1GB dados",  
        "descricao": "Plano Pos",  
        "vantagens": "vantagens plano pos",  
        "valor": 30.0,  
        "tipo": "POS",  
        "_links": {  
          "self": {  
            "href": "http://localhost:8080/plano/5596cca7dc1a45f2a997d29"  
          }  
        }  
      },  
      {  
        "id": "5596cf3cdcb1a45f2a997d2a",  
        "nome": "Plano Controle 500MB",  
        "descricao": "Plano Controle",  
        "vantagens": "desc vantagens",  
        "valor": 50.0,  
        "tipo": "CONTROLE",  
        "_links": {  
          "self": {  
            "href": "http://localhost:8080/plano/5596cf3cdcb1a45f2a997d2a"  
          }  
        }  
      }  
    ]  
  },  
  "page": {  
    "size": 20,  
    "totalElements": 2,  
    "totalPages": 1,  
    "number": 0  
  }  
}
```

Por fim, para remover um registro basta realizar uma requisição DELETE para a URL <http://localhost:8080/plano/{identificador}>. O identificador especifica o registro que será removido da base de dados.

Além da abordagem apresentada neste artigo para a criação de uma API REST de alta produtividade por meio de métodos de busca do Spring Data e das características não relacionais do MongoDB, há muito a explorar sobre essa alternativa apresentada. O Spring Boot possibilita ainda configurar a inicialização de diversos aspectos e ferramentas necessárias para o funcionamento

das aplicações, como o acesso a dados, segurança e integração, proporcionando muitas alternativas rápidas para diversos aspectos do desenvolvimento de um software. Enquanto isso, o Spring Data fornece integração com diferentes bancos de dados e provê uma interface que possibilita a criação de métodos de busca mais complexos, envolvendo diferentes tipos de mapeamento, permitindo o desenvolvimento de aplicações onde a lógica de acesso a dados é complexa sem o uso de SQL diretamente.

Autor



Gabriel Novaís Amorim

novaís.amorim@gmail.com – blog.gabrielamorim.com

Engenheiro de software, trabalha com desenvolvimento de sistemas há seis anos e atualmente tem implementado soluções SOA na plataforma Java. Possui as certificações OCJP, OCEJWCD, OCEEJBD, OCEJWSD, IBM-OOAD, IBM-RUP, CompTIA Cloud Essentials e SOACP. Seus interesses incluem arquitetura de software e metodologias ágeis. Tecnólogo em Análise e Desenvolvimento de Sistemas (Unifieo) e especialista (MBA) em Engenharia de Software (FIAP). Escreve também em: blog.gabrielamorim.com



Autor



Kleber Andrade

kleber.andrade.reis@gmail.com

Analista de Sistemas, trabalha com desenvolvimento de sistemas há oito anos. Atualmente trabalha na implementação de soluções na plataforma Java para telecomunicações. Possui as certificações SCJP e SCWCD. Seus interesses incluem arquitetura de software, metodologias ágeis e SOA. Graduação em Ciência da Computação e Técnico em Automação. Trabalhou também com automação de sistemas: softwares SCADA e PLCs.



Links:

Documentação do Spring Boot

<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>

Como utilizar o Spring Data com o MongoDB.

<http://docs.spring.io/spring-data/mongodb/docs/current/reference/html/>

Spring Data: Repositories and query methods.

<http://docs.spring.io/spring-data/data-commons/docs/1.10.1.RELEASE/reference/html/#repositories.query-methods.details>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Modelagem e implementação com TDD e DDD

Evoluindo o código-fonte do negócio de forma clara e eficiente

Segundo o livro *The Passionate Programmer* (Chad Fowler, 2009), uma das características mais importantes de um programador é sua capacidade de entender o negócio da empresa na qual trabalha (vide **BOX 1**). Muitos de nós atuam profissionalmente no desenvolvimento de software em empresas que não têm o cerne do seu negócio em TI (como, por exemplo, no setor bancário, de seguros, de petróleo, etc.).

Não se trata, contudo, em ser um especialista em negócio (afinal, não é nossa profissão – como programadores, nosso trabalho é criar software), mas é importante que se conheçam os conceitos relevantes ao negócio e como os mesmos se relacionam, de forma a entender as necessidades dos clientes, abrindo caminho para a criação de um produto de software adequado. Apenas ser especialista em tecnologia, num contexto como este, não basta: o programador deve entender o que o cliente realmente precisa e saber transportar esta necessidade para software.

A questão aqui é comunicação e entendimento. Ao conhecer o negócio da empresa, o programador está mais apto a entender, criticar e questionar requisitos e, posteriormente, a implementar software de valor. Contudo, sabe-se que a falha de comunicação é um problema comum em desenvolvimento de software: o maior exemplo disto talvez seja aquele onde um mesmo termo é empregado com múltiplos significados (o que leva o dito por uma pessoa a ser entendido diferentemente por outra).

Desconsiderando o problema de comunicação por um momento, ainda resta a questão técnica de escrever software com base em conceitos do negócio.

Fique por dentro

Este artigo é útil por apresentar o TDD como uma técnica de design (estendendo seu enfoque em relação à abordagem tradicional de testes) e como ele pode ser usado em conjunto com os padrões do DDD de forma a construir um modelo de negócios claro, descritivo, extensível e testável. Neste contexto, será considerado um problema do mundo real – a modelagem de um domínio para um software pessoal de gerenciamento financeiro – com o intuito de demonstrar a aplicação da técnica proposta.

BOX 1. O programador apaixonado e o mercado de trabalho

O livro *The Passionate Programmer*, escrito em 2009 por Chad Fowler, apresenta um conjunto de ótimos conselhos sobre como trabalhar com software em empresas. Dentre dicas sobre gerenciamento da própria carreira até sugestões controversas, mas justificáveis (como a importância de ser o pior programador dentre os melhores), o autor oferece um panorama realista sobre o dia a dia daqueles que trabalham com programação de software – e como manter-se apaixonado pela profissão, apesar de tudo. Leitura recomendada.

Existem tantas formas de se fazer isto quanto existem programadores – e nem todas as formas empregadas resultam em código-fonte legível, claro, extensível e testável, e que representa claramente o negócio da empresa. No dia a dia, observa-se que dúvidas sobre as regras de negócio são repassadas para o código-fonte, o que resultará em problemas no futuro. Por mais que um código-fonte esteja bem escrito, se os conceitos assumidos divergirem dos conceitos do negócio, problemas podem ocorrer por conta de dificuldades de comunicação e entendimento no que se espera que deva ser feito.

Neste contexto, as técnicas de modelagem e implementação que envolvem DDD e TDD podem ser empregadas em conjunto para lidar com os problemas de entendimento de negócio e representação adequada do mesmo em código-fonte Java com qualidade.

Conceituando o TDD enquanto ferramenta de design

A técnica de TDD (*Test-Driven Development*) é bastante conhecida atualmente pela sua característica controversa de escrever testes antes do código-fonte. Ao se considerar o percentual de cobertura de código-fonte com testes como uma métrica de qualidade, TDD apresenta-se como uma forma eficaz de se programar.

Contudo, TDD também deve ser relacionado com design de software. O criador da técnica, em seu livro *Test-Driven Development By Example* (Kent Beck, 2003), aponta que as duas regras básicas do TDD (escrever código novo apenas quando o teste automatizado falhar e eliminar duplicação) levam a duas implicações em design: a) o design surge de forma *orgânica*, com código em execução provendo feedback entre decisões; e b) o design deve consistir em componentes com baixo acoplamento e alta coesão, para permitir que seja fácil testar.

Um design *orgânico* é aquele que surge conforme a aplicação evolui. Este design é chamado de *Evolucionário* por Martin Fowler, e é caracterizado por ser parte do processo de programação, crescendo enquanto o software é implementado. Ainda segundo Fowler, este design é um desastre em sua forma comum, pois o que resulta é um conjunto de decisões *ad-hoc* que tornam o código difícil de manter. Uma forma de fazer o design evolucionário funcionar é pela utilização de práticas que imponham o design, tais como testes, refatoração e integração contínua – práticas empregadas no método XP.

O TDD é uma técnica a ser utilizada com o design *Evolucionário*, em contraposição ao design *Planejado*. Num design *Evolucionário* em sua forma comum, como já dito anteriormente, pode-se entender que não existe design de fato (todas as decisões são *ad-hoc* em termos de codificação e não existe uma preocupação com design como um todo), o que leva a um aumento da entropia de software, dificultando evoluções e manutenções. Já um design *Planejado* é o inverso: sugere que deve haver um planejamento prévio e de alto nível, que considere todas as necessidades de mudanças possíveis, antes mesmo que o software seja desenvolvido.

Ambas as abordagens (planejada e evolucionária) falham em seus objetivos enquanto conceitos isolados em si, quando é considerada a questão da mudança em software. Se a entropia de software causada pelo design *Evolucionário* dificulta mudanças, também o design *Planejado* o faz, no sentido de que adota uma postura na qual mudanças não são encorajadas (todo impacto em design deveria ter sido pensado no início do projeto, antes da implementação), ou adota uma postura de flexibilidade total a todo tipo de mudanças, o que torna a fase de design extremamente cara (pois é preciso antecipar todos os problemas que ainda não existem), e pode tornar a implementação pouco clara e eficiente (pois design genérico demais para atender a problemas de forma muito flexível pode demandar implementação pouco performática).

e difícil de entender e manter). A proposta do método XP é a de utilizar um design *Evolucionário* de forma controlada, e o TDD é uma das técnicas empregadas para controlar a entropia.

Segundo Kent Beck, “design existe para permitir que mudanças em software sejam realizadas facilmente por um longo tempo”. O software que faz uso de TDD possui componentes criados com alta coesão e baixo acoplamento, pois é resultado do processo que gera código-fonte testável. Estas características da orientação a objetos que emergem durante a prática de TDD propiciam a facilidade em mudanças sugeridas por Kent. Contudo, caso o TDD seja abandonado, o design pode deteriorar com o tempo, aumentando a entropia e tornando o software mais difícil de manter e corrigir.

Não é verdadeiro afirmar que apenas TDD garante qualidade em design, mas pode-se considerar que a prática do TDD, enquanto baseada na criação de testes automatizados, força a existência de um design de qualidade. Este design, contudo, é puramente técnico, voltado às práticas de orientação a objeto visando flexibilidade, mas sem relação direta com o negócio da empresa – e é neste ínterim que o relacionamento com uma técnica de modelagem de negócio faz-se relevante.

O DDD e a modelagem do negócio

O termo DDD (*Domain-Driven Design*) foi cunhado por Eric Evans, em seu livro de 2004. A obra apresenta um conjunto de práticas que visam tornar o entendimento do negócio explícito. As práticas sugeridas no livro não são novas. Contudo, o mérito de Evans está em agrupá-las e organizá-las de forma a permitir que o difícil trabalho de mapear um domínio seja possível de realização.

De acordo com Evans, o foco central do desenvolvimento de software deve ser a modelagem de domínio, e não a tecnologia. Isto pode parecer um tanto quanto radical à primeira vista (em especial aos programadores que aprenderam a desenvolver utilizando padrões apenas voltados a aspectos técnicos). Mas a ideia passa a fazer sentido ao considerarmos o cenário citado



no início deste artigo (onde é comum a atuação profissional em desenvolvimento de software em empresas que não têm o cerne do seu negócio em TI): nos cenários onde o sistema de software serve para dar suporte ao negócio, é extremamente importante que o negócio seja explícito e claro aos envolvidos. DDD oferece um conjunto de conceitos e padrões neste sentido.

O primeiro destes padrões de grande importância é a *Linguagem Onipresente*. De modo simples, pode-se dizer que ela está relacionada às dificuldades geradas com o uso de algum termo que possua muitos sentidos/significados entre as pessoas envolvidas no projeto. Isto gera problemas no entendimento dos conceitos e provavelmente se propagará para a implementação, resultando em código confuso. O modelo de domínio, ao representar conhecimento sem ambiguidade e com conceitos explícitos, permite que haja uma linguagem eficaz e comum para todo o projeto. O conceito de Linguagem Onipresente sugere que a linguagem adotada deve ser única para todos os envolvidos no projeto: todos devem falar a mesma língua. Os mesmos termos devem ser utilizados pelo especialista do negócio e pelos desenvolvedores implementando o software, e o mesmo significado deve ser entendido. Isto não significa dizer que os desenvolvedores precisam ser especialistas no negócio, mas sim que precisam ter o conhecimento daquilo que estão implementando, em termos do nome certo e do que significa. A Linguagem Onipresente é um conceito simples e poderoso: apenas exige que saibamos dar o nome correto aos conceitos, que trabalhemos na explicitação de conceitos e que não permitamos ambiguidades.

Para que a implementação do software esteja em sintonia com o modelo, o DDD apresenta um conjunto de padrões. A seguir são apresentados os mais comuns:

- **Entidade:** É aquela classe que possui uma identidade única. Essa identidade pode ser o CPF de um cliente, por exemplo. Geralmente, as responsabilidades da entidade são realizadas através da chamada a outros objetos, como objetos de valor;

- **Objeto de Valor:** Representa um conceito, mas sem identidade única definida. É um objeto que pode ser empregado em diversas situações sem que uma instância em específico seja relevante. Por exemplo, o número 5 pode ser representado como um objeto de valor, de forma que qualquer instância do número 5 (e não necessariamente uma instância específica) pode ser usada em vários contextos diferentes. Um bom design procura colocar regras em métodos de objetos de valor, fazendo com que as entidades utilizem tais objetos para atender suas responsabilidades (em vez de concentrar as regras nas próprias entidades). Mas ao considerar a questão de separar código em muitos objetos, alguém pode questionar o efeito no uso da memória da aplicação. O **BOX 2** trata desta questão, apresentando uma relação entre a criação de muitos objetos de valor e entidades e o consumo de memória;

BOX 2. Criar muitos objetos impacta o consumo de memória?

Alguém pode considerar que a criação de muitos objetos de valor e entidades é algo nocivo pelo grande número de objetos criados, e que melhor seria criar grandes serviços que tendem ao procedural, de forma a economizar memória. Contudo, este argumento geralmente não se sustenta. A maioria das máquinas virtuais usa, no garbage collector, o algoritmo generational copying para liberação de memória. Este algoritmo considera que a grande maioria dos objetos têm vida muito curta, sendo rapidamente descartados, e apenas poucos objetos têm vida longa. Desta forma, ao espalhar a lógica de grandes serviços que fazem muitas coisas em objetos pequenos e especializados, seu design passa, na verdade, a ser mais eficiente em termos de gerenciamento de memória – além de estar aderente às boas práticas de orientação a objetos.

- **Serviços:** Devem ser utilizados quando um dado conceito não faz sentido ao ser mapeado apenas para um objeto. Ele é definido em termos do que pode fazer, e desta forma, geralmente é nomeado como um verbo (como, por exemplo, GerenciadorDeCobrança). Um problema comum – e muito recorrente – é o de criar serviços para tudo, até mesmo para conceitos que poderiam ser representados como objetos, tornando o código procedural. Esta tendência deve ser evitada, de forma que um serviço só deve ser criado quando fizer sentido como um serviço – e que conceitos que podem ser representados como objetos devem ser utilizados como tal;

- **Módulos:** São um agrupamento de objetos. É o mesmo conceito dos pacotes do Java, com uma diferença importante: em DDD, os módulos precisam agrupar classes com similaridade semântica, enquanto em Java geralmente os pacotes são agrupados em termos de camadas com significado técnico. É comum que, em Java, um conjunto de classes que trata, por exemplo, de um domínio de cobrança, espalhe classes de cobrança em vários pacotes (um pacote de persistência, outro de serviços, outro de fachada, etc.), enquanto, no DDD, todas as classes devem estar num único pacote conceitual “cobrança”. Na verdade, o DDD sugere um modelo em camadas (com interface, aplicativo, domínio e infraestrutura), mas exige que toda lógica de negócio fique na camada de domínio (pode haver um repositório que converse com uma camada de infraestrutura para obter dados, mas o repositório deve ficar na



camada de domínio e representar algo para o negócio como, por exemplo, uma classe **RepositorioDeCobranca** ou uma classe **Cobrancas**;

- **Agregados:** Este conceito gira em torno do escopo de objetos agrupados por módulos. Ou seja, alguns objetos não deveriam ter acesso a outros, simplesmente por não haver nenhum sentido nisso. Este é um conceito poderoso, onde o design em termos de uma organização de visibilidade de objetos garante que objetos não sejam utilizados de forma indevida. Por exemplo, dentro do domínio de um carro, a entidade **Carro** pode ter acesso às classes **Roda** e **Pneu**. Contudo, uma entidade **Motor** não precisa ter acesso ao **Pneu**, mas precisa estar relacionada com o **Carro**. Em Java, este conceito pode ser implementado pelo uso dos modificadores de acesso em nível de classe (**public** e **default**) em pacotes: classes com o modificador **public** podem ser acessíveis por outros pacotes, e classes **default** estão restritas ao próprio pacote. Neste exemplo, **Carro** seria uma classe **public**, **Pneu** e **Roda** seriam classes **default**, e **Motor** estaria em outro pacote. A entidade **Carro**, por ser aquela que serve de “ponto de entrada” para o uso dos demais objetos do módulo, é chamada de *Raiz do Agregado*;

- **Fábricas:** Servem para a criação de objetos complicados. Podem criar agregados inteiros, encapsulando comportamento interno. Em vez de deixar a instanciação de vários objetos a cargo do cliente, é melhor que haja uma fábrica capaz de criar exatamente o que o cliente precisa, encapsulando os passos. Por exemplo, criar uma entidade **Carro** e suas dependências pode ser um procedimento custoso, que exige que o cliente saiba qual a ordem e relação entre todos os objetos, quando na verdade o cliente apenas deseja utilizar o carro. Cabe à fábrica dar um carro pronto ao cliente, sem que o mesmo precise conhecer todos os objetos que compõem um carro, abstraindo toda esta complexidade. Uma fábrica precisa conseguir criar um objeto (e suas dependências) de forma consistente. Quando fábricas criam objetos de valor, criam objetos imutáveis (o objeto criado não pode ser modificado), e quando criam entidades, criam objetos com um estado inicial, que poderá mudar posteriormente;

- **Repositórios:** Têm a função de abstrair a relação com o banco de dados (e outras tecnologias de integração/dados), de forma que código técnico não invada o domínio, destruindo o conceito de negócio em detrimento de um conceito técnico, espalhando lógica ao invés de encapsular e centralizar. Um repositório é um objeto de negócio que representa a recuperação de entidades e objetos de valor, abstraindo a tecnologia que efetivamente realiza a recuperação.

Faremos uso destes padrões do DDD em conjunto com as técnicas de TDD para gerar um código-fonte de qualidade e aderente às regras de negócio.

TDD e DDD: Relação conceitual e emprego da técnica

Conforme visto até aqui, os conceitos de TDD e DDD podem ser relacionados com design de software, embora visando objetivos

diferentes. Em TDD, a função do design é técnica. O design emerge através de código com alta coesão e baixo acoplamento, oriundo da prática de gerar testes antes de criar o código a ser testado. Já em DDD, a função do design é voltada à modelagem do negócio, sendo considerada até mesmo mais importante do que a tecnologia sendo empregada; o design é criado antes da construção do software.

Agora, iremos consolidar os aspectos de design já discutidos numa técnica que permita utilizar o design emergente do TDD com o design voltado ao negócio do DDD.

A técnica aqui empregada é composta pelos seguintes passos:

- Definir um problema de domínio a ser resolvido;
- Modelar o problema de acordo com o diagrama de classes e padrões do DDD, tais como Entidades, Objetos de Valor, Agregados, Repositórios, dentre outros, considerando módulos específicos, por conceito. Este problema não será profundamente modelado a priori: apenas será feita uma análise suficiente para que os conceitos iniciais estejam explícitos e claros;
- Criar um projeto em Java para a implementação;
- No projeto Java criado, iniciar o ciclo de TDD, guiado pelo domínio modelado. Implementar as classes identificadas de acordo com a técnica de TDD, gerando código com testes. Como a modelagem inicial de DDD já definiu o conceito a ser seguido, o TDD apenas será usado para implementá-lo. Posteriormente, no entanto, é natural que novos conceitos surjam pelo melhor entendimento do problema, sendo então representados em código por meio da prática do TDD e do conhecimento de padrões do DDD, sem que nova diagramação seja necessária.

Exemplo de caso: DDD e TDD num sistema real

Com os conceitos de TDD e DDD explicitados, a próxima etapa será a de aplicá-los em um problema real.

Definindo um problema

Para o exemplo aqui proposto, o problema escolhido foi o de gerenciamento pessoal de movimentação financeira. O que se deseja é um controle de movimentos de crédito/débito entre contas pessoais (contas bancárias, carteira, contas fixas – como luz, água – e variáveis). Com base no registro das movimentações, deseja-se obter relatórios consolidados, que apresentem gastos e ganhos mensais, e comparativos por período. Com isto, espera-se melhorar o controle pessoal financeiro, tornando os ganhos e gastos mais explícitos, de forma a otimizar as contas e permitir que haja maior excedente para ser investido.

Tendo o problema definido, passemos à elaboração de uma modelagem inicial do domínio.

Modelagem inicial do domínio

O primeiro passo é modelar o problema de forma simples, mas suficiente para que os conceitos estejam representados. O problema escolhido é representado como um diagrama de classes na **Figura 1**.

Na imagem, pode-se observar que o modelo está separado em três módulos: contas, movimentos financeiros e relatórios.

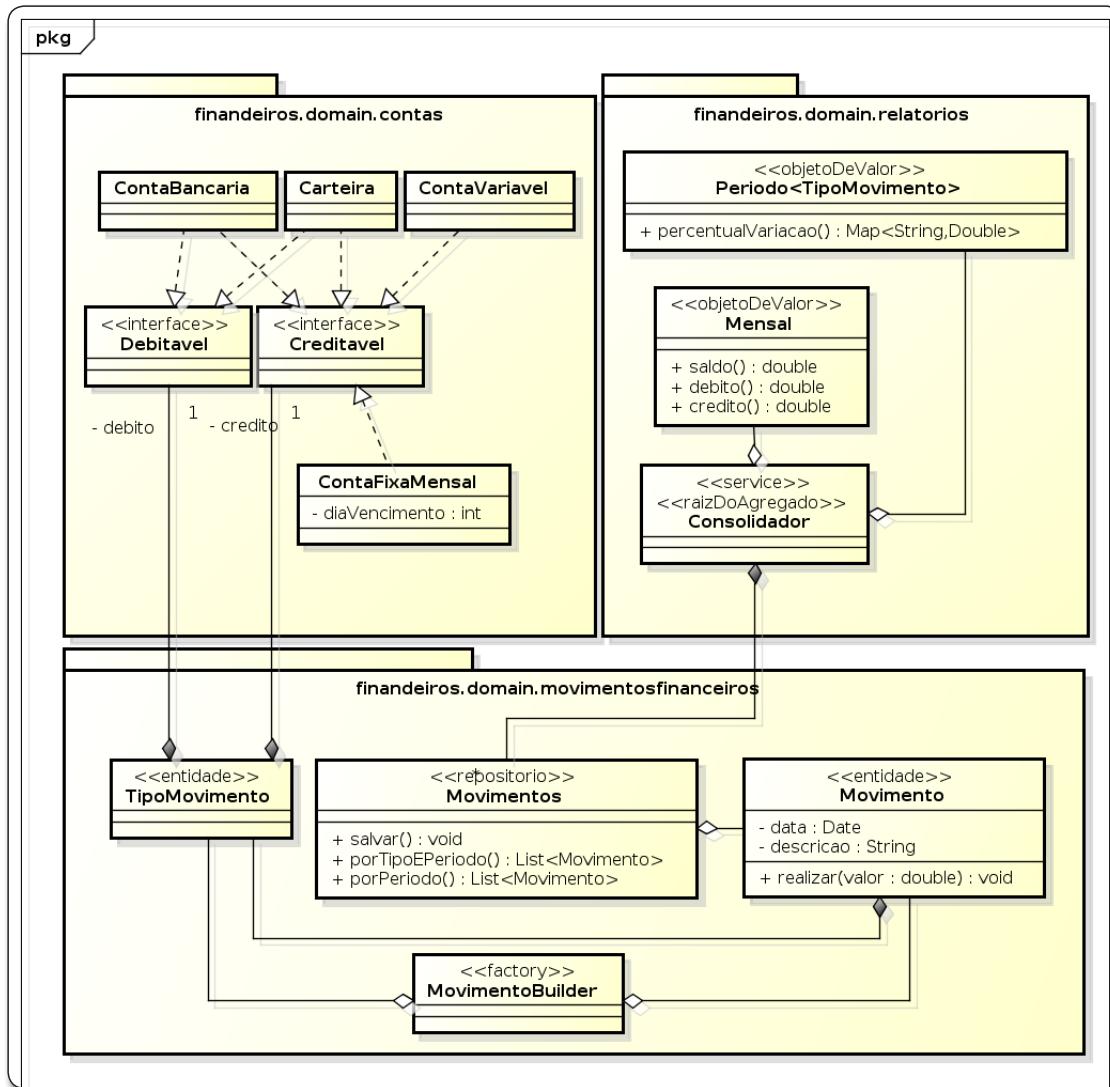


Figura 1. Modelo de domínio simplificado para gerenciamento financeiro

O domínio de contas possui classes com os tipos debitável e creditável, para representar o conceito de conta de crédito, débito ou de ambos, dependendo de sua natureza. A conta fixa mensal, diferentemente das outras, possui um dia de vencimento – e isto é importante porque desejamos considerar o caso de um pagamento antes do vencimento (como o de uma conta de luz, por exemplo).

O módulo *movimentosfinanceiros* provê entidades para representar o movimento financeiro e seu tipo. Desta forma, o domínio sugere que movimentações só podem ser realizadas com tipos de débito e crédito previamente associados. O nome *TipoMovimento*, definido para essa associação, não parece um bom nome, pois não é bem um tipo de movimento que estamos representando nesta classe (e nomes significativos importam, como dito anteriormente). Contudo, deixaremos desta forma por enquanto, por falta de um nome melhor (mas já tendo em mente que será preciso extrair um nome mais significativo desta classe em algum momento).

Ainda no módulo *movimentosfinanceiros*, observa-se que existem fábricas e repositórios. Os repositórios abstraem o acesso aos dados, e as fábricas criam objetos. Ambas as classes fazem parte do conceito de domínio, representando padrões do DDD e, por isto, encontram-se num módulo de domínio, em vez de estarem em módulos específicos para persistência ou serviço – como comumente é adotado em aplicações Java.

O módulo *relatorios* possui um serviço **Consolidador**. Esta classe é um serviço porque a forma de gerar relatórios não está diretamente associada a nenhuma entidade em específico. O **Consolidador** abstrai o uso de objetos específicos para diferentes formas de consolidação de dados, como Mensal ou por Período. Por isto, é o único objeto acessível por outros módulos, sendo considerado, por isto, a raiz do agregado.

Novamente, observa-se que o modelo não é completo, e sim apenas uma visão inicial do domínio. Conforme o problema vai sendo revisitado, cabe ao desenvolvedor atualizar o diagrama de

acordo, ou representar o conhecimento de alguma outra forma. Na técnica aqui proposta, o modelo não será mais atualizado: todo novo entendimento de conhecimento será representado diretamente em código-fonte pelo uso de TDD, o que significa que o diagrama serviu apenas para dar uma visão geral e inicial do problema. Isso não significa, contudo, que os padrões do DDD serão abandonados junto com o diagrama: durante a escrita de código com TDD, quando se tratar de código de domínio, nossas decisões *baby-step* serão pautadas considerando padrões do DDD.

Também pode-se observar que algumas decisões de design já tentaram ser antecipadas neste modelo: por exemplo, o padrão *command-query separation* (vide **BOX 3**) foi considerado ao modelar objetos de valor (cujos métodos apenas retornam valor sem modificar estado) e entidades (que possuem identidade, e atributos que podem ser modificados). Contudo, a ideia é a de não gastar muito tempo considerando técnicas de design na modelagem: o design de qualidade irá emergir posteriormente, pelo uso do TDD (e neste momento o conhecimento de técnicas específicas de design de qualidade poderá ser empregado).

BOX 3. Padrão Command-query separation

O termo *command-query separation* foi cunhado por Bertrand Meyer (o fundador do conceito de design por contrato), em seu livro *Object Oriented Software Construction*, de 1997. A ideia é a de dividir métodos em duas categorias principais: Queries e Commands:

- Queries são métodos que apenas retornam valores, sem alterar estado. São, portanto, livres de efeitos colaterais.
- Commands são métodos que alteram valores, gerando efeitos colaterais. Alteram valores, mas não os retornam.

O conceito de Meyer é o de que um método deve ser ou um query ou um command, mas não ambos. Ou seja, um método deve retornar um valor sem que haja efeitos colaterais, ou mudar um valor sem que nada seja retornado. Observa-se que Robert Martin também menciona algo parecido em seu livro *Clean Code*, de 2009, afirmando que um método deve fazer apenas uma coisa; neste contexto, deve retornar um valor ou modificar um valor.

Os ganhos com essa abordagem de design são de maior clareza, pois tornam explícito o funcionamento de um método pelo seu cliente: um método query pode ser chamado com mais confiança do que um método command, que altera algum estado (a propósito, ter um método com nome descritivo e conciso é suficiente para que possa ser chamado sem que a implementação precise ser entendida é também um bom indicativo de um bom design).

Ainda que existam exceções à regra, a prática *command-query separation* é mais uma a ser considerada ao criar design de software.

Criando um projeto em Java para implementação

Com o Maven devidamente instalado em seu ambiente, execute o comando da **Listagem 1** para criar um novo projeto.

Feito isso, abra o projeto em sua IDE preferida e edite o arquivo *pom.xml*. Neste arquivo, remova o conteúdo existente entre as tags `<dependencies></dependencies>`, e adicione, entre estas mesmas tags, o código informado na **Listagem 2**.

Concluída esta etapa, apague os diretórios abaixo de *src/main/java* e *src/test/java*. Estes diretórios são criados por default pelo archetype do Maven e não serão utilizados. Depois desta ação,

seu projeto estará configurado para utilizar os exemplos de TDD deste artigo, por meio do JUnit.

O código-fonte completo pode ser baixado no site da DevMedia. Além disso, é válido ressaltar que este artigo fará uso da IDE IntelliJ versão 14.0.2.

Listagem 1. Criação de um novo projeto utilizando o Maven.

```
/home/daniel]$ mvn archetype:generate -DgroupId=dddcomtdd
-DartifactId=finandeiros -DarchetypeArtifactId=maven-archetype-quickstart
-DinteractiveMode=false
```

Listagem 2. Dependências do arquivo pom.xml.

```
01 <dependency>
02   <groupId>junit</groupId>
03   <artifactId>junit</artifactId>
04   <version>4.12</version>
05   <scope>test</scope>
06 </dependency>
07 <dependency>
08   <groupId>org.hamcrest</groupId>
09   <artifactId>hamcrest-all</artifactId>
10  <version>1.3</version>
11  <scope>test</scope>
12 </dependency>
13 <dependency>
14   <groupId>joda-time</groupId>
15   <artifactId>joda-time</artifactId>
16   <version>2.7</version>
17 </dependency>
```

Aplicando TDD com base no diagrama

Observe que estamos com o diagrama de classes do modelo e a aplicação já configurada. Podemos, então, iniciar a implementação com TDD, que será aplicado a partir dos aspectos menos dependentes do diagrama. Isto significa que iniciaremos a escrita de código pelas classes periféricas (aqueles que apresentam menor dependência). Comecemos, portanto, com as classes de contas.



Dito isso, crie um pacote de nome `finandeiros.domain.contas` dentro do diretório `src/test/java`. Em seguida, dentro deste pacote, crie a classe `ContaFixaMensalTest`, com o método para testar a descrição e o dia de vencimento, conforme apresentado na [Listagem 3](#).

Seguindo o fluxo do TDD, deve-se escrever um código que quebre o teste, então escrever sua correção e depois refatorar. Façamos isto. O primeiro código consiste em criar uma instância de `ContaFixaMensal`, informando o nome da conta e a data de vencimento, de acordo com a [Listagem 4](#).

Listagem 3. Classe de teste para conta fixa mensal.

```
01 package finandeiros.domain.contas;
02 import org.junit.Test;
03 public class ContaFixaMensalTest {
04     @Test
05     public void deveTerDescricaoEDataDeVencimento() {
06     }
07 }
```

Listagem 4. Instanciando a conta fixa mensal no teste.

```
01 package finandeiros.domain.contas;
02 import org.junit.Test;
03 public class ContaFixaMensalTest {
04     @Test
05     public void deveTerDescricaoEDataDeVencimento() {
06         ContaFixaMensal conta = new ContaFixaMensal("Luz", 15);
07     }
08 }
```

Neste ponto, observe que o código-fonte nem compila, pois a classe cliente do teste ainda não existe. Deste modo, crie o pacote `finandeiros.domain.contas` dentro do diretório `src/main/java` e depois crie a classe `ContaFixaMensal`, com o construtor esperado pelo teste, de acordo com a [Listagem 5](#).

Note que agora o teste compila. Confirmando isso, complete o método de teste escrevendo o código para validar a existência de descrição e o dia de vencimento, conforme a [Listagem 6](#).

Neste momento, observe que o teste voltou a não compilar, porque os métodos `descricao()` e `diaVencimento()` não existem de fato. Portanto, crie os métodos de acordo com a [Listagem 7](#), que traz a implementação necessária apenas para fazer o teste compilar.

Listagem 5. Código da classe ContaFixaMensal.

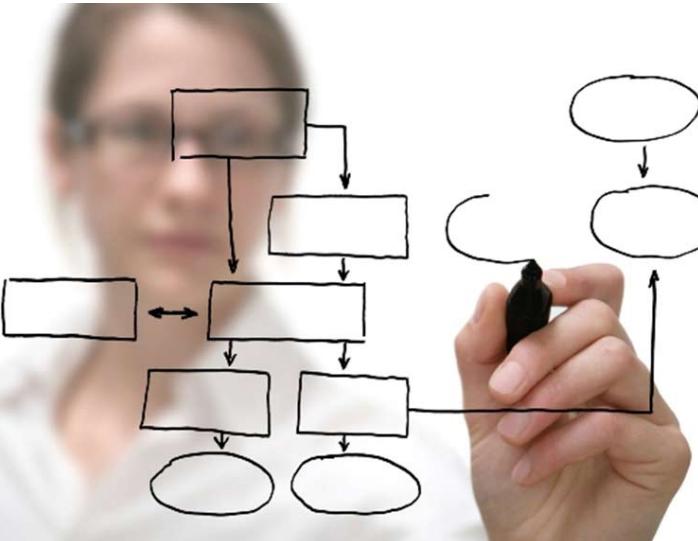
```
01 package finandeiros.domain.contas;
02 public class ContaFixaMensal {
03
04     private final String descricao;
05     private final int diaVencimento;
06
07     public ContaFixaMensal(String descricao, int diaVencimento) {
08         this.descricao = descricao;
09         this.diaVencimento = diaVencimento;
10     }
11 }
```

Listagem 6. Completando o teste da conta fixa mensal.

```
01 package finandeiros.domain.contas;
02
03 import org.junit.Test;
04 import static org.hamcrest.MatcherAssert.assertThat;
05 import static org.hamcrest.core.IsEqual.equalTo;
06
07 public class ContaFixaMensalTest {
08     @Test
09     public void deveTerDescricaoEDataDeVencimento() {
10         ContaFixaMensal conta = new ContaFixaMensal("Luz", 15);
11         assertThat(conta.descricao(), equalTo("Luz"));
12         assertThat(conta.diaVencimento(), equalTo(15));
13     }
14 }
```

Listagem 7. Código da classe ContaFixaMensal com métodos acessores.

```
01 package finandeiros.domain.contas;
02 public class ContaFixaMensal {
03
04     private final String descricao;
05     private final int diaVencimento;
06
07     public ContaFixaMensal(String descricao, int diaVencimento) {
08         this.descricao = descricao;
09         this.diaVencimento = diaVencimento;
10     }
11     public String descricao() {
12         return null;
13     }
14
15     public int diaVencimento() {
16         return 0;
17     }
18 }
```



Com isso o teste compila e pode ser executado. Ao fazer isso, observa-se, contudo, que o mesmo falha. Isto ocorre porque os métodos não estão retornando os valores esperados. É preciso, então, ajustar a classe sendo testada para que retorne os valores de acordo com o esperado. Sendo assim, altere a classe `ContaFixaMensal` de acordo com a [Listagem 8](#) e execute o teste novamente, verificando que o mesmo passa com sucesso.

Com o teste já passando, a última etapa é refatorar. Neste caso, a classe de produção não apresenta necessidade de refatoração, mas há uma duplicação na classe de teste que deve ser removida e seu método, por ter duas responsabilidades, pode ser dividido em dois. A versão final da classe de teste é apresentada na **Listagem 9**.

Listagem 8. Classe ContaFixaMensal com métodos acessores após a refatoração.

```
01 package finandeiros.domain.contas;
02 public class ContaFixaMensal {
03
04     private final String descricao;
05     private final int diaVencimento;
06
07     public ContaFixaMensal(String descricao, int diaVencimento) {
08         this.descricao = descricao;
09         this.diaVencimento = diaVencimento;
10    }
11    public String descricao() {
12        return null;
13    }
14
15    public int diaVencimento() {
16        return 0;
17    }
18}
```

Listagem 9. Refatorando a classe de teste da conta fixa mensal.

```
01 package finandeiros.domain.contas;
02
03 import org.junit.Test;
04 import static org.hamcrest.MatcherAssert.assertThat;
05 import static org.hamcrest.core.IsEqual.equalTo;
06
07 public class ContaFixaMensalTest {
08
09     private static final String DESCRIÇÃO = "Luz";
10     private static final int DATA_DIA_VENCIMENTO = 15;
11     private ContaFixaMensal conta;
12
13     @Before
14     public void setup() {
15         this.conta = new ContaFixaMensal(DESCRÍCIAO, DIA_VENCIMENTO);
16     }
17
18     @Test
19     public void deveTerDescrição() {
20         assertThat(conta.descricao(), equalTo(DESCRÍCIAO));
21     }
22
23     @Test
24     public void deveTerDataDeVencimento() {
25         assertThat(conta.diaVencimento(), equalTo(DIA_VENCIMENTO));
26     }
27}
```

Estes passos demonstraram como o TDD pode ser aplicado em conjunto com uma modelagem gerada com DDD para criação de um código-fonte com qualidade. O design resultante da prática do TDD acabou por ser estendido em relação ao diagrama, pois

não era dito que **ContaFixaMensal** precisava ter um construtor com dois parâmetros. O problema foi constatado ao criar a classe utilizando TDD. A atividade de desenvolvimento com TDD adicionou um detalhe ao design que a versão inicial da modelagem não havia considerado. Contudo, o código não alterou o conceito que o diagrama representa, mas sim apenas estendeu o design para um conhecimento emergente.

Seguindo o diagrama, vamos criar a interface **Creditavel** e fazer com que **ContaFixaMensal** a implemente. No momento, esta é apenas uma interface marcadora (vide **BOX 4**). Após criar e utilizar a interface (veja as **Listagens 10** e **11**), lembre-se de rodar o teste unitário novamente. Você notará que o código continua executando com sucesso, o que indica que os ajustes não quebraram a implementação existente (este é um exemplo bem óbvio que o código não seria quebrado, mas o hábito de executar os testes automatizados a cada alteração em código é uma boa prática que evidencia rapidamente problemas na consistência do design).

BOX 4. Interface marcadora

Uma interface marcadora é aquela que existe para definir uma semântica para o conjunto de classes que a implementa. É chamada de marcadora justamente por isto: apenas marca uma classe em termos de um significado. Na verdade, nada é implementado, pois a interface marcadora não possui métodos. Contudo, ao modelar design, sua importância emerge: podemos desejar que um dado objeto apenas lide com objetos que possuem uma semântica específica – no caso deste artigo, o **TipoMovimento** recebe como primeiro parâmetro de seu construtor apenas contas que sejam **Debitáveis**. Daí uma utilidade deste tipo especial de interface em termos de design.

Listagem 10. Código da interface marcadora **Creditavel**.

```
01 package finandeiros.domain.contas;
02
03 public interface Creditavel {
04 }
```

Listagem 11. Código da classe **ContaFixaMensal** com interface marcadora.

```
01 package finandeiros.domain.contas;
02 public class ContaFixaMensal implements Creditavel {
03
04     private final String descricao;
05     private final int diaVencimento;
06
07     public ContaFixaMensal(String descricao, int diaVencimento) {
08         this.descricao = descricao;
09         this.diaVencimento = diaVencimento;
10    }
11    public String descricao() {
12        return null;
13    }
14
15    public int diaVencimento() {
16        return 0;
17    }
18}
```

Modelagem e implementação com TDD e DDD

Feito isso, codificaremos a interface **Debitavel** e a implementação **ContaBancaria**. O fluxo de TDD a ser aplicado para gerar esse código é o mesmo utilizado anteriormente e, portanto, não será reproduzido novamente. O resultado deve ser algo semelhante ao conteúdo apresentado nas **Listagens 12 a 14**.

Listagem 12. Código da classe de teste da classe ContaBancaria.

```
01 package finandeiros.domain.contas;
02 import org.junit.Before;
03 import org.junit.Test;
04 import static org.hamcrest.CoreMatchers.instanceOf;
05 import static org.hamcrest.MatcherAssert.assertThat;
06 import static org.hamcrest.core.AreEqual.equalTo;
07
08 public class ContaBancariaTest {
09     private static final String NOME_CONTA = "Minha conta Bradesco";
10     private ContaBancaria conta;
11
12     @Before
13     public void setup() {
14         this.conta = new ContaBancaria(NOME_CONTA);
15     }
16
17     @Test
18     public void deveTerUmNome() {
19         assertThat(conta.nome(), equalTo(NOME_CONTA));
20     }
21     @Test
22     public void deveSerUmaContaDebitavel() {
23         assertThat(conta, instanceOf(Debitavel.class));
24     }
25
26     @Test
27     public void deveSerUmaContaCreditavel() {
28         assertThat(conta, instanceOf(Creditavel.class));
29     }
30 }
```

Listagem 13. Código da classe ContaBancaria.

```
01 package finandeiros.domain.contas;
02
03 public class ContaBancaria implements Creditavel, Debitavel {
04     private String nomeConta;
05
06     public ContaBancaria(String nomeConta) {
07         this.nomeConta = nomeConta;
08     }
09     public String nome() {
10         return nomeConta;
11     }
12 }
```

Listagem 14. Código da interface Debitavel.

```
01 package finandeiros.domain.contas;
02 public interface Debitavel {
03 }
```

No entanto, por mais que possa haver discussão a respeito de um teste ser válido ou não, o valor de design emergente não pode ser descartado ao fazer uso do TDD neste cenário. A prática de TDD permitiu que o construtor fosse elaborado do ponto de vista de seu cliente – ou seja, o cliente passa os valores desejados para o construtor e, a partir daí, o construtor é criado. O teste possui, ao menos, uma função de design ao especificar como a classe deveria ser instanciada. Desta forma, até os menores testes são importantes na dinâmica de TDD – ainda que, posteriormente, alguns testes possam ser removidos ou refatorados, caso julguem-se desnecessários.

Agora, continuaremos aplicando a dinâmica de TDD para implementar a classe **TipoMovimento**, que depende das classes creditáveis e debitáveis, apresentadas anteriormente. Com base nisso, as classes expostas nas **Listagens 15 e 16** devem ser criadas.

Listagem 15. Código da classe de teste da classe TipoMovimento.

```
01 package finandeiros.domain.movimentosfinanceiros;
02
03 //imports omitidos...
04
05 public class TipoMovimentoTest {
06
07     private TipoMovimento tipoMovimento;
08
09     @Before
10     public void setup() {
11         this.tipoMovimento = new TipoMovimento(new ContaBancaria(
12             "Conta Bradesco"), new
13             ContaFixaMensal("Luz", 15));
14     }
15     @Test
16     public void deveTerContaCredito() {
17         assertThat(tipoMovimento.contaCredito(), instanceOf(Creditavel.class));
18     }
19     @Test
20     public void deveTerContaDebito() {
21         assertThat(tipoMovimento.contaDebito(), instanceOf(Debitavel.class));
22     }
```

Listagem 16. Código da classe TipoMovimento.

```
01 package finandeiros.domain.movimentosfinanceiros;
02
03 import finandeiros.domain.contas.Creditavel;
04 import finandeiros.domain.contas.Debitavel;
05
06 public class TipoMovimento {
07
08     private final Debitavel contaDebito;
09     private final Creditavel contaCredito;
10
11     public TipoMovimento(Debitavel contaDebito, Creditavel contaCredito) {
12         this.contaDebito = contaDebito;
13         this.contaCredito = contaCredito;
14     }
15     public Creditavel contaCredito() {
16         return contaCredito;
17     }
18     public Debitavel contaDebito() {
19         return contaDebito;
20     }
21 }
```

Neste momento, alguém pode questionar a validade dos testes. Afinal, até agora os testes apenas validaram se os atributos de um construtor conseguem ser recuperados corretamente. Uma vez que este tipo de teste pode ser considerado muito básico, é natural que surjam dúvidas sobre a utilidade do mesmo.

Observa-se que **TipoMovimento** e seu teste foram criados em um módulo diferente e que **TipoMovimento** apenas é um agregador de contas de crédito e débito. Como o conceito de “agregador de contas” parece fazer mais sentido do que o conceito de tipo de movimento para este caso, o nome da classe deve ser alterado para **AgregadorContas**, refletindo assim o novo conceito.

E agora que a classe chama-se **AgregadorContas**, parece redundante que os métodos tenham o prefixo “conta”. Portanto, tal prefixo será removido do nome dos métodos, resultando no código apresentado na **Listagem 17**.

O código de teste deve ser ajustado de acordo e executado mais uma vez, para garantir que o sistema continue funcionando conforme desejado.

Ao consultar novamente o diagrama, verifica-se que a classe **TipoMovimento** (agora **AgregadorContas**) é utilizada pela

Listagem 17. Classe TipoMovimento renomeada e refatorada buscando nomenclatura significativa.

```
01 package finandeiros.domain.movimentosfinanceiros;
02 import finandeiros.domain.contas.Creditavel;
03 import finandeiros.domain.contas.Debitavel;
04
05 public class AgregadorContas {
06     private final Debitavel contaDebito;
07     private final Creditavel contaCredito;
08
09     public AgregadorContas(Debitavel contaDebito, Creditavel contaCredito) {
10         this.contaDebito = contaDebito;
11         this.contaCredito = contaCredito;
12     }
13     public Creditavel credito() {
14         return contaCredito;
15     }
16     public Debitavel debito() {
17         return contaDebito;
18     }
19 }
```

Listagem 18. Código da classe de teste da classe Movimento.

```
01 package finandeiros.domain.movimentosfinanceiros;
02
03 //imports omitidos...
04
05 public class MovimentoTest {
06     private static final String DESCRICAO = "pagamento de conta de luz";
07     private Movimento movimento;
08
09     @Before
10     public void setup() {
11         AgregadorContas agregadorContas = new AgregadorContas(
12             new ContaBancaria("Conta Bradesco"),
13             new ContaFixaMensal("Luz", 15));
14         this.movimento = new Movimento(agregadorContas,
15             new DateTime(2015, 6, 11, 15, 20).toDate(), DESCRICAO);
16     }
17     @Test
18     public void deveTerContaCredito() {
19         assertEquals(movimento.contaCredito(), notNullValue());
20     }
21     public void deveTerContaDebito() {
```

entidade **Movimento** e pela factory de nome **MovimentoBuilder**. Vamos, então, analisar a construção de cada um desse objetos.

A classe **Movimento** é construída de modo a possuir uma data e uma descrição, e deve ser capaz de realizar um movimento entre contas de crédito e débito. No caso de uma conta fixa mensal, sabe-se que o sistema não deve permitir o pagamento após o vencimento.

Nesta etapa, da mesma forma que realizado anteriormente, execute mais uma vez o fluxo de TDD, mas considerando a classe **Movimento**. O código gerado a partir disso é apresentado nas **Listagens 18 e 19**.

Repare que a classe **Movimento** foi criada com visibilidade **default**. Isto significa que não queremos que classes de outros módulos acessem **Movimento** diretamente. O acesso deverá ser feito pelas classes que implementam os padrões Factory ou Repository. Relacionada a isso, uma prática utilizada pelo DDD é a da permissão de acesso de classes, que viabiliza o controle de quais objetos podem e quais não podem ser acessados por clientes de outros módulos .

Para construir um **Movimento**, vamos implementar a Factory **MovimentoBuilder**. Esta classe fará uso do padrão de projeto Builder por este ser um padrão adequado para a construção de novos objetos. O resultado pode ser verificado nas **Listagens 20 e 21**.

Para simplificar o exemplo, esta implementação de builder não está considerando o cenário onde nem todos os parâmetros necessários são informados. No entanto, a classe builder pode ser revisitada posteriormente, para melhorar as validações por meio da extensão dos cenários de testes, que tornarão os problemas explícitos.

A próxima etapa é implementar o repositório responsável por salvar e recuperar os movimentos.

```
22     assertEquals(movimento.contaDebito(), notNullValue());
23 }
24 @Test
25 public void deveTerData() {
26     assertEquals(movimento.data(), notNullValue());
27 }
28 @Test
29 public void deveTerDescricao() {
30     assertEquals(movimento.descricao(), notNullValue());
31 }
32 @Test
33 public void deveRealizarMovimentoIgnorandoSinalNegativo() {
34     movimento.realizar(-1500.0);
35     assertEquals(movimento.credito(), equalTo(1500.0));
36 }
37 @Test
38 public void deveRealizarMovimentoIgnorandoSinalPositivo() {
39     movimento.realizar(1200.0);
40     assertEquals(movimento.debito(), equalTo(-1200.0));
41 }
42 }
```

Modelagem e implementação com TDD e DDD

Listagem 19. Código da classe Movimento.

```
01 package finandeiros.domain.movimentosfinanceiros;
02 import java.util.Date;
03
04 class Movimento {
05     private AgregadorContas agregadorContas;
06     private final Date data;
07     private final String descricao;
08     private double valor;
09
10    Movimento(AgregadorContas agregadorContas, Date data, String descricao) {
11        this.agregadorContas = agregadorContas;
12        this.data = data;
13        this.descricao = descricao;
14    }
15    public void realizar(double valor) {
16        this.valor = Math.abs(valor);
17    }
18    public double credito() {
19        return valor;
20    }
21    public double debito() {
22        return -valor;
23    }
24    public Date data() {
25        return data;
26    }
27    public Debitavel contaDebito() {
28        return agregadorContas.debito();
29    }
30    public Creditavel contaCredito() {
31        return agregadorContas.credito();
32    }
33    public String descricao() {
34        return descricao;
35    }
36}
```

Listagem 20. Código da classe de teste da classe MovimentoBuilder.

```
01 package finandeiros.domain.movimentosfinanceiros;
02
03 //imports omitidos...
04
05 public class MovimentoBuilderTest {
06     private Debitavel contaDebito = new ContaBancaria("Bradesco");
07     private Creditavel contaCredito = new ContaFixaMensal("Luz", 15);
08     private Date data = new DateTime(2015, 7, 11, 0, 0).toDate();
09     private String descricao = "pagamento conta luz de julho";
10     private Movimento movimento;
11
12     @Before
13     public void setup() {
14         MovimentoBuilder movimentoBuilder = new MovimentoBuilder();
15         this.movimento = movimentoBuilder.comContaDebito(contaDebito)
16             .eContaCredito(contaCredito)
17             .eData(data).eDescricao(descricao).build();
18     }
19     @Test
20     public void deveCriarMovimento() {
21         assertThat(movimento, notNullValue());
22     }
23     @Test
24     public void deveCriarMovimentoComContaCredito() {
25         assertThat(movimento.contaCredito(), equalTo(contaCredito));
26     }
27     @Test
28     public void deveCriarMovimentoComContaDebito() {
29         assertThat(movimento.contaDebito(), equalTo(contaDebito));
30     }
31     @Test
32     public void deveCriarMovimentoComData() {
33         assertThat(movimento.data(), equalTo(data));
34     }
35     @Test
36     public void deveCriarMovimentoComDescricao() {
37         assertThat(movimento.descricao(), equalTo(descricao));
38 }
```

Listagem 21. Código da classe MovimentoBuilder.

```
01 package finandeiros.domain.movimentosfinanceiros;
02
03 import finandeiros.domain.contas.Creditavel;
04 import finandeiros.domain.contas.Debitavel;
05 import java.util.Date;
06
07 public class MovimentoBuilder {
08     private Debitavel contaDebito;
09     private Creditavel contaCredito;
10     private Date data;
11     private String descricao;
12
13     public MovimentoBuilder comContaDebito(Debitavel contaDebito) {
14         this.contaDebito = contaDebito;
15         return this;
16     }
17     public MovimentoBuilder eContaCredito(Creditavel contaCredito) {
18         this.contaCredito = contaCredito;
19         return this;
20     }
21     public MovimentoBuilder eData(Date data) {
22         this.data = data;
23         return this;
24     }
25     public MovimentoBuilder eDescricao(String descricao) {
26         this.descricao = descricao;
27         return this;
28     }
29     public Movimento build() {
30         AgregadorContas agregadorContas =
31             new AgregadorContas(contaDebito, contaCredito);
32         return new Movimento(agregadorContas, data, descricao);
33     }
34 }
```

De modo a simplificar a demonstração, iremos implementar o repositório como um *stub*, para abstrair as complexidades de acesso a dados (veja o **BOX 5**).

BOX 5. Stubs e Mocks: simulando comportamento

Algumas vezes, precisamos simular o comportamento de objetos, seja para abstrair um acesso a recursos externos dos quais não temos acesso, seja para construir parte de uma funcionalidade, deixando a API pronta para que seja posteriormente implementada. Para cenários como este, existe o conceito de Stub. Um Stub é um método que retorna sempre um mesmo objeto ou valor. Isto não é útil para código de produção, mas pode ser importante durante a fase de desenvolvimento (quando nem todas as APIs externas podem estar disponíveis, por exemplo). Já o conceito de mock é diferente do de stub, pois o mock é um código que “programa” o comportamento de um método em tempo de execução de teste para que o mesmo se comporte de acordo com uma determinada necessidade, simulando um comportamento falso. Mocks são comuns em cenários onde se quer testar alguns aspectos de um objeto, ignorando ou alterando o comportamento de outros para algo predefinido (comum em casos de código com alto acoplamento e baixa coesão, como códigos procedurais).

Seguindo o mesmo roteiro de criação de testes e código do TDD, e com base no diagrama de classes gerado com DDD, foram criadas três classes: a classe de teste do repositório (**MovimentosTest**, que testa os movimentos); a classe do repositório (**Movimentos**);

e a classe do stub do repositório (**MovimentosStub**, que implementa o stub de **Movimentos** através do mecanismo de herança e que deve ser removida assim que **Movimentos** puder implementar código real). Veja as **Listagens 22 a 24**.

Listagem 22. Código da classe de teste da classe Movimentos.

```
01 package finandeiros.domain.movimentosfinanceiros;
02
03 //imports omitidos...
04
05 public class MovimentosTest {
06     private Movimentos repositorio;
07
08     @Before
09     public void setup() {
10         repositorio = new MovimentosStub();
11     }
12
13     @Test
14     public void deveSalvarMovimentosNovos() {
15         Debitavel contaDebito = new ContaBancaria("Bradesco");
16         Creditable contaCredito = new ContaFixaMensal("Luz", 15);
17         Date data = new DateTime(2015, 7, 11, 0, 0).toDate();
18         String descricao = "pagamento conta luz de julho";
19         Movimento movimento = new MovimentoBuilder()
20             .comContaDebito(contaDebito)
21             .eContaCredito(contaCredito).eData(data).eDescricao(descricao)
22             .build();
23
24         repositorio.salvar(movimento);
25         assertThat(movimento.getId(), notNullValue());
26     }
27
28     @Test
29     public void deveObterMovimentosPorPeriodo() {
30         Date inicio = new DateTime(2015, 1, 1, 0, 0).toDate();
31         Date fim = new DateTime(2015, 6, 30, 0, 0).toDate();
32         List<Movimento> movimentosPorPeriodo =
33             repositorio.todosPorPeriodo(inicio, fim);
34         assertThat(movimentosPorPeriodo.size(), equalTo(5));
35     }
36 }
```

Listagem 23. Código da classe Movimentos.

```
01 package finandeiros.domain.movimentosfinanceiros;
02 import finandeiros.domain.contas.ContaBancaria;
03 import java.util.ArrayList;
04 import java.util.Date;
05 import java.util.List;
06
07 public class Movimentos {
08     public void salvar(Movimento movimento) {
09         throw new NotImplementedException();
10     }
11
12     public List<Movimento> todosPorPeriodo(Date inicio, Date fim) {
13         throw new NotImplementedException();
14     }
15 }
```

Listagem 24. Código da classe MovimentosStub.

```
01 package finandeiros.domain.movimentosfinanceiros;
02
03 import java.util.ArrayList;
04 import java.util.Date;
05 import java.util.List;
06
07 public class MovimentosStub extends Movimentos {
08     private List<Movimento> stubList = new ArrayList<Movimento>();
09
10     public MovimentosStub() {
11         createStub();
12     }
13
14     private void createStub() {
15         MovimentoBuilder builder = new MovimentoBuilder();
16         for (int i = 0; i < 5; i++) {
17             stubList.add(builder.build());
18         }
19     }
20
21     @Override
22     public void salvar(Movimento movimento) {
23         movimento.setId(1L);
24     }
25
26     @Override
27     public List<Movimento> todosPorPeriodo(Date inicio, Date fim) {
28         return stubList;
29     }
30 }
```

Pode-se verificar, nas **Listagens 22 e 24**, que os métodos **getId()** e **setId()** foram adicionados ao **Movimento**. Isto foi feito porque ao salvar um movimento é preciso que haja um ID, de forma a garantir a identidade da entidade. Note ainda que o código dessas listagens não compila, justamente por não haver tais métodos. Seguindo a prática do TDD, crie tais métodos e execute novamente os testes, que confirmarão o sucesso na compilação.

Dos métodos de consulta definidos no diagrama, o método **porPeriodo()** foi implementado como **todosPorPeriodo()**, por ser um nome mais descriptivo, e o método **porTipoEPeriodo()** não foi implementado (não observou-se a necessidade de implementá-lo neste momento).

Com isto, as classes dos módulos de movimentos financeiros e contas foram implementadas, aplicando TDD com base em conceitos e padrões de domínio oriundos das práticas de DDD. O código do módulo de relatórios não será apresentado neste artigo, ficando sua implementação a cargo do leitor, como um exercício que possibilita a prática dos conceitos aqui explicitados.

Este artigo procurou apresentar uma técnica que faz uso do conjunto das práticas e padrões de TDD e DDD. Ainda assim, existem vários outros padrões de DDD que não foram apresentados, como contextos delimitados, mapas de contexto, especificações, etc. Ao leitor que desejar se aprofundar no assunto, é essencial a leitura do livro *"Domain-Driven Design"*, de Eric Evans, para maior

aprofundamento nos padrões de DDD, e do livro “*Test-Driven Development By Example*”, de Kent Beck, para maior entendimento da dinâmica da aplicação de TDD em problemas reais.

O problema proposto, embora tenha adotado toda a dinâmica da relação entre os conceitos, não foi completamente finalizado. O repositório foi simulado, faltando a escrita do código-fonte para sua comunicação com o banco de dados e, a partir daí, a inclusão do framework Spring (e Spring Test) para testes automatizados com o código integrado com a base de dados. Além disto, todo o pacote de relatórios não foi implementado, ficando este esforço a cargo do leitor, como oportunidade de pôr em prática o conteúdo abordado.

Ao fim destas implementações haverá um domínio testável, mas ainda cabe ao leitor a inclusão deste em uma aplicação real. No caso de fazer isto em uma aplicação web, pode-se incorporar à aplicação um framework MVC, por ser um padrão consolidado para a construção de soluções web. Outra prática comum em aplicações reais é a divisão do código-fonte em camadas, que separam as responsabilidades de apresentação, negócio e persistência. O livro *Domain-Driven Design* oferece um modelo para este tipo de divisão.

Vale frisar, ainda, que o domínio de negócios apresentado pode ser estendido para envolver novos conceitos, como o investimento no Tesouro Direto. Contudo, antes de continuar por este caminho, sugere-se a leitura do padrão do DDD Contexto Delimitado.

Autor



Daniel Medeiros de Assis

daniel.medeiros.assis@gmail.com

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

Mestrando 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:

Livro “The Passionate Programmer” na Amazon.

<http://www.amazon.com/The-Passionate-Programmer-Remarkable-Development/dp/1934356344>

Livro “Object-Oriented Software Construction” na Amazon.

<http://www.amazon.com/gp/product/0136291554?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0136291554>

Artigo “Is Design Dead”, de Martin Fowler.

<http://martinfowler.com/articles/designDead.html>

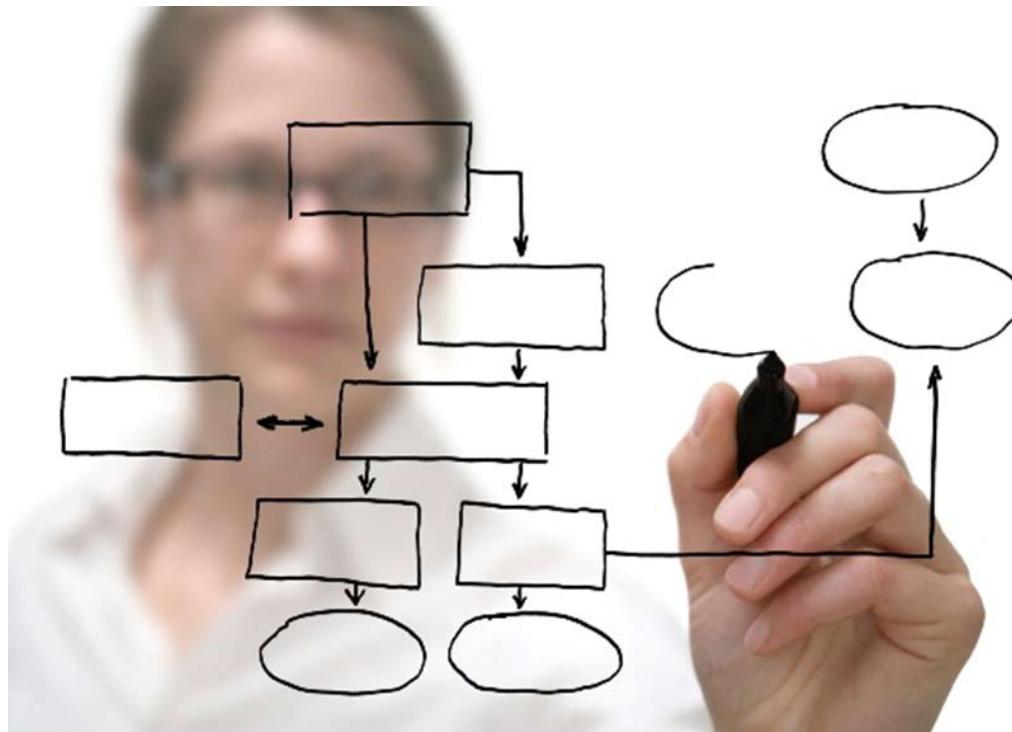
Maven in 5 Minutes.

<https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



FÓRUM DEVMEDIA

O lugar perfeito para você ficar por dentro de tudo
o que acontece nas tecnologias do mercado atual



No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!

ACESSE AGORA
www.devmedia.com.br/forum

Apache Spark: Como criar um mecanismo de sugestão de produtos

Veja neste artigo como turbinar seu e-commerce com o Apache Spark e Big Data

Atualmente a Amazon, maior e-commerce do mundo, possui em seu estoque cerca de 400 mil livros sobre TI, mais de 1 milhão de livros classificados como Matemática e Ciências e quase dois milhões categorizados em Negócios e Finanças. Seria impossível analisar todos esses livros de forma manual (considerando que cada página demore um segundo para ser carregada, seria necessário quase uma semana de acesso contínuo para visualizar apenas os livros de TI). Por isso, os sistemas computacionais, especialmente os de e-commerce, investem cada vez mais em mecanismos de buscas e formas de sumarizar, apresentar e recomendar produtos que facilitem a vida de seus clientes na tarefa de encontrar seu produto ideal.

Como uma das respostas à dificuldade das pessoas em escolher entre uma grande variedade de produtos e serviços, muitos sistemas de recomendação foram sugeridos. A evolução destes sistemas e o fato deles trabalharem com grandes bases de informações permitiram que recomendações computacionais pudesse ter – muitas vezes – uma credibilidade maior que uma indicação humana. Os mecanismos desse tipo normalmente usam algum modelo de Aprendizado de Máquina, dentre os quais se destaca a Filtragem Colaborativa, proposta no início dos anos 90. Nessa época, os autores do Tapestry, um dos primeiros sistemas de recomendação, cunharam a expressão “Filtragem Colaborativa”, pois esse sistema recomenda informações com o auxílio humano, ou seja, filtra conteúdos via a colaboração de grupos de interessados.

Gigantes como a Amazon, Google e Ebay já usam esse tipo de mecanismo há anos, porém os avanços

Fique por dentro

Esse artigo é útil para estudantes e profissionais que desejam adicionar capacidade de aprendizado de máquina aos seus sistemas web. Gigantes como a Amazon, Google e Ebay já usam esse tipo de mecanismo há anos, porém os avanços motivados pela popularização do Big Data e de Computação em Nuvem permitem hoje que essas tecnologias estejam acessíveis para qualquer desenvolvedor Java. Nesse aspecto, o Apache Spark se destaca por oferecer uma maneira simples, rápida e altamente escalável para o desenvolvimento de sistemas baseados em Aprendizado de Máquina.

Assim, esse artigo demonstra o uso do Apache Spark em um mecanismo para um e-commerce hipotético, no qual os clientes recebem sugestões de produtos a partir de recomendações feitas por outros usuários. Para completar a aplicação, os dados são armazenados no Elasticsearch, em uma arquitetura inspirada na Arquitetura Lambda, um padrão no desenvolvimento de software para Big Data que facilita a organização e o entendimento das aplicações.

motivados pela popularização do Big Data e de Computação em Nuvem permitem hoje que essas tecnologias estejam acessíveis a e-commerces de qualquer tamanho. Nesse aspecto, o Apache Spark se destaca por oferecer uma maneira simples, rápida e altamente escalável para o desenvolvimento de sistemas baseados em Aprendizado de Máquina.

Usando o Spark, sites de e-commerce – infinitamente menores que a Amazon – podem ter acesso ao mesmo tipo de mecanismo, melhorando a experiência de uso de seus sistemas e, principalmente, alavancando suas vendas. Para isso, o Spark oferece pré-implementadas as funções de Filtragem Colaborativa.

Ao longo dos anos, vários pesquisadores acabaram adotando esta terminologia para denominar qualquer tipo de sistema de recomendação, porém é importante frisar que existem sistemas de recomendação sem nenhuma colaboração entre as pessoas (por exemplo, quando as recomendações são baseadas no posicionamento geográfico).

Com base nesses conceitos, este artigo apresentará o emprego do mecanismo de filtragem colaborativa dentro do Apache Spark, com especial foco em sistemas de e-commerce. Ao longo do estudo, será mostrada a utilização do MLlib – nome que reflete a sigla em inglês para biblioteca de aprendizado de máquina –, componente do Spark que implementa distintos mecanismos de aprendizado de máquina, incluindo a Filtragem Colaborativa. Além disso, outros importantes aspectos do Spark serão abordados, dentre os quais ressaltamos: o mecanismo de clusters, o uso intensivo de memória principal e a baixa latência, como principal requisito de projeto.

Conceitos principais

Os sistemas de recomendação visam determinar o conteúdo mais relevante que deve ser apresentado para o usuário e são caracterizados por realizar filtragens – ou seja, sumarizar para escolher os itens mais relevantes – em um conjunto de conteúdos, que podem, por exemplo, ser produtos em um e-commerce, notícias em um portal, ou posts em uma rede social.

Existem diversas formas de lidar com a recomendação, mas duas são as principais: recomendação baseada no usuário ou baseada no item. Como resumido pela **Figura 1**, no tipo de recomendação Baseada no Item, o usuário receberá recomendações de itens similares a itens preferidos no passado, enquanto na Baseada no Usuário, o usuário receberá recomendações de itens que pessoas com gostos similares aos dele preferiram no passado.

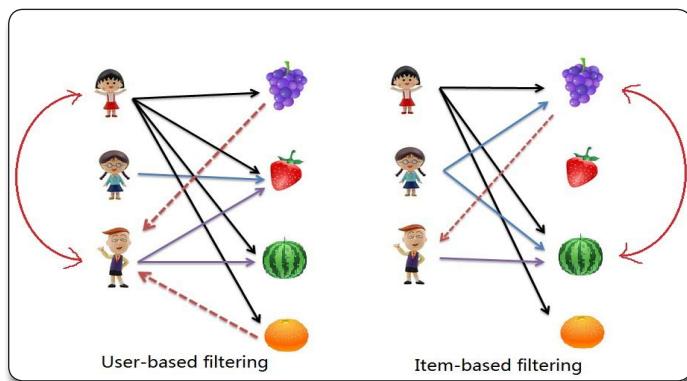


Figura 1. Tipos de recomendação

A Filtragem Colaborativa é o principal método para recomendação baseada em usuário e é parte da área de Aprendizado de Máquina, já que para ser efetivo, esse método espera que haja uma fase de treinamento, onde o mecanismo irá aprender quais itens são mais relevantes para cada usuário. O nome do método reflete o fato que esse tipo de sistema utiliza a opinião de usuários

sobre um determinado item para sugerir itens que outros usuários também podem estar interessados, em vez de utilizar a análise das características dos itens.

De maneira mais formal, a ideia da filtragem colaborativa pode ser explicada pela **Tabela 1**, onde são apresentadas as avaliações (notas de 0 a 10) de três clientes (primeira coluna) para quatro livros (primeira linha). Percebe-se, a partir desta tabela, que José não realizou a avaliação do produto “Introdução ao Elasticsearch”, entretanto, avaliou os produtos “Java 8: a bíblia” e “Dominando o Apache Spark” com notas altas e o produto “Big Data e seus conceitos” com nota baixa. Percebe-se, também, que uma avaliação semelhante à do cliente José foi realizada por Maria, devido à proximidade das notas atribuídas aos produtos. Diante dessa situação e considerando a abordagem colaborativa, que recomenda produtos de acordo com as avaliações realizadas por usuários similares, uma ótima recomendação para José seria o livro “Introdução ao Elasticsearch”, uma vez que Maria possui avaliações semelhantes a ele e avaliou esse produto muito bem.

	Java 8: a bíblia	Introdução ao Elasticsearch	Dominando o Apache Spark	Big Data e seus conceitos
Maria	9	8	10	1
João	2	1	0	1
José	10	-	8	2

Tabela 1. Exemplo de filtragem colaborativa

Além disso, a Filtragem Colaborativa é subdividida em duas categorias: baseada em memória e baseada em modelos. A primeira utiliza dados de opiniões do usuário (geralmente expressada por uma nota) para calcular a similaridade entre os usuários ou entre os itens. Esta era a abordagem usada em muitos sistemas comerciais por ser eficaz e fácil de implementar. Na segunda são utilizados algoritmos de aprendizado de máquina para criar modelos a partir dos padrões encontrados. Existem muitos algoritmos de Filtragem Colaborativa com base em modelos, entre eles os de redes bayesianas, clustering, semântica latente, análise probabilística e com processo de decisão baseado em modelos de Markov.

Esses conceitos não têm relação com o Big Data por si só. Isto é, a Filtragem Colaborativa e o Aprendizado de Máquina são independentes de qualquer um dos novos mecanismos para Big Data. Porém, devido ao incrível aumento na quantidade de dados armazenados nos sistemas computacionais, a aprendizagem de máquina e os sistemas de recomendação são cada vez mais importantes para o Big Data.

Nesse contexto, o Apache Spark é uma ferramenta nova e extremamente interessante, pois, assim como outros novos frameworks para Big Data desenvolvidos pela Fundação Apache (Kafka, Flume e Storm), também é uma solução voltada para Fast Data (um conceito considerado a evolução do Big Data, que prevê a análise de grandes quantidades de informação em tempo real), compartilhando o foco em baixa latência nas respostas e oferecendo facilidade

Apache Spark: Como criar um mecanismo de sugestão de produtos

de uso e o uso de streaming. Como diferencial em relação aos outros frameworks para Fast Data, o Spark possui interessantes ferramentas para processamento de grafos, de análise estatística e aprendizado de máquina pré-implementadas, conforme ilustrado na **Figura 2**. Neste artigo, o foco será sobre a MLlib, ferramenta que oferece os algoritmos de aprendizado de máquina tradicionais e necessários para a construção de aplicações inteligentes.

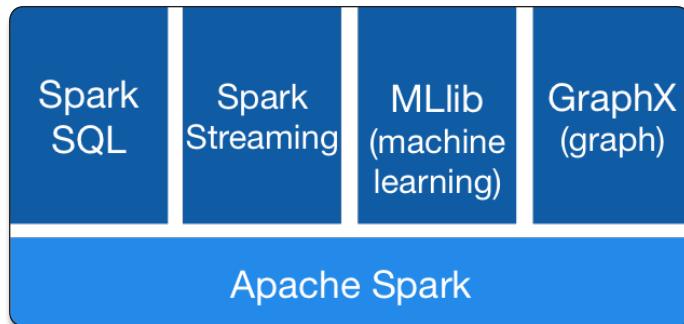


Figura 2. Componentes do Apache Spark

Apache Spark

Ainda que não seja o objeto de estudo deste artigo, é importante entender o funcionamento interno e os conceitos gerais do cluster Spark.

A arquitetura do Spark, ilustrada na **Figura 3**, é bastante simples. Com base nela, cada aplicação deve instanciar um contexto do Spark (*Spark Context*), elemento que contém as informações necessárias para acessar o cluster onde a mesma será executada. Esse elemento, por sua vez, comunica-se com o Cluster Manager, um componente responsável por distribuir a execução de jobs em Workers disponíveis nos nós do cluster. A principal vantagem desta arquitetura é que todo o processamento dos dados é feito em uma memória compartilhada entre os nós, sendo que o acesso ao disco é evitado sempre que possível para melhorar o desempenho do sistema.

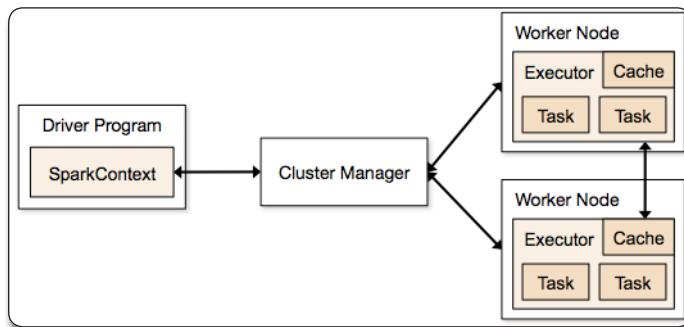


Figura 3. Visão geral da arquitetura do Spark

Além da arquitetura geral, o Spark traz os seguintes conceitos:

- **Resilient Distributed Datasets (RDD):** Abstraem um conjunto de objetos distribuídos no cluster, geralmente executados em memória principal. Esses objetos podem estar armazenados em sistemas de arquivo tradicional, no HDFS e em alguns bancos de dados NoSQL como Cassandra e HBase;

- **Operações:** Representam transformações (como agrupamentos, filtros e mapeamentos entre os dados) ou ações (como contagens e persistências);

- **Contexto Spark (Spark Context):** Como ilustrado anteriormente, representa o principal ponto de entrada ao cluster Spark para uma aplicação que o está utilizando.

Antes de implementar um exemplo com o Spark, é interessante ter uma visão geral de como seu cluster funciona. Como podemos verificar na **Figura 4**, este cluster também trabalha com o paradigma Gerente/Trabalhador. Assim, para cada cluster haverá um Cluster Manager, que pode ser chamado de Spark Master (Mestre), e diversos Spark Workers (Trabalhadores). Em resumo, o Gerente é responsável por registrar os Trabalhadores, garantir que completem seu trabalho e, no caso de uma falha, redirecionar esse trabalho para outro trabalhador. O trabalhador, por sua vez, é responsável por realizar os processamentos da aplicação. Neste caso, podemos ver que os mesmos vão manipular os RDDs e executar as transformações.

Usando o MLlib para filtragem colaborativa em um e-commerce

Para facilitar o entendimento dos conceitos e o uso do MLlib, vamos apresentar agora uma aplicação de recomendação de produtos de um e-commerce hipotético. Esse e-commerce tem os dados de seus produtos e de seus clientes armazenados no Elasticsearch e permite que os clientes avaliem os produtos com uma nota de 0 (que significa o mínimo de recomendação) até 10 (o máximo de recomendação). Assim, o objetivo desse e-commerce é que quando um cliente busque por um certo produto, não apenas outros produtos semelhantes sejam apresentados, mas também produtos que foram recomendados por usuários com gosto similar a esse cliente sejam indicados.

A aplicação desenvolvida nesse exemplo tem como base a Arquitetura Lambda, a fim de organizar seu funcionamento e exemplificar uma aplicação do mundo real. Nesse tipo de arquitetura, certas ações do usuário geram dois fluxos de execução que são processados por duas camadas distintas, conforme ilustrado na **Figura 5**: *Speed Layer*, que trata os dados para processamento em tempo real; e *Batch Layer*, que armazena os dados e analisa os mesmos de forma mais profunda. A decisão de quais operações serão analisadas em cada camada depende do domínio da aplicação, mas de maneira geral, a *Speed Layer* serve para atualizações dos dados que darão ao usuário final uma impressão de execução em tempo real. Por exemplo, se o usuário escreve uma recomendação sobre certo produto, é esperado que essa recomendação esteja disponível imediatamente para outros usuários. Por outro lado, a *Batch Layer* deve ser utilizada para executar ações nas quais o tempo não é fundamental e, normalmente, cujo resultado não será apresentado para um cliente comum. Por exemplo, pode-se citar os cálculos estatísticos, geração de relatórios, agregação de informações e treinamento de mecanismos de aprendizado de máquina.

Assim, o Apache Spark poderia atuar em duas partes da aplicação: o processamento em tempo real e a análise baseada

em aprendizado de máquina. Essa versatilidade facilita o desenvolvimento e o gerenciamento das aplicações de Big Data. Entretanto, neste artigo, vamos focar apenas nos mecanismos de recomendação. Caso tenha interesse, foi publicado na Java Magazine 142 um artigo que explica o desenvolvimento de software em tempo real.

Conhecidos estes conceitos, para o desenvolvimento da aplicação, também será utilizado o Eclipse como IDE e o Apache Maven como gerenciador de pacotes. Para configurar o ambiente de desenvolvimento, a primeira etapa é instalar uma IDE. A partir desta IDE podemos criar um projeto Maven e alterar seu arquivo *pom.xml* conforme apresentado na **Listagem 1**. Como podemos verificar, as dependências do projeto são o Spark 1.4.1 com o MLlib e o Elasticsearch 1.7. O código desse artigo foi implementado no Eclipse Luna, porém qualquer IDE poderia ser utilizada.

O próximo passo é a instalação do Elasticsearch, que consiste simplesmente em baixá-lo do site da Elastic (vide [Links](#)) e descompactá-lo. Mais detalhes sobre esta ferramenta também podem ser encontrados em um recente artigo publicado na Java Magazine 137. Para iniciar o Elasticsearch, acesse a pasta de instalação */bin* e

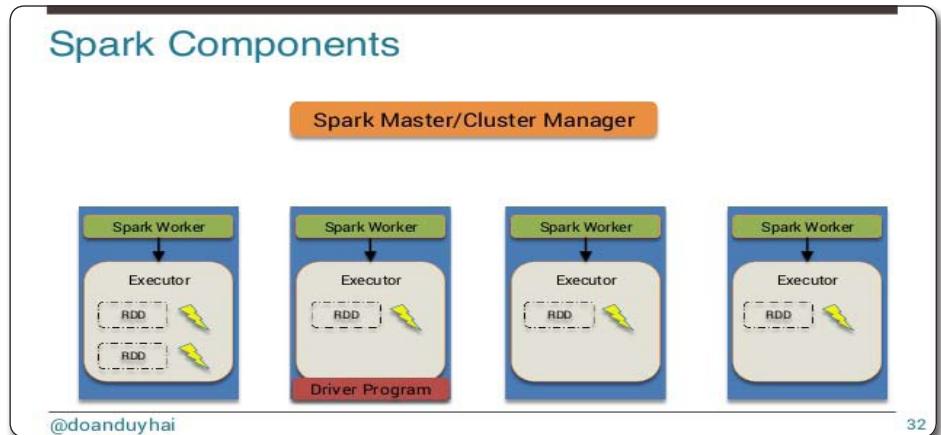


Figura 4. Master e workers no cluster Spark

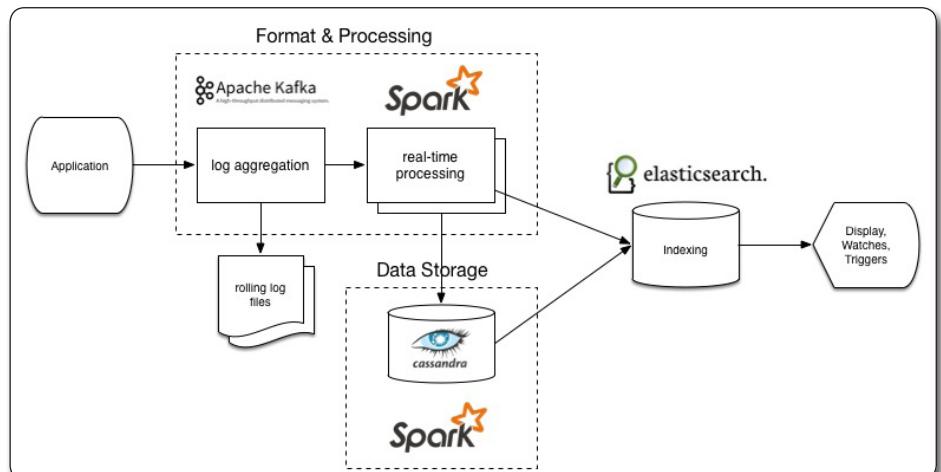


Figura 5. Arquitetura da Aplicação

Listagem 1. Arquivo pom.xml com a configuração de dependências Maven para a aplicação exemplo com Spark.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>com.devmedia</groupId>
<artifactId>machinelearning-spark</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>storm-spark</name>
<url>http://maven.apache.org</url>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
  <dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>1.7.</version>
  </dependency>
  <dependency>
```

```
<groupId>org.apache.spark</groupId>
<artifactId>spark-mllib_2.10</artifactId>
<version>1.4.1</version>
</dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <mainClass>fully.qualified.MainClass</mainClass>
          </manifest>
        </archive>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

Apache Spark: Como criar um mecanismo de sugestão de produtos

execute o script *elasticsearch*. Para facilitar o envio de comandos a ele, recomendamos o uso do plugin *Sense*, no navegador Chrome (veja a seção [Links](#)).

O primeiro passo no Elasticsearch é a criação do índice e dos tipos para armazenar os produtos, os clientes e as recomendações. A **Listagem 2** apresenta a criação do índice *e-commerce* e os tipos *produto*, *usuario* e *recomendacao*. Como podemos verificar, o tipo *recomendacao* tem um relacionamento com *produto* através do campo *_parent*. Obviamente, esse mapeamento foi simplificado ao máximo para facilitar a compreensão. Em um sistema real, deveríamos nos preocupar também com analisadores, filtros e outros aspectos que estão explicados no artigo sobre Elasticsearch, citado anteriormente.

Listagem 2. Criação do índice e dos tipos para produto, cliente e recomendação no Elasticsearch.

```
PUT /ecommerce

POST /ecommerce/_mapping/produto
{
  "produto": {
    "properties": {
      "nome": {
        "type": "string"
      },
      "marca": {
        "type": "string"
      },
      "preco": {
        "type": "double"
      }
    }
  }
}

POST /ecommerce/_mapping/usuario
{
  "usuario": {
    "properties": {
      "nome": {
        "type": "string"
      }
    }
  }
}

POST /ecommerce/_mapping/recomendacao
{
  "recomendacao": {
    "_parent": {
      "type": "produto"
    },
    "properties": {
      "texto": {
        "type": "string"
      },
      "nota": {
        "type": "integer"
      }
    }
  }
}
```

A fim de testar os códigos que serão apresentados a seguir, podemos inserir os mesmos dados apresentados na **Tabela 1**. A **Listagem 3** apresenta como inserir os produtos, os usuários e suas respectivas recomendações no Elasticsearch. Por brevidade, são apresentados somente os dados da usuária Maria.

Listagem 3. Inserção de dados no Elasticsearch.

```
PUT /ecommerce/produto/1
{
  "nome": "Java 8: a bíblia",
  "preco": 100
}

PUT /ecommerce/produto/2
{
  "nome": "Introdução ao Elasticsearch",
  "preco": 80
}

PUT /ecommerce/produto/3
{
  "nome": "Dominando o Apache Spark",
  "preco": 120
}

PUT /ecommerce/produto/4
{
  "nome": "Big Data e seus conceitos",
  "preco": 70
}

PUT /ecommerce/usuario/1
{
  "nome": "Maria"
}

PUT /ecommerce/recomendacao/1?parent=1
{
  "texto": "Adorei esse livro!",
  "nota": 9
}

PUT /ecommerce/recomendacao/1?parent=2
{
  "texto": "Muito bom, completo em conceitos",
  "nota": 8
}

PUT /ecommerce/recomendacao/1?parent=3
{
  "texto": "Melhor livro que já li na vida",
  "nota": 10
}

PUT /ecommerce/recomendacao/1?parent=4
{
  "texto": "Não gostei, muito genérico",
  "nota": 1
}
```

O próximo passo é instalar o Apache Spark. Para isso, baixe o cluster compilado no site do projeto (veja o endereço na seção [Links](#)). Na página de downloads, escolha o release do Spark (atualmente na versão 1.4.1) e o tipo do pacote. Como o Spark pode ser integrado ao Hadoop, o pacote define com qual versão do Hadoop será feita a integração.

Figura 6. Painel de controle do Spark

Visto que neste artigo não temos nada relacionado ao Hadoop, pode ser escolhida qualquer versão. Ainda assim, recomendamos que utilize a última (Hadoop 2.6 or later). Feito isso, descompacte o arquivo baixado e execute: `bin\spark-class.cmd org.apache.spark.deploy.master.Master`, o que irá iniciar o mestre do cluster. Na sequência, acesse `http://localhost:8080/` (vide **Figura 6**) e veja qual é o endereço de rede desse mestre (geralmente `spark://<IP>:7077`). Com o mestre rodando, execute `bin\spark-class.cmd org.apache.spark.deploy.worker.Worker spark://<IP>:7077`, substituindo `<IP>` pelo endereço do mestre. Ao recarregar a página `http://localhost:8080/`, você verá que o primeiro worker foi adicionado.

Com o Elasticsearch e o Spark já configurados, podemos voltar ao Eclipse para criar o código que irá recuperar as informações indexadas no Elasticsearch e alimentar o sistema de filtragem colaborativa no Spark. Portanto, o próximo passo, apresentado na **Listagem 4**, é inicializar o contexto do Spark, que, como dito anteriormente, servirá para enviar as tarefas de aprendizado ao cluster Spark em execução.

O código da **Listagem 5**, que traz a implementação do método `treinar()`, também faz parte da classe `FiltragemColaborativa`. Em termos gerais, ele conecta-se ao Elasticsearch e busca todos os produtos cadastrados. Posteriormente, para cada produto, treina o sistema de recomendação ao recuperar todas as recomendações desse produto e cria um novo modelo para a filtragem colaborativa, que será usado na sequência. Este método deve ser usado em processos executados em background.

Listagem 4. Inicialização do contexto do Spark.

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaDoubleRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.mllib.classification.NaiveBayesModel;
import org.apache.spark.mllib.recommendation.ALS;
import org.apache.spark.mllib.recommendation.MatrixFactorizationModel;
import org.apache.spark.mllib.recommendation.Rating;
import org.apache.spark.mllib.regression.LabeledPoint;

public class FiltragemColaborativa{
    public void inicio(String [] model){
        conf = new SparkConf().setAppName("SVM Classifier")
            .setMaster("spark://luiz-System-Product-Name:7077")
            .setJars(new String[]{"target/smart-crawler-0.0.1-SNAPSHOT.jar"})
            .set("spark.akka.frameSize", "20");
        sc = new JavaSparkContext(conf);
    }
    ...
}
```

Para que o processo de treinamento seja efetivo, é necessário recuperar os dados que estão armazenados no Elasticsearch para que sejam usados de acordo com um item, um usuário e suas notas. Nesse exemplo, a nota vai ser apresentada de forma explícita – já que o usuário diz qual sua nota para certo produto –, porém, poderíamos criar uma aplicação que ponderasse uma

nota, por exemplo, a partir da quantidade de acessos, vendas ou clicks que esse produto obtivesse. Nesse segundo caso, diríamos que o sistema provê uma nota implícita. O código de treinamento e recomendação seria exatamente o mesmo, mas deveríamos informar essa escolha ao algoritmo de treinamento para que o Spark otimize seu trabalho.

Também no código da **Listagem 5**, a chamada do método **train()** da classe **ALS** recebe os seguintes parâmetros:

- **Ratings:** Define as notas inseridas nos dados de treinamento, nesse caso esses valores estão armazenados no RDD;
- **Rank:** Define a escala de medição nas notas;
- **Iterations:** Especifica o número de iterações a serem executadas para o treinamento da rede;
- **Lambda:** Especifica o parâmetro de regularização do método de treinamento para evitar uma situação de *overfitting*, ou seja, que o modelo criado seja tão especializado que não seja capaz de prever os resultados para novos dados;
- **NumBlocks:** Representa o número de blocos usados para parallelizar a computação do mecanismo de recomendação (definido como -1 para configuração geral);
- **Alfa:** Parâmetro aplicável para a variante de feedback implícita que define a confiança nas observações de preferência (não está representado na **Listagem 5**, pois estamos usando feedback explícito).

Ainda sobre o código do método **treinar()**, podemos ver que o Spark utiliza o método de fatoração ou decomposição de matrizes chamado *Alternating Least Squares* (na classe **ALS**). Esse método permite que a fatoração – uma tarefa que está no núcleo da Filtragem Colaborativa – seja otimizada através de uma solução algébrica mais fácil e, por isso, mais escalável. Esta abordagem é discutida no artigo “Large-Scale Parallel Collaborative Filtering for the Netflix Prize” (vide **Links**) e faz o método menos dependente do tamanho do conjunto de dados de treinamento. Assim, pode treinar o modelo a partir de um subconjunto amostral (ou seja, em um conjunto pequeno de dados e por isso mais rapidamente) e aplicá-lo ao conjunto de dados completo.

Ao final de **treinar()** temos a criação do modelo de predição, modelo que pode ser armazenado em disco, conforme o código **model.save()**, que por padrão salva o arquivo na pasta na raiz do projeto. No entanto, esse modelo não necessariamente precisa ser persistido. Caso a aplicação tenha um modelo mais volátil (ou seja, que muda muito ao longo do tempo), o mesmo pode permanecer apenas na memória principal.

Obviamente, o sistema deve realizar a atividade de treinamento com alguma frequência, pois caso contrário o modelo de recomendação já não será mais fiel ao estado atual da opinião de seus usuários. Por outro lado, o treinamento é uma atividade extremamente cara do ponto de vista computacional, especialmente para conjuntos de dados grandes. Assim, justifica-se sua inclusão na camada Batch da arquitetura.

Durante o treinamento, um mecanismo interessante provido pelo Spark para otimização dos resultados é a avaliação de erro.

Listagem 5. Treinamento do mecanismo de filtragem colaborativa.

```
public void treinar() {  
    // Load and parse the data  
    String path = "data/mllib/als/test.data";  
    JavaRDD<String> data = sc.textFile(path);  
    JavaRDD<Rating> ratings = data.map(  
        new Function<String, Rating>() {  
            public Rating call(String s) {  
                String[] sarray = s.split(",");  
                return new Rating(Integer.parseInt(sarray[0]), Integer.parseInt(sarray[1]),  
                    Double.parseDouble(sarray[2]));  
            }  
        }  
    );  
    logger.info("Mean Squared Error = " + evaluate(ratings));  
    // Treinar o modelo de recomendação  
    int rank = 10;  
    int numIterations = 20;  
    int numBlocks= 10;  
    double lambda= 0.01;  
    MatrixFactorizationModel model = ALS.train(JavaRDD.toRDD(ratings),  
        rank, numIterations, lambda, numBlocks);  
    // Save and load model  
    model.save(sc.sc(), "novoModelo");  
    sc.close();  
}
```

Conforme ilustrado na **Listagem 6**, com a classe **MSE**, podemos usar o erro quadrático médio (do inglês *Mean Squared Error*) para verificar a capacidade de predição do modelo – quanto menor o erro, melhor será essa capacidade. Com base nesse erro, podemos tomar decisões como refazer o treinamento com mais dados, combinar a técnica de filtragem com outros métodos de predição, ou, ao menos, informar os responsáveis pelo sistema a fim de que os mesmos tomem uma atitude. Como mostrado no código, essa avaliação é feita através da comparação (ou junção, do inglês *join*) entre a predição (do inglês *predictions*) e o modelo criado.

Com o modelo criado, já podemos recomendar informações para nossos usuários, como ilustrado na **Listagem 7**. Como podemos verificar, o código para essa operação é simplesmente uma leitura do modelo salvo previamente. Note que ambos os métodos, **recomendarProduto()** e **recomendarUsuarios()**, usam o método **load()** do objeto **MatrixFactorizationModel** para carregar o modelo de predição.

O método **recomendarProdutos()** recebe o código do usuário e retorna uma lista dos produtos que devem ser interessantes para ele. Esses produtos são retornados no formato da classe **Rating**, que contém, entre outras informações, o identificador dos produtos recomendados e o rating, que é um valor numérico que define o quanto esse produto é relevante para o usuário. A partir do identificador é possível recuperar o nome do produto no Elasticsearch. Essa mesma ideia pode ser aplicada se quisermos recuperar uma série de usuários com base em um certo produto. Tal funcionalidade seria interessante para, por exemplo, escolher quais usuários receberão um e-mail com a lista desses produtos em promoção.

Listagem 6. Avaliação do mecanismo de filtragem colaborativa

```
public double avaliar(JavaRDD<Rating> ratings){  
  
    // Evaluate the model on rating data  
    JavaRDD<Tuple2<Object, Object>> userProducts = ratings.map(  
        new Function<Rating, Tuple2<Object, Object>>() {  
            public Tuple2<Object, Object> call(Rating r) {  
                return new Tuple2<Object, Object>(r.user(), r.product());  
            }  
        }  
    );  
  
    JavaPairRDD<Tuple2<Integer, Integer>, Double>  
    predictions = JavaPairRDD.fromJavaRDD(model.predict(JavaRDD.toRDD(userProducts)).toJavaRDD().map(  
        new Function<Rating, Tuple2<Integer, Integer>>() {  
            public Tuple2<Integer, Integer>, Double> call(Rating r){  
                return new Tuple2<Integer, Integer>(r.user(), r.rating());  
            }  
        }  
    ));  
  
    JavaRDD<Tuple2<Double, Double>> ratesAndPreds =  
    JavaPairRDD.fromJavaRDD(ratings.map(  
        new Function<Rating, Tuple2<Integer, Integer>, Double>>() {  
            public Tuple2<Integer, Integer>, Double> call(Rating r){  
                return new Tuple2<Integer, Integer>(r.user(), r.rating());  
            }  
        }  
    )).join(predictions).values();  
  
    double MSE = JavaDoubleRDD.fromRDD(ratesAndPreds.map(  
        new Function<Tuple2<Double, Double>, Object>() {  
            public Object call(Tuple2<Double, Double> pair) {  
                Double err = pair._1() - pair._2();  
                return err * err;  
            }  
        }  
    ).rdd()).mean();  
  
    return MSE;  
}
```

Listagem 7. Recomendação de produtos.

```
public void recomendarProdutos(int usuario){  
    Index index = new Index();  
    model = MatrixFactorizationModel.load(sc.sc(), "novoModelo");  
    Rating[] ratings = model.recommendProducts(usuario, 10);  
  
    for(Rating rating: ratings){  
        index.getProduto(rating.product());  
        logger.info(String.valueOf(rating.rating()));  
    }  
}  
  
public void recomendarUsuarios(int produto){  
    Index index = new Index();  
    model = MatrixFactorizationModel.load(sc.sc(), "novoModelo");  
    Rating[] ratings = model.recommendUsers(produto, 10);  
  
    for(Rating rating: ratings){  
        index.getUser(rating.user());  
        logger.info(String.valueOf(rating.rating()));  
    }  
}
```

Para ilustrar o funcionamento do nosso sistema de recomendação, podemos fazer um pequeno teste com o JUnit (vide [Listagem 8](#)) para avaliar os resultados esperados. Os métodos `testTreinar()` e `testAvaliar()` demonstram, respectivamente, o treinamento e a avaliação do modelo de sugestão. Por sua vez, os métodos `testRecomendarProduto()` e `testRecomendarUsuarios()` fazem uso do modelo criado em `testTreinar()` para recomendar produtos e usuários de acordo com a informação de entrada.

Listagem 8. Recomendação de produtos.

```
public class CFImplTest extends TestCase {  
  
    CFImpl cf = new CFImpl();  
  
    Index index = new Index();  
    List<Rating> ratings = index.getRatings();  
  
    public void testTreinar() {  
        cf.treinar(ratings);  
    }  
  
    public void testAvaliar() {  
        cf.avaliar(ratings);  
    }  
  
    public void testRecomendarProduto() {  
        cf.recomendarProduto(1);  
    }  
  
    public void testRecomendarUsuarios() {  
        cf.recomendarUsuarios(1);  
    }  
}
```

Se usarmos os dados do exemplo apresentado na seção de introdução, a resposta do JUnit, conforme esperado, será o produto “Introdução ao Elasticsearch” ao usuário 1, enquanto ao produto 2 será apresentado o usuário “Maria”.

Ainda que não seja o foco do artigo, a [Listagem 9](#) apresenta o código da classe `Index`, classe que se conecta e recupera as informações do Elasticsearch. Para isso, o método `getClient()` faz a conexão com o Elasticsearch, utilizando o padrão de projeto `Singleton`; `getProduto()` e `getUser()` retornam os valores para um produto ou usuário baseando-se em seu id; `getRatings()` retorna todos os ratings armazenados no Elasticsearch; e `query()` é um método privado que efetiva a busca no Elasticsearch.

Nos últimos anos, o aprendizado de máquina está ganhando cada vez mais força como um método para facilitar o acesso à cada vez maior quantidade de informações armazenada pelos ambientes computacionais atuais. Sinais dessa tendência são a recente abertura em Paris do Laboratório de Inteligência Artificial do Facebook, e a contribuição de grandes players como LinkedIn e Twitter para bibliotecas *open source* de aprendizado de máquina (veja mais informações na seção [Links](#)).

Nesse contexto, o Apache Spark é parte de um movimento maior que tem como objetivo permitir análises mais complexas sobre os dados da Big Data.

Apache Spark: Como criar um mecanismo de sugestão de produtos

Listagem 9. Acesso ao índice do Elasticsearch.

```
public class Index {  
  
    private static Client client;  
  
    public static synchronized Client getClient() throws Exception {  
  
        if (client == null) {  
            TransportClient transportClient;  
  
            Settings settings = ImmutableSettings.settingsBuilder()  
                .put("cluster.name", "ecommerce").build();  
            transportClient = new TransportClient(settings);  
            client = transportClient  
                .addTransportAddress(new InetSocketAddress("localhost",  
                    9200));  
        }  
  
        return client;  
    }  
  
    public String getProduto(int id){  
        GetRequestBuilder request = client  
            .prepareGet()  
            .setIndex("ecommerce")  
            .setType("produto")  
            .setId(String.valueOf(id));  
  
        return request.get().getSource().get("nome").toString();  
    }  
  
    public String getUser(int id) throws Exception {  
        GetRequestBuilder request = client  
            .prepareGet()  
            .setIndex("ecommerce")  
            .setType("user")  
            .setID(String.valueOf(id));  
  
        return request.get().getSource().get("nome").toString();  
    }  
}
```

```
public List<Rating> getRatings() throws ElasticsearchException, Exception {  
  
    MatchAllQueryBuilder qb = QueryBuilders.matchAllQuery();  
  
    Set<SearchHit> hits = query(qb, 0, 1000);  
  
    List<Rating> ratings = new ArrayList<Rating>();  
  
    for(SearchHit hit: hits){  
        ratings.add(new Rating(Integer.valueOf(hit.getSource().get("user").toString()),  
            Integer.valueOf(hit.getSource().get("product").toString()),  
            Double.valueOf(hit.getSource().get("rating").toString())));  
    }  
  
    return ratings;  
}  
  
private Set<SearchHit> query(QueryBuilder qb, int from, int size)  
throws ElasticsearchException, Exception{  
    SearchResponse scrollResp = GeneralElasticsearch.getClient()  
        .prepareSearch("ecommerce")  
        .setSearchType(SearchType.SCAN).setScroll(new TimeValue(60000))  
        .setQuery(qb).setFrom(from).setSize(size).execute().actionGet();  
  
    Set<SearchHit> response = new HashSet<SearchHit>();  
  
    while (true) {  
        response.addAll(Arrays.asList(scrollResp.getHits().getHits()));  
  
        scrollResp = GeneralElasticsearch.getClient()  
            .prepareSearchScroll(scrollResp.getScrollId())  
            .setScroll(new TimeValue(600000)).execute().actionGet();  
  
        if (scrollResp.getHits().getHits().length == 0) {  
            break;  
        }  
    }  
  
    return response;  
}
```

Conhecimento faz diferença!



The image shows three issues of the 'engenharia de software' magazine. The first issue, 'Gerência de Configuração', has a green and yellow cover with a person working on puzzle pieces. The second issue, 'Evolução do Software', has a blue cover with a hand holding a small plant. The third issue, 'Automação de Testes', has a dark cover featuring a black robot. A large red starburst graphic on the right side of the 'Automação de Testes' cover contains the text '+ de 290 vídeos para assinantes'.

Faça já sua assinatura digital! | www.devmedia.com.br/es

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!**



DEV MEDIA

Assim, o Spark possui o mesmo objetivo de outros frameworks desenvolvidos recentemente, e apesar dos autores o entenderem como a melhor escolha, é importante conhecer outras soluções. Do ponto de vista do processamento em tempo real, os frameworks com propostas similares são o Apache Kafka, desenvolvido e usado pelo LinkedIn, o Apache Flume, usado pelo SoundCloud, e o Apache Storm, desenvolvido pelo Twitter. Por outro lado, o Apache Mahout é uma alternativa para os mecanismos de aprendizagem de máquina.

Em relação a essas opções, a principal vantagem do Spark é prover uma solução que engloba muitos aspectos e necessidades de Big Data em apenas um mecanismo. Além disso, o Spark está recebendo contínuos sinais de apoio de empresas como IBM e Databricks, o que garante um perene interesse e suporte para seu desenvolvimento. Vale acrescentar que o Spark é apoiado pela Fundação Apache. Deste modo, podemos confiar na qualidade do seu projeto e, principalmente, nos assegurarmos que seremos apoiados pela comunidade e teremos acesso ao código fonte.

Para os leitores interessados em utilizar o Spark como parte de uma solução, fica a sugestão de conhecer mais sobre a arquitetura Lambda – proposta por Nathan Marz (também criador do Apache Storm) –, que organiza em uma única visão o processamento em tempo real com o processamento batch, e o Elasticsearch, que sem dúvida provê uma maneira ótima de armazenar os resultados do processamento feito no Spark.

Uma forma interessante de expandir a aplicação proposta nesse artigo é através do uso de métodos híbridos na recomendação, que permite combinar tanto estratégias de recomendação baseadas em conteúdo quanto estratégias baseadas em colaboração. Um método híbrido aumentaria a capacidade de recomendação da aplicação e melhoraria os resultados da Filtragem Colaborativa. Para tal, fica a cargo do leitor expandir o exemplo apresentado usando técnicas como SVM, Naive Bayes e cálculos estatísticos, também presentes no MLlib.

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Autor



Luiz Henrique Zambom Santana

lhzsantana@gmail.com

É bacharel e mestre em Ciência da Computação. Atualmente cursa doutorado também em Ciência da Computação na UFSC. Possui mais de 10 anos de experiência em programação Java e há dois anos atua com tecnologias de Big Data. Trabalhou em projetos para grandes empresas no Brasil, Argentina e Alemanha. Atualmente é consultor de Elasticsearch e startupper na 33!.



Autor



Eduardo Felipe Zambom Santana

ezambomsantana@gmail.com

É bacharel e mestre em Ciência da Computação pela UFSCar. Possui mais de 10 anos de experiência em programação. Atualmente é aluno de doutorado na USP e é professor na Universidade Anhembi Morumbi.



Links:

Endereço para download do Elasticsearch.

<https://www.elastic.co/downloads/elasticsearch>

Endereço para download do Spark.

<http://spark.apache.org/downloads.html>

GitHub do Apache Spark.

<https://github.com/apache/spark>

GitHub do autor, com os códigos apresentados no artigo.

<https://github.com/lhzsantana/mllib-javamagazine>

Facebook abrirá laboratório de inteligência artificial na França.

<http://info.abril.com.br/noticias/internet/2015/06/facebook-abrira-laboratorio-de-inteligencia-artificial-em-paris.shtml>

Zhou, Yunhong, et al. "Large-scale parallel collaborative filtering for the netflix prize." *Algorithmic Aspects in Information and Management*. Springer Berlin Heidelberg, 2008. 337-348.

Introdução ao Spring Batch – Parte 2

Aprenda neste artigo como processar itens em lotes na prática

ESTE ARTIGO FAZ PARTE DE UM CURSO

No primeiro artigo da série estudamos todos os principais conceitos de sistemas de processamento em lotes e a forma como o Spring Batch os define ao longo de sua API. Sugerimos, também, um contexto de aplicação para que pudéssemos desenvolver um projeto prático que explorasse os elementos essenciais deste framework.

Neste texto, discutiremos a composição do único job deste projeto sob uma ótica inteiramente prática. A partir da análise do código-fonte e alguns arquivos de configuração, entenderemos os detalhes do comportamento de *item readers*, *item processors*, *item writers* e, principalmente, o quanto simples é configurá-los e customizá-los utilizando a API do Spring Batch.

Ao final do artigo, teremos um projeto pronto e funcional, que nos permitirá exercitá-lo com segurança e base conceitual todo o conteúdo estudado ao longo da série de artigos sobre o framework. A partir de tudo o que teremos visto, o leitor estará em plenas condições de ir além, passando a explorar os elementos e as API mais específicas e complexas desta poderosa biblioteca.

Formatos e origens de dados

Para este tutorial, convidamos o leitor a montar conhecimento do cenário ilustrado na **Figura 1**. Agora, imagine um parque de infância repleto de livros. Todos esses livros estão, neste nosso ambiente hipotético, cadastrados manualmente a partir de planilhas. Como este trabalho é bastante sujeito a falhas e, em um futuro breve, há planos de inaugurar um serviço de empréstimo de exemplares para que as crianças possam levá-los para casa, a equipe diretora do estabelecimento começou a se mobilizar no desenho de um novo sistema de gerenciamento de todo o acervo.

Fique por dentro

Este artigo é útil por apresentar na prática como processar grandes volumes de dados com uma tecnologia de referência: o Spring Batch. Para isso, a partir de uma análise detalhada do código-fonte e da configuração dos principais componentes do projeto, identificaremos tudo aquilo que é necessário para que a leitura, o processamento e escrita de itens funcione corretamente dentro de uma aplicação desta natureza. Deste modo, o leitor estará apto a começar o desenvolvimento de suas próprias soluções, com uma base suficientemente boa tanto no aspecto teórico quanto no prático.

Entretanto, uma das grandes preocupações que existem é que todos os registros já mantidos em planilhas possam ser aproveitados; caso isso não fosse possível, haveria um grande retrabalho necessário para, a partir do novo sistema, cadastrar todas as obras novamente. Além disso, já sabemos, de antemão, que o novo sistema será baseado em um banco de dados relacional, por entender-se que esta é a maneira ideal de representar e manter os registros dali em diante.

Imaginou? Este é um cenário típico em que um ETL se encaixaria perfeitamente, pelos seguintes motivos:

1. Há uma fonte de entrada de dados bem definida (planilhas);
2. Há uma fonte ‘final’ de dados também muito bem definida (banco de dados);
3. Há algum processamento a ser realizado na interpretação dos registros de origem e, posteriormente, conversão para um novo modelo de representação.

A **Figura 2** ilustra a planilha que utilizaremos ao longo deste tutorial. Para podermos aproveitar todo o seu conteúdo, exportá-la-emos para um arquivo CSV. Este é um recurso, inclusive, comum em programas gerenciadores de planilhas como Excel, e a conversão para CSV é importante por facilitar significativamente a manipulação de seus registros.

O resultado deste processo de extração pode ser visto na **Listagem 1** e, por ela, aprendemos que todo livro é representado pelos seguintes dados:

- Título;
- Subtítulo;
- Nome do autor;
- Número de páginas;
- Nome da editora.

Listagem 1. Registros de livros no formato CSV.

```
Livro A,Primeiro livro,Pedro,120,Editora 123
Livro B,Segundo livro,Lucas,300,Editora 123
Livro C,Terceiro livro ,João ,140,Editora 123
Livro D,Quarto livro,Mateus,250,Editora 123
```

Entretanto, esta não é a forma ideal de representar um livro; trata-se, apenas, de um recurso que podemos utilizar para aproveitar todos os registros já criados para, enfim, carregar o banco de dados a ser usado pelo novo sistema.

A partir desses registros, por exemplo, podemos ver que os dados de editora são frequentemente repetidos. Além disso, como o processo de escrita de registros em planilha é completamente manual, há uma grande chance de se introduzir erros provenientes de digitação, o que poderia resultar em registros inconsistentes. Com o uso de sistemas de bancos de dados relacionais, entretanto, conseguimos eliminar este problema a partir do uso de *constraints* – como chaves primárias e estrangeiras, garantindo a unidade de registros e relacionando os dados entre si para representar o domínio da aplicação.

A **Tabela 1** traz um resumo de como ficou a nossa análise de dados a partir da estrutura da planilha apresentada na **Figura 2**. Nela, todas as principais entidades – que serão convertidas, no banco de dados, em tabelas – estão devidamente mapeadas. Como o leitor pode perceber, alguns desses campos não estão presentes nos registros CSV e serão introduzidos apenas para permitir que a representação dos dados seja mais completa.

A extração de itens

Agora que já entendemos os formatos e a origem dos dados, estudaremos como devemos proceder para escrever a nossa lógica de extração. Afinal de contas, como configuramos um leitor de itens usando Spring Batch?

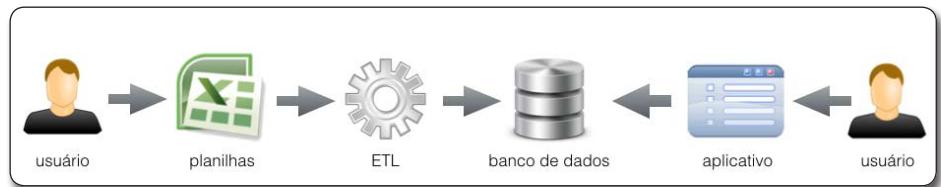


Figura 1. Planejamento e escopo do ETL

DevMedia-SpringBatchTutorial					
	A	B	C	D	E
1	Título	Subtítulo	Autor	Páginas	Editora
2	Livro A	Primeiro livro	Pedro	120	Editora 123
3	Livro B	Segundo livro	Lucas	300	Editora 123
4	Livro C	Terceiro livro	João	140	Editora 123
5	Livro D	Quarto livro	Mateus	250	Editora 123
6					

Figura 2. Foto da planilha utilizada para gerenciamento de registros de uma biblioteca

Entidade	Dados relevantes
Livro	ID, Título, Subtítulo, Número de páginas, ISBN
Autor	ID, Nome, Sobrenome
Editora	Código, Nome, Descrição

Tabela 1. Mapeamento do modelo de dados

Comecemos retornando à **Listagem 2**. Nossa *item reader* está configurado a partir de um bean identificado como *itemReader*. Este objeto corresponde a um tipo denominado `br.com.devmedia.articles.spring.batch.etl.extractors.CsvExtractor`, cujo código-fonte pode ser visto na **Listagem 3**.

Criamos esta classe como uma especialização de uma das implementações básicas de *item reader* do Spring Batch: `org.springframework.batch.item.file.FlatFileItemReader`.

Esta classe já contém todo o comportamento de leitura de arquivos ‘flat’ pré-definido. Ao mesmo tempo, é altamente configurável, permitindo que se sobrecreva praticamente todos os componentes com os quais se relaciona. Pela **Listagem 3**, observamos que nosso *item reader* sobrecreve os seguintes comportamentos e características:

- A localização do arquivo de entrada de dados, a partir de uma propriedade chamada **resource**;

- Um objeto usado para mapear o conteúdo das linhas do arquivo;
- Um tipo de objeto utilizado para encapsular cada registro do arquivo.

Há, naturalmente, inúmeras outras propriedades que podem ser customizadas em um item reader, mas nossa aplicação exigirá a configuração apenas destes três. Para mais informações sobre item readers e todos os seus atributos, verifique a documentação oficial (veja o endereço na seção **Links**).

O mapeamento das linhas de um arquivo

Todo item reader possui um atributo denominado **lineMapper**, que é um tipo de objeto associado, sempre, a outros dois. São eles:

1. Um identificador de tokens, linha a linha do arquivo;
2. Um *wrapper*, que faz a associação de cada um dos tokens identificados em uma linha do arquivo com um atributo do objeto que, por fim, representará o registro.

Introdução ao Spring Batch – Parte 2

Em nosso ETL, de acordo com a **Listagem 2**, vemos que um objeto do tipo **org.springframework.batch.item.file.mapping.DefaultLineMapper** foi configurado para este propósito, e identificado no descriptor da listagem como **lineMapper**.

O elemento que identifica os tokens de uma linha do arquivo CSV, de acordo com a **Listagem 2**, é um objeto do tipo **org.spring-**

framework.batch.item.file.mapping.DelimitedLineTokenizer. Observe que, em sua declaração, definimos o caractere que separa cada token, em cada linha, e também os nomes dos campos a serem usados para guardar cada token identificado. Os nomes definidos para os campos seguem uma regra muito importante, à qual devemos estar sempre atentos: os campos citados devem corresponder

Listagem 2. etlapp-context.xml – Configuração do módulo de processamento em lotes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress SpringFacetInspection -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:batch="http://www.springframework.org/schema/batch"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/batch
                           http://www.springframework.org/schema/batch/spring-batch.xsd
                           http://www.springframework.org/schema/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd">

    <!-- Importando definições de objetos de acesso a banco de dados e serviços
        associados à persistência de objetos -->
    <import resource="classpath:etl_persistence-context.xml"/>

    <!-- Arquivo de configuração do Spring Batch -->

    <!-- Leitura de dados a partir de um arquivo CSV -->
    <bean id="csvRecord" scope="prototype"
          class="br.com.devmedia.articles.spring.batch.etl.model.CsvRecord"/>

    <bean id="itemReader"
          class="br.com.devmedia.articles.spring.batch.etl.extractors.CsvExtractor">
        <property name="resource" value="file:${csv.location}"/>
        <property name="lineMapper">
            <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
                <property name="lineTokenizer">
                    <bean class="org.springframework.batch.item.file.transform.
                               DelimitedLineTokenizer">
                        <property name="names" value="bookTitle, bookSubTitle, authorName,
                               number_of_Pages, publishingHouseName"/>
                        <property name="delimiter" value=","/>
                    </bean>
                </property>
                <property name="fieldSetMapper">
                    <bean class="org.springframework.batch.item.file.mapping.
                               BeanWrapperFieldSetMapper">
                        <property name="prototypeBeanName" value="csvRecord"/>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>

```

</property>

```
</bean>

    <!-- Processador responsável pela transformação de objetos CSV em objetos Book -->
    <bean id="bookProcessor" class="br.com.devmedia.articles.spring.batch.etl.
        transformers.Processor"/>

    <!-- Carregamento de dados de livros em um banco de dados relacional (MySQL) -->
    <bean id="bookWriter" class="br.com.devmedia.articles.spring.batch.etl.loaders.
        MySQLLoader">
        <property name="service" ref="bookService" />
    </bean>

    <bean id="jobRepository"
          class="org.springframework.batch.core.repository.support.JobRepositoryFactoryBean">
        <property name="dataSource" ref="datasource"/>
        <property name="transactionManager" ref="transactionManager" />
        <property name="databaseType" value="mysql" />
    </bean>

    <!-- Configuração do Job propriamente dito -->
    <batch:job id="devmedia_job" restartable="false">
        <batch:step id="singlestep" >
            <batch:tasklet >
                <batch:chunk reader="itemReader" processor="bookProcessor"
                           writer="bookWriter" commit-interval="1" />
            </batch:tasklet>
        </batch:step>
    </batch:job>

    <bean id="jobLauncher"
          class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
        <property name="jobRepository" ref="jobRepository" />
    </bean>

    <!-- Scripts utilizados para configurar o banco de dados automaticamente com as
        tabelas relacionadas ao Batch -->
    <jdbc:initialize-database data-source="datasource">
        <jdbc:script location="org/springframework/batch/core/schema-drop-mysql.sql"/>
        <jdbc:script location="org/springframework/batch/core/schema-mysql.sql"/>
    </jdbc:initialize-database>
</beans>
```

Listagem 3. CsvExtractor – Extrator de dados de um arquivo CSV.

```
package br.com.devmedia.articles.spring.batch.etl.extractors;

import br.com.devmedia.articles.spring.batch.etl.model.CsvRecord;
import org.springframework.batch.item.file.FlatFileItemReader;

import java.util.logging.Logger;

/**
 * Lê uma linha de um arquivo CSV, referente a um livro, e constrói uma representação do
 * mesmo a partir de um objeto CsvRecord. Esta classe, na verdade, só foi criada para
 * demonstrar o comportamento de um item reader. Ela não seria necessária,
 * até porque é possível vermos que o que fazemos é bem pouco:
 * <ul>
 *   <li>Lemos um registro a partir do método doRead() da API do
 *   FlatFileItemReader</li>
 *   <li>Imprimimos o conteúdo deste objeto para visualizar o fluxo da operação
 * </ul>

```

```
*      a partir de arquivos de log.</li>
* </ul>
*/
public class CsvExtractor extends FlatFileItemReader<CsvRecord> {

    private static Logger LOGGER = Logger.getLogger("CsvExtractor");

    public CsvRecord read() throws Exception {
        CsvRecord record = doRead();
        LOGGER.info(String.format(
            "Reading book with title {} from author {}",
            record.getBookTitle(), record.getAuthorName()));
        return record;
    }
}
```

exatamente aos nomes dos atributos do tipo de objeto que será usado para representar cada registro deste arquivo.

O componente que, por fim, mapeia todos os tokens identificados, linha a linha, está definido a partir do atributo `fieldMapper` de nosso `DefaultLineMapper`. Utilizamos, no caso, o tipo `org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper`. Perceba, em sua configuração, que uma propriedade denominada `prototypeBeanName` foi utilizada para fazer referência a um bean identificado como `csvRecord`. Ao olharmos para a definição deste bean na [Listagem 2](#), vemos que se trata de um objeto do tipo `CsvRecord`. Este é o tipo de objeto que representará cada linha do arquivo CSV. Exibimos seu conteúdo na [Listagem 4](#) para que o leitor possa verificar que, como esperado, o nome dos atributos corresponde exatamente aos campos empregados na configuração do objeto identificado como `fieldMapper`.

Desta forma, o job terá o que precisa para, internamente, fazer a associação entre os tokens identificados e cada atributo da classe que deverá representar cada linha do arquivo de entrada de dados.

Toda a interpretação e a representação de cada linha do arquivo CSV ocorre exatamente quando o *item reader* é instruído a realizar uma leitura. Isto acontece a partir da chamada, pelo chunk, ao método `read()` da API da interface `ItemReader` do Spring Batch, implementado por `CsvExtractor`, e que podemos ver a partir da [Listagem 3](#).

Este método `read()` é automaticamente invocado pelo chunk. Quando isto ocorrer, realizamos uma chamada ao método `doRead()`, definido na classe `org.springframework.batch.item.file.FlatFileItemReader`, que se encarrega de ler o próximo item no arquivo de entrada.

Neste instante, todos os mapeamentos ocorrem nos bastidores (lembre-se, aqui, do *line mapper* e do *bean wrapper* discutidos há pouco) e o resultado final será um objeto do tipo `CsvRecord`. Quando este objeto estiver pronto, é retornado pelo item reader ao chunk. A cada vez que isso acontecer, o chunk verificará se o número de registros acumulado por ele já atingiu o valor definido em seu atributo `commit-interval`. Sempre que isso ocorrer, todos os objetos acumulados serão enviados para o próximo componente da cadeia que, neste caso, será um *item writer* representado pela classe `Processor`.

A transformação de itens

Todo processador escrito a partir da API do Spring Batch implementa, em última instância, a interface `org.springframework.batch.item.ItemProcessor`. Neste tópico, discutiremos um pouco as suas características e a classe, dentro de nosso ETL, responsável pelo processamento de registros CSV referentes a livros.

Comecemos pela observação do conteúdo da [Listagem 5](#). A classe `Processor`, nesta listagem, contém o código que escrevemos para, a partir de objetos do tipo `CsvRecord`, construirmos

Listagem 4. CsvRecord – representação na forma de objeto de cada linha do arquivo CSV.

```
package br.com.devmedia.articles.spring.batch.etl.model;

import java.io.Serializable;

public class CsvRecord implements Serializable {

    private String bookTitle;
    private String bookSubTitle;
    private String authorName;
    private Integer numberofPages;
    private String publishingHouseName;

    public CsvRecord(){}

    public CsvRecord(String bookTitle, String bookSubTitle, String authorName,
    Integer numberofPages, String publishingHouseName) {
        this();
        this.bookTitle = bookTitle;
        this.bookSubTitle = bookSubTitle;
        this.authorName = authorName;
        this.numberofPages = numberofPages;
        this.publishingHouseName = publishingHouseName;
    }

    public String getBookTitle() {
        return bookTitle;
    }

    public void setBookTitle(String bookTitle) {
        this.bookTitle = bookTitle;
    }

    public String getBookSubTitle() {
        return bookSubTitle;
    }

    public void setBookSubTitle(String bookSubTitle) {
        this.bookSubTitle = bookSubTitle;
    }

    public String getAuthorName() {
        return authorName;
    }

    public void setAuthorName(String authorName) {
        this.authorName = authorName;
    }

    public Integer getNumberofPages() {
        return numberofPages;
    }

    public void setNumberofPages(Integer numberofPages) {
        this.numberofPages = numberofPages;
    }

    public String getPublishingHouseName() {
        return publishingHouseName;
    }

    public void setPublishingHouseName(String publishingHouseName) {
        this.publishingHouseName = publishingHouseName;
    }
}
```

Introdução ao Spring Batch – Parte 2

uma representação virtual de um livro. Usaremos, para este mapeamento, uma classe que criamos no módulo model, intitulada **Book**.

A interface **ItemProcessor**, implementada por nossa classe **Processor**, possui apenas um método, cuja assinatura é:

```
O process(I var1) throws Exception;
```

No caso de nossa classe **Processor**, o parâmetro **I** será vinculado ao tipo de objetos **CsvRecord** (construído pelo **CsvExtractor**), enquanto o parâmetro **O** será associado ao tipo **Book**.

Como dito no início do artigo, este é o momento em que uma série de transformações, verificações e validações pode – e deve – ser realizada. Começamos o texto listando algumas delas, e o propósito é apenas estimular o leitor a não apenas baixar o código-fonte que produzimos, mas também alterá-lo e experimentá-lo trabalhando nessas pequenas sugestões que lançamos.

Para relembrar, estes foram os pontos que introduzimos e convidamos o leitor para olhar:

- A mesma editora e o mesmo autor podem ser citados por vários registros, mas o banco de dados deve garantir que haverá somente um registro para cada autor e, da mesma forma, somente um registro para cada editora;
- Além da possibilidade de duplicação de registros, levantada no ponto anterior, há também uma chance de, em arquivos com muitos registros, haver um mesmo autor ou uma mesma editora digitados de forma diferente.

No código original da classe **Processor**, exibido na **Listagem 5**, implementamos apenas a conversão de objetos **CsvRecord** em outros do tipo **Book**.

A escrita de itens

Assim que os objetos recuperados de um arquivo CSV forem processados, estarão disponíveis no chunk para serem, enfim, salvos no banco de dados de destino.

O mecanismo de persistência em nosso ETL está ilustrado na **Figura 3**. Nele, vemos uma classe intitulada **MySQLLoader**, que representa o item writer propriamente dito. O item writer, por sua vez, se associa a um objeto do tipo **BookService**, que é o responsável direto pela coordenação dos objetos DAO para manipulação de registros junto ao banco de dados.

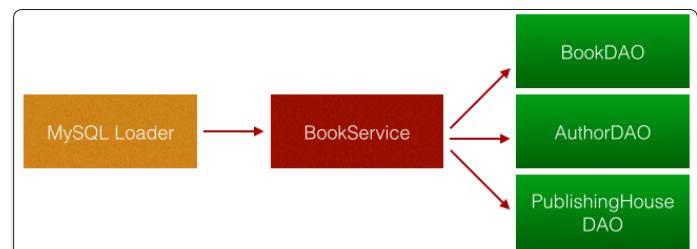


Figura 3. Mecanismo de escrita de livros no banco de dados da solução

Estamos, portanto, escrevendo a nossa própria lógica de gerenciamento de transações. Poderíamos, é verdade, utilizar recursos do próprio Spring Batch para resolver o problema.

Listagem 5. Processor – Transformador de registros CSV em objetos de negócio.

```
package br.com.devmedia.articles.spring.batch.etl.transformers;

import br.com.devmedia.articles.spring.batch.etl.model.Author;
import br.com.devmedia.articles.spring.batch.etl.model.Book;
import br.com.devmedia.articles.spring.batch.etl.model.CsvRecord;
import br.com.devmedia.articles.spring.batch.etl.model.PublishingHouse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.ItemProcessor;

/**
 * Uma vez que registros CSV já estão disponíveis para o chunk, este processador será
 * responsável por representá-los de acordo com o domínio da aplicação, a partir de
 * entidades como livros, autores e editoras.
 */
public class Processor implements ItemProcessor<CsvRecord, Book> {

    private static Logger LOGGER = LoggerFactory.getLogger(Processor.class);

    /**
     * Realiza uma simples transformação do registro CSV, representando-o através de uma
     * composição em que autores, editoras e livros são associados entre si, mas,
     * ao mesmo tempo, representados a partir de tipos de objetos
     * distintos, separados.
     *
     * @param csvRecord O registro CSV a ser processado
     * @return O livro já devidamente representado
     * @throws Exception Em caso de algum problema que venha a ocorrer ao longo
     * do processamento
     */
    public Book process(CsvRecord csvRecord) throws Exception {
        Book book = new Book();
        book.setNumberOfPages(csvRecord.getNumberOfPages());
        book.setTitle(csvRecord.getBookTitle());
        book.setSubTitle(csvRecord.getBookSubTitle());

        LOGGER.info("Iniciando o processamento do registro CSV referente ao livro {}", csvRecord.getBookTitle());

        PublishingHouse publishingHouse = new PublishingHouse();
        publishingHouse.setName(csvRecord.getPublishingHouseName());

        LOGGER.info("Iniciando a criação de um objeto para representar a editora {}", csvRecord.getPublishingHouseName());
        publishingHouse.setAddress(csvRecord.getAddress());
        publishingHouse.setCity(csvRecord.getCity());
        publishingHouse.setCountry(csvRecord.getCountry());
        publishingHouse.setPostalCode(csvRecord.getPostalCode());
        publishingHouse.setPhone(csvRecord.getPhone());
        publishingHouse.setEmail(csvRecord.getEmail());
        publishingHouse.setFax(csvRecord.getFax());

        LOGGER.info("Configurando um objeto para representar o autor {}", csvRecord.getAuthorName());
        book.setAuthor(new Author(csvRecord.getAuthorName()));
        book.setPublishingHouse(publishingHouse);

        LOGGER.info("Livro devidamente processado e pronto para ser devolvido ao
        chunk");
        return book;
    }
}
```

Listagem 6. Código da classe MySQLLoader.

```
package br.com.devmedia.articles.spring.batch.etl.loaders;

import br.com.devmedia.articles.spring.batch.etl.loaders.services.BookService;
import br.com.devmedia.articles.spring.batch.etl.model.Book;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.item.ItemWriter;
import java.util.List;

/**
 * À medida que os registros estão prontos, transformados em objetos de negócio,
 * este item writer será responsável por, enfim, salvá-los no banco de dados
 * MySQL para, desta forma, torná-los disponíveis para consumidores
 * de toda ordem (outros sistemas, acesso direto...)
 */
public class MySQLLoader implements ItemWriter<Book> {

    private static Logger LOGGER = LoggerFactory.getLogger(MySQLLoader.class);

    private BookService service;

    /**
     * Salva uma lista de livros já devidamente representados nos domínios da aplicação.
     *
     * @param books A lista de livros a ser escrita no banco de dados relacional
     * @throws Exception Caso qualquer problema ocorra durante a escrita dos objetos
     */
    public void write(List<? extends Book> books) throws Exception {
        for (Book eachBook : books) {
            service.saveBook(eachBook);
        }
    }

    public void setService(BookService service) {
        this.service = service;
    }
}
```

Entretanto, uma vez que o escopo deste artigo é introduzir apenas os conceitos e a estrutura fundamentais do framework, decidimos empregar em nosso ETL um mecanismo mais tradicional e conhecido do grande público. Para os leitores que desejam se aprofundar neste assunto, no entanto, listamos uma boa referência na seção **Links**.

A importância do uso de transações neste código está na garantia de que todo o processo de persistência de um registro composto, como é um objeto do tipo **Book**, seja realizado consistentemente. Uma falha em qualquer das etapas da gravação de um livro (como na persistência de uma editora ainda não cadastrada no banco, por exemplo) deve significar, invariavelmente, o cancelamento da operação de persistência em sua totalidade.

Daremos início ao estudo do código de nosso item writer pelo conteúdo da **Listagem 6**. Nele, exibimos a implementação da classe **MySQLLoader**.

Aqui, o primeiro ponto a ressaltar é que, por ser um item writer baseado no Spring Batch, esta classe implementa a interface **org.springframework.batch.item.ItemWriter**, que, como já dissemos, é a API básica deste tipo de componente, sendo composta por apenas um método, de assinatura:

```
void write(List<? extends T> var1) throws Exception;
```

A partir desta assinatura, percebe-se que todo *item writer* espera, como entrada, um conjunto de registros para, enfim, persisti-lo e/ou disponibilizá-lo em algum canal para potenciais consumidores. No caso específico da classe **MySQLLoader**, fizemos uso de um serviço denominado **BookService** para persistir, objeto a objeto (e de forma transacional), cada um dos livros no banco de dados. Ao final deste processo, nosso sistema deverá garantir o seguinte panorama:

- Todos os livros foram devidamente salvos no banco de dados;
- Todos os autores foram devidamente cadastrados, sem repetição;

```
private BookService service;

    /**
     * Salva uma lista de livros já devidamente representados nos domínios da aplicação.
     *
     * @param books A lista de livros a ser escrita no banco de dados relacional
     * @throws Exception Caso qualquer problema ocorra durante a escrita dos objetos
     */
    public void write(List<? extends Book> books) throws Exception {
        for (Book eachBook : books) {
            service.saveBook(eachBook);
        }
    }

    public void setService(BookService service) {
        this.service = service;
    }
}
```

- Todas as editoras foram devidamente cadastradas, também sem repetição.

A classe **BookService**, como se vê na **Listagem 7**, será responsável pela coordenação dos objetos de acesso a dados (DAO), garantindo que:

- Os DAOs sejam invocados na sequência correta e sejam alimentados com todas as informações que precisam para executar suas atividades. Não seria aceitável, por exemplo, que um livro fosse persistido antes que a editora e o autor com os quais se relaciona tenham sido salvos antes;
- Um livro duplicado (originado a partir de registros CSV duplicados, por exemplo) jamais seja cadastrado duas vezes no banco de dados.

Os detalhes da implementação do serviço e dos objetos DAO não serão cobertos no texto por uma mera questão de espaço. No entanto, encontram-se bem documentados no código-fonte, oferecendo ao leitor todas as informações que precisará saber para entender o seu funcionamento. Outro motivo para não prolongarmos o artigo com esses detalhes é que, embora sejam importantes para a operação do ETL em si, não representam conceitos do Spring Batch propriamente dito.

Da mesma forma que fizemos ao descrever o conceito de *item readers*, é importante mencionar que poderíamos ter economizado bastante na escrita de nosso *item writer* se tivéssemos adotado algum dos componentes da API do Spring Batch.

O motivo aqui é exatamente igual: nossa intenção foi a de, com a abordagem que adotamos, tornar o texto mais fluido e amigável para o leitor. Deixamos, entretanto, um convite para que visite a documentação oficial do framework. Além de muito bem documentada, ela traz exemplos de tudo aquilo que é possível fazermos com o Spring Batch. No caso específico de *item writers*, por exemplo, haviam candidatos para utilizarmos na escrita de

Introdução ao Spring Batch – Parte 2

registros em bancos de dados. Estamos falando, por exemplo, de componentes como `org.springframework.batch.item.database.JdbcBatchItemWriter` ou, ainda, `org.springframework.batch.item.database.HibernateItemWriter`.

Listagem 7. BookService – Serviço de persistência de livros no banco de dados.

```
package br.com.devmedia.articles.spring.batch.etl.loaders.services;

import br.com.devmedia.articles.spring.batch.etl.loaders.databases.AuthorDao;
import br.com.devmedia.articles.spring.batch.etl.loaders.databases.BookDao;
import br.com.devmedia.articles.spring.batch.etl.loaders.databases.PublishingHouseDao;
import br.com.devmedia.articles.spring.batch.etl.loaders.databases.exceptions.DuplicateRecordException;
import br.com.devmedia.articles.spring.batch.etl.model.Book;
import org.springframework.transaction.annotation.Transactional;

/**
 * Gerencia o cadastro de livros a partir da coordenação de todos os objetos de acesso a dados que devem ser envolvidos neste processo.
 * Como objetos que representam livros são complexos e se relacionam com outros tipos de entidades como autores e editoras, é importante que a operação de salvamento de registros em qualquer fonte de dados seja transacional, para garantirmos que o processo seja revertido caso qualquer das etapas falhe. Este é o modo pelo qual garantiremos a consistência dos dados.
 */
public class BookService {

    private BookDao bookDao;
    private AuthorDao authorDao;
    private PublishingHouseDao publishingHouseDao;

    @Transactional(rollbackFor = {DuplicateRecordException.class})
    public Book saveBook(final Book book) throws DuplicateRecordException {
        publishingHouseDao.save(book.get PublishingHouse());
        book.set PublishingHouse(publishingHouseDao.find(book.get PublishingHouse().getName()));
        authorDao.save(book.get Author());
        book.set Author(authorDao.find(book.get Author().getName(),
            book.get Author().get Surname()));
        bookDao.save(book);
        return book;
    }

    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }

    public void setAuthorDao(AuthorDao authorDao) {
        this.authorDao = authorDao;
    }

    public void setPublishingHouseDao(PublishingHouseDao publishingHouseDao)
    {
        this.publishingHouseDao = publishingHouseDao;
    }
}
```

Executando o ETL

Pronto! Agora que já percorremos os principais tipos de objetos de nosso ETL, é hora de aprendermos a colocá-lo em execução. Observemos, para isso, o código da **Listagem 8**. Esta é a classe principal de nosso projeto.

Listagem 8. Main – Classe principal, que inicia a execução da tarefa.

```
package br.com.devmedia.articles.spring.batch.etl;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersInvalidException;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.batch.core.repository.JobExecutionAlreadyRunningException;
import org.springframework.batch.core.repository.JobInstanceAlreadyCompleteException;
import org.springframework.batch.core.repository.JobRestartException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * Classe principal do nosso módulo ETL, responsável pela inicialização da aplicação.
 * A partir desta classe é que o contexto da aplicação é carregado e o sistema é, enfim, executado.
 */
public class Main {

    private static Logger logger = LoggerFactory.getLogger(Main.class);

    public static void main (String[] arguments) {
        ApplicationContext context = new ClassPathXmlApplicationContext ("etlapp-context.xml");
        JobLauncher launcher = (JobLauncher) context.getBean("jobLauncher");
        Job job = (Job) context.getBean("devmedia_job");

        try {
            JobExecution execution = launcher.run(job, new JobParameters());
            logger.info("Execution has finished with exit code " +
                execution.getExitStatus().getExitCode());
            logger.info("Execution exit description: " + execution.getExitStatus().getExitDescription());

        } catch (JobExecutionAlreadyRunning e) {
            e.printStackTrace();
        } catch (JobRestartException e) {
            e.printStackTrace();
        } catch (JobInstanceAlreadyCompleteException e) {
            e.printStackTrace();
        } catch (JobParametersInvalidException e) {
            e.printStackTrace();
        }
    }
}
```

Observe que, como primeira instrução dentro do método `main()`, estamos carregando todo o contexto declarado no arquivo descriptor `etlapp-context.xml`, exibido na **Listagem 2**. Quando isto ocorre, todos os objetos de todos os beans declarados neste arquivo e, também, no descriptor dos beans de persistência (`etlapp_persistence-context.xml`) são carregados em memória e já se encontram prontos para serem utilizados.

A partir daí, este objeto de contexto será acessado para se obter uma referência ao objeto JobLauncher, o responsável por, enfim, iniciar a execução do Job. Na sequência, o Job é colocado, enfim, para executar, e uma referência ao objeto JobExecution correspondente é guardada apenas para exibirmos parte de seus atributos, quando a execução for finalizada.

Neste artigo introduzimos os conceitos elementares de um poderoso framework para o desenvolvimento de sistemas de processamento em lotes: o Spring Batch. A partir de tudo o que abordamos, o leitor já terá subsídios suficientes para iniciar sua jornada no desenvolvimento de aplicações deste gênero, que, conforme comentamos, é especialmente comum no âmbito corporativo.

Há, entretanto, muito mais sobre Spring Batch do que o conteúdo do artigo pôde cobrir. Com este framework, podemos interceptar praticamente todas as fases da execução de um job através do uso de listeners (para fins de registros em arquivos de log ou qualquer outra necessidade verificada). Paralelização de execução, delegação de processamento em nível de *item writers* e *item readers* e particionamento são só alguns exemplos a mais de recursos que podem ser utilizados, tornando evidente o quanto poderosa é esta solução.

Esperamos, no entanto, que tudo aquilo que vimos ao longo do texto seja suficiente para estimular o leitor a, a partir das referências mencionadas, buscar mais informações e experimentar em formas diversas parte de tudo aquilo que o Spring Batch tem a oferecer.

Autor



Pedro E. Cunha Brigatto

pedrobrigatto@gmail.com

Engenheiro da Computação graduado pela Universidade Federal de São Carlos, desenvolvedor certificado SAP Netweaver (Java Stack) e programador certificado SCJP. Especialista em Engenharia de Software graduado pela Unimep e pós-graduado em Administração pela Fundação BI-FGV, atua com desenvolvimento de software desde 2005. Atualmente atua como consultor técnico no desenvolvimento de soluções de alta disponibilidade na Avaya.



Links:

Código-fonte do projeto.

https://bitbucket.org/pedrobrigatto/devmedia-springbatch_etl

Documentação de referência do Spring Batch.

<http://docs.spring.io/spring-batch/reference/html/>

Excelentes tutoriais para iniciação com Spring Batch.

<http://www.mkyong.com/tutorials/spring-batch-tutorial/>

Uso de transações com Spring.

<http://www.journaldev.com/2603/spring-transaction-management-example-with-jdbc>

Uso de transações com Spring Batch.

<https://blog.codecentric.de/en/2012/03/transactions-in-spring-batch-part-1-the-basics/>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Relatórios avançados com Hibernate, Jasper-Reports e PrimeFaces – Parte 3

Aprenda neste artigo a construir templates com Facelets e como integrar o JSF à arquitetura proposta para nosso simulador

ESTE ARTIGO FAZ PARTE DE UM CURSO

Chegamos, enfim, à última parte da série de artigos que apresentou como desenvolver um simulador de perguntas e respostas. Para isso, já exploramos tecnologias como JasperReports, iReport, Tomcat, MySQL, Hibernate e padrões como MVC e DAO. Nesta última parte faremos a integração do framework JSF à arquitetura construída para nosso simulador.

Para facilitar a implementação da interface, faremos uso do PrimeFaces e do Facelets, importante projeto que simplifica a construção de templates e viabiliza o reuso de código, bem como um pouco de CSS, JavaScript e outros recursos relacionados.

Criando o arquivo de mensagens

Com a primeira e a segunda parte do simulador concluídas, iniciamos esta última etapa com a criação do arquivo de mensagens da aplicação. Este procedimento é recomendado para implementar a internacionalização. Deste modo, podemos apresentar informações em diferentes idiomas, simplesmente ao referenciar chaves textuais genéricas. Por exemplo, para apresentar uma mensagem de boas-vindas na página inicial da aplicação em portu-

Fique por dentro

O sucesso de um projeto de software está diretamente relacionado com a escolha de sua arquitetura. Sabendo disso, é importante adotar soluções arquiteturais que tanto nos trarão maior produtividade no desenvolvimento com a reutilização de código, quanto aumentarão a manutenibilidade da aplicação. Com base nesta meta, neste artigo demonstraremos como integrar os frameworks Java Server Faces e Hibernate em uma arquitetura MVC e como construir a interface da aplicação de modo a maximizar o reuso de código.

guês ou inglês, teríamos chaves com as mesmas assinaturas, porém em diferentes arquivos, cada um com um valor para esta chave de acordo com o idioma. Portanto, para a chave **mensagem.saudacao** do arquivo de idioma em português, teríamos a mensagem “Seja bem-vindo!”, e para o arquivo com os termos em inglês, a chave **mensagem.saudacao** apresentaria o valor “Welcome!”. A alteração do idioma da aplicação é viabilizada pelo recurso de internacionalização em conjunto com o browser, que identifica o idioma padrão do usuário. Assim, ganhamos tanto na manutenibilidade quanto melhoramos a organização das informações textuais do sistema e, ainda, contribuímos para o seu reuso.

A primeira coisa que devemos fazer para configurar este mecanismo é alterar o arquivo faces-config.xml, que se encontra no diretório src/main/webapp/WEB-INF, para que fique igual ao apresentado na **Listagem 1**.

O próximo passo é criar o arquivo de mensagens. Dito isso, dentro da pasta `src/main/resources`, crie um arquivo de propriedades com o nome “messages”. Observe que o nome do arquivo deve ser o mesmo da tag **base-name** do *faces-config*. Finalmente, acrescente as informações expostas na **Listagem 2**.

Listagem 1. Código do arquivo faces-config.xml.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
03 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
05 http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
06 version="2.2">
07
08   <application>
09     <resource-bundle>
10       <base-name>messages</base-name>
11       <var>message</var>
12     </resource-bundle>
13   </application>
14 </faces-config>
```

Listagem 2. Código do arquivo messages.properties.

```
01 main.app.title=Simulador
02 main.footer.text=Javamagazine
03 submenu.menu=Menu
04 submenu.simulado=Simulado
05 pagina.timeout=Tempo Limite Expirou ! Reinicie o Teste.
06 pagina.simples.label=Marque a Correta
07 pagina.multipla.label=Marque as Corretas
08 pagina.arrasta.solta.label.iniciar=Iniciar
09 pagina.arrasta.solta.label.disponivel=Nenhuma Resposta Dispon\u00edvel
10 pagina.arrasta.solta.label.selecionada=Nenhuma Resposta Selecionada
11 pagina.arrasta.solta.field.disponivel=Dispon\u00edvel
12 pagina.arrasta.solta.field.selecionada=Selecionadas
13 pagina.arrasta.solta.label.resposta=Resposta
```

Listagem 3. Código do arquivo estilo.css.

```
01 label {
02   display: block;
03   margin: 10px;
04 }
05 #geral {
06   width: 95%;
07   margin: 0 auto;
08   clear: both;
09 }
10 #miolo {
11   width: 99%;
12   display: block;
13   clear: both;
14   background: #FFF;
15   border: 1px solid #CCC;
16   min-height: 500px;
17   height: auto !important;
18   height: 500px;
19   margin-top: 15px;
20   margin-bottom: 15px;
21 }
22 #rodape {
23   bottom: 0;
24   width: 98%;
25   margin: 0 auto;
26   clear: both;
27   height: 30px;
28   background-color: #f5f5f5;
29   text-align: center;
30   padding: 10px;
31 }
32 #timeout {
33   color: #FF0000;
34   text-align: center;
35 }
```

Criando a folha de estilo do sistema

Optamos também por centralizar toda a formatação visual do simulador através da utilização de uma folha de estilo CSS. Nossa intenção é a organização e reutilização das estruturas responsáveis pelo layout dos elementos de interface da aplicação, de forma a aumentar a produtividade. Por exemplo, ao declarar no arquivo CSS a propriedade `label {margin:10px;}` para o elemento `label` do HTML, especificamos que todas as páginas que utilizarem esse CSS e declararem tags `outputLabel` do PrimeFaces terão os respectivos componentes apresentados com a margem de 10 pixels.

Sabendo disso, nosso próximo passo será a criação do arquivo de estilo. Para tanto, dentro de `src/main/webapp`, criaremos primeiramente uma pasta chamada `resources`, e a partir dela, criaremos mais uma, com o nome `css`. Por fim, vamos gerar, dentro desta pasta, um arquivo CSS chamado “estilo”. Com esta etapa concluída, adicione o código da **Listagem 3** em `estilo.css`.

Criando as páginas da aplicação

Dando continuidade ao desenvolvimento da aplicação, vamos à construção das páginas XHTML. Neste momento vale lembrar que desde a primeira parte do artigo demonstramos a importância em

codificar estruturas coesas, com baixo acoplamento e empregando ao máximo os recursos de uma linguagem orientada a objetos, e aqui não será diferente, pois utilizaremos um recurso muito poderoso do JSF, o Facelets. Com ele construiremos um template que será reutilizado por todas as páginas do sistema e, assim, ganharemos mais uma vez na produtividade e organização do código.

Com base nisso, nosso primeiro passo será codificar o menu da aplicação, que será reutilizado por todas as páginas do simulador. É através dele que podemos iniciar o quiz. Para isso, dentro de `src/main/webapp`, crie um arquivo XHTML com o nome “menu” e substitua seu código pelo da **Listagem 4**.

Os trechos de código mais importantes dessa listagem são:

- Linhas 6 a 12: O componente `<p:menubar />` do PrimeFaces apresenta a barra de menu da aplicação;
- Linhas 7-11: O componente `<p:submenu />`, também do PrimeFaces, cria um submenu que nos possibilita acessar a página de início do simulador.

Criando o template

Criaremos agora, com o auxílio de Facelets, o template das páginas da nossa aplicação. Essa estrutura que viabiliza a reutilização

de código será empregada para compartilhar trechos comuns a todas as páginas do sistema como, por exemplo, o menu e o rodapé.

Listagem 4. Código da página menu.xhtml.

```
01 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
02   xmlns:f="http://java.sun.com/jsf/core"
03   xmlns:p="http://primefaces.org/ui" xmlns:h="http://java.sun.com/jsf/html"
04   xmlns:ui="http://java.sun.com/jsf/facelets">
05 
06   <h:form>
07     <p:menubar>
08       <p:submenu label="#{message['submenu.menu']}'>
09         <p:submenu label="#{message['submenu.simulado']}'>
10           <p:menuitem value="#{message['menuitem.iniciar.simulador']}"
11             url="/avaliacao/simulador.xhtml"/>
12         </p:submenu>
13       </p:menubar>
14     </h:form>
15   </ui:composition>
```

Listagem 5. Código da página main.xhtml.

```
01 <html xmlns:f="http://java.sun.com/jsf/core"
02   xmlns:p="http://primefaces.org/ui"
03   xmlns:h="http://java.sun.com/jsf/html"
04   xmlns:ui="http://java.sun.com/jsf/facelets">
05 
06   <f:view contentType="text/html" locale="pt_BR"/>
07 
08   <ui:param name="PATH" value="#{facesContext.externalContext.request
09   ContextPath}"/>
10 
11   <h:head>
12     <title>#{message['main.app.title']}</title>
13     <link type="text/css" rel="stylesheet" href="#{PATH}/resources/css/estilo.css"/>
14   </h:head>
15 
16   <h:body>
17     <table id="geral">
18       <tr>
19         <td height="30"><ui:include src="/menu.xhtml"/></td>
20       </tr>
21 
22       <tr valign="top">
23         <td height="200"><ui:insert name="body"/></td>
24       </tr>
25 
26       <tr align="center">
27         <td height="20" align="center">
28           <div id="rodape" style="font-weight:bold;">
29             =#{message['main.app.title']} - #{message['main.footer.text']}
```

Assim, dentro da pasta `src/main/webapp`, crie uma nova pasta, chamada `template`. Na sequência, crie um novo arquivo XHTML, de nome “main”, e substitua o código gerado automaticamente pelo Eclipse pelo código exposto na **Listagem 5**.

Os principais trechos desse código são:

- **Linha 8:** O componente `<ui:param />` declara um parâmetro que pode ser utilizado por qualquer página que fizer uso deste

template. Neste exemplo declaramos um parâmetro de nome `PATH` que guarda o nome do contexto da aplicação. Assim, podemos referenciar arquivos de estilo ou de script sem ter a necessidade de replicar o contexto da aplicação como prefixo do caminho destes arquivos, como podemos observar na linha 12, comentada mais adiante;

- **Linha 11:** Acessamos nosso arquivo de mensagens através da marcação `#{{message['chave']}}.` Para que isto seja possível, o nome da marcação deve coincidir com o nome declarado pela tag **base-name** do arquivo `faces-config`. Já o valor da chave deve corresponder ao de uma das chaves declaradas no arquivo de mensagens;
- **Linha 12:** Declaramos nossa folha de estilo. Observe também que utilizamos o parâmetro declarado anteriormente para indicar o local onde se encontra o arquivo CSS;
- **Linha 18:** O componente `<ui:include />` serve para que possamos incluir estruturas de código reutilizáveis. Neste trecho incluímos o menu da aplicação. Deste modo, todas as páginas que estenderem este template também irão herdar o menu;
- **Linha 22:** O componente `<ui:insert />` declara um local que poderá receber códigos de outras estruturas. Com isso, todas as páginas que utilizarem este template terão a possibilidade de inserir códigos neste ponto. Para tanto, basta declarar outra tag do Facelets, a `<ui:define />`, no arquivo envolvendo os trechos de código que se deseja reutilizar. A partir disso, em `<ui:insert />`, basta informar o **name** especificado em `<ui:define />` para reutilizar o código;
- **Linha 26:** Declaramos o rodapé, mais uma porção de código reutilizável e que será replicada em todas as páginas que estenderem o template.

Criando a página inicial

Com nosso `template` criado, vamos elaborar a página inicial da aplicação. Desta maneira, crie o arquivo `pagina_inicial.xhtml` dentro de `src/main/webapp` e, da mesma forma que fizemos ao criar o menu, modifique o código gerado pelo Eclipse para ficar semelhante ao da **Listagem 6**.

Listagem 6. Código da página pagina_inicial.xhtml.

```
01 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
02   xmlns:f="http://java.sun.com/jsf/core"
03   xmlns:p="http://primefaces.org/ui"
04   xmlns:h="http://java.sun.com/jsf/html"
05   xmlns:ui="http://java.sun.com/jsf/facelets"
06   template="/template/main.xhtml">
07   </ui:composition>
```

Podemos observar na linha 1 que declaramos o componente `<ui:composition />` para informar ao JSF que utilizaremos um template. Na linha 6, dentro desta tag, indicamos que a página `main.xhtml` será nosso `template`. Desta forma, todas as porções de código reutilizáveis, como o *menu* e o *rodapé*, poderão ser verificadas ao carregar esta página e assim evitamos a replicação de

código em outros lugares e simplificamos a implementação.

Antes de continuar com a implementação das páginas do simulador, codificaremos um conversor que será necessário para o funcionamento correto de determinados componentes do JSF e do PrimeFaces, como veremos logo mais.

Criando o converter do projeto

Quando trabalhamos com sistemas web, temos a necessidade de apresentar informações textuais em nossas páginas. No entanto, muitas vezes estas informações são provenientes de objetos complexos, em vez de simples strings. Quando isso acontece, é preciso desenvolver um conversor para realizar a transformação destes objetos complexos em strings e vice-versa de forma automática. Como exemplo disso, temos um combobox de avaliações na página do simulador para listar objetos do tipo **Avaliacao** e exibir o conteúdo de sua propriedade **descricao**. A partir deste componente, para que o sistema identifique o respectivo objeto a partir da **descricao** selecionada pelo usuário, precisamos indicar no componente qual conversor ele deverá utilizar. Dependendo da necessidade, em uma aplicação podemos ter diversos convertores, mas no nosso simulador, teremos apenas um. Portanto, crie a classe **EntityConverter** no pacote **br.com.javamagazine.simulador.conve** e substitua o código gerado pelo da [Listagem 7](#).

Como dito anteriormente, uma aplicação web precisa de conversores para apresentar informações textuais de objetos complexos. Uma das maneiras de conseguir isso é utilizar o atributo

Listagem 7. Código da classe EntityConverter.

```
01 package br.com.javamagazine.simulador.conve;
02
03 import javax.faces.component.UIComponent;
04 import javax.faces.context.FacesContext;
05 import javax.faces.convert.Converter;
06 import javax.faces.convert.FacesConverter;
07 import br.com.javamagazine.simulador.persistence.HibernateUtil;
08
09 @FacesConverter("domainConverter")
10 public class EntityConverter implements Converter {
11
12     @Override
13     public String getAsString(FacesContext facesContext,
14                               UIComponent component, Object object) {
14         if (object == null) {return null;}
15         try {
16             Class<?> classe = object.getClass();
17             Long id = (Long) classe.getMethod("getId").invoke(object);
18             return classe.getName() + "." + id;
19         } catch (Exception e) {return null;}
20     }
21
22     @Override
23     public Object getAsObject(FacesContext facesContext,
24                               UIComponent component, String string) {
24         if (string == null || string.isEmpty()) {return null;}
25         try {
26             String[] values = string.split("-");
27             return HibernateUtil.getEntityManager().find(Class.forName(values[0]),
28                                           Long.valueOf(values[1]));
28         } catch (Exception e) {return null;}
29     }
30 }
```

RENOVE JÁ!

Sua assinatura pode estar acabando

Renovando a assinatura de sua revista favorita você ganha brindes e descontos exclusivos.

www.devmedia.com.br/renovacao ou (21)3382-5038

converter em determinados componentes; geralmente os que trabalham com listas de objetos como é o caso do **selectOneMenu** do PrimeFaces.

Nesta listagem, anotamos a classe com **@FacesConverter** (vide linha 9) e especificamos o valor **domainConverter**, que deve ser o mesmo utilizado pelo atributo

converter nos componentes. Já na linha 13 temos o método **getAsString()**, que recebe um objeto qualquer e retorna uma **String** no formato “*nome_completo_da_classe-id_objeto*”, formato este que será utilizado pelo método comentado a seguir. E na linha 23 implementamos o método **getAsObject()**, que recebe a **String** formada

pelo método anterior e a divide em duas partes. Com isso temos a possibilidade de utilizá-las como argumento do método **find(“*nome_completo_da_classe*”, *id_objeto*)** do Hibernate para recuperar determinados objetos do banco de dados, como pode ser visto na linha 27.

Criando os tipos de questões do simulador

Como vimos anteriormente, ao utilizar facelets em nosso projeto ganhamos a possibilidade de criar pedaços de códigos reutilizáveis. Pensando nisso e lembrando que nosso simulador possui diferentes tipos de questões (marcação simples com apenas uma alternativa possível (**Figura 1**); e múltiplas escolhas, que podem ser exibidas na forma de múltiplas marcações (**Figura 2**) ou na forma de arrastar e soltar (**Figura 3**)), optamos por criar separadamente cada uma dessas estruturas. A intenção com isso é possibilitar a renderização das respostas em diferentes formatos, de acordo com o tipo da questão.

Para isso, cada objeto pergunta terá um campo chamado **tipoQuestaoEnum** e nele determinaremos se a pergunta terá uma ou mais respostas e ainda, como suas respostas serão apresentadas. Por exemplo, caso uma pergunta seja de marcação simples, ela terá como propriedade enum **tipoQuestao** o valor **MARCAR_SIMPLES**. Diante disso, na página do simulador teremos um trecho de código que verificará o tipo de questão e incluirá trechos de código equivalentes ao tipo identificado. Isto é feito através da tag **<ui:include>** do Facelets, que neste caso vai inserir a porção de código referente a questões de marcação simples e esta, por sua vez, apresentará as respostas no formato apropriado.

Sabendo disso, vamos criar o bloco de código que apresentará as respostas em formato de marcação simples. Para tanto, crie a pasta **tipos_questoes** dentro de **src/main/webapp** e depois o arquivo **simples.xhtml**. Seu código deve ficar igual ao apresentado na **Listagem 8**.

Sobre essa listagem, vale destacar:

- **Linha 7:** O componente **<ui:insert />** declara uma área de código que pode ser reutilizada. Desta forma, seu conteúdo

The screenshot shows a question card with the following details:
Nome: Avaliacao 1 **Dificuldade:** Difícil **Tipo Questão:** Marcar Simples **Tema:** Tema 2
Questão: 2 **Total Questões:** 5 **Tempo:** 00:01:23 **Data Avaliação:** 20/08/2015
Pergunta: Pergunta 2
Marque a Correta
 Reposta 0
 Reposta 1
 Reposta 2
 Reposta 3
 Reposta 4

Figura 1. Questão do tipo marcação simples

The screenshot shows a question card with the following details:
Nome: Avaliacao 1 **Dificuldade:** Muito Difícil **Tipo Questão:** Marcar Múltiplas **Tema:** Tema 2
Questão: 3 **Total Questões:** 5 **Tempo:** 00:00:54 **Data Avaliação:** 20/08/2015
Pergunta: Pergunta 3
Marque as Corretas
 Reposta 0 Reposta 1
 Reposta 2 Reposta 3
 Reposta 4

Figura 2. Questão do tipo múltiplas marcações

The screenshot shows a question card with the following details:
Nome: Avaliacao 1 **Dificuldade:** Muito Difícil **Tipo Questão:** Arrastar e Soltar **Tema:** Tema B
Questão: 1 **Total Questões:** 5 **Tempo:** 00:01:30 **Data Avaliação:** 02/06/2015
Pergunta: Pergunta 1
Disponíveis
Clique em Iniciar!
 Reposta 0
 Reposta 1
 Reposta 2
 Reposta 3
 Reposta 4
Selecionadas
Nenhuma Resposta Selecionada

Figura 3. Questão do tipo arrastar e soltar

pode ser inserido em outras páginas que a referenciarem através do uso da tag `<ui:include />`, como veremos ao criar a página `simulador.xhtml`;

• **Linha 13:** Para renderizar as respostas de uma determinada pergunta no formato de marcação simples, usamos a tag `<p:selectOneMenu />` do PrimeFaces. Com ela determinamos que uma pergunta terá apenas uma resposta. Como este componente trabalha com objetos complexos, precisamos sinalizar ao JSF qual conversor utilizar, o que é feito através do atributo `converter`.

Listagem 8. Código da página simples.xhtml.

```
01 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
02   xmlns:f="http://java.sun.com/jsf/core"
03   xmlns:p="http://primefaces.org/ui"
04   xmlns:h="http://java.sun.com/jsf/html"
05   xmlns:ui="http://java.sun.com/jsf/facelets">
06
07   <ui:insert name="respostaSimples">
08
09     <p:fieldset style="margin-top:10px">
10       <h:panelGrid>
11         <p:outputLabel value="#{message['pagina.simples.label']}"
12           style="font-weight:bold;" />
13
14         <p:selectOneRadio value="#{simuladorBean.avaliacaoVO.pergunta.
15           respostaSimples}" converter="domainConverter" layout="grid"
16           columns="1" disabled="#{!simuladorBean.iniciar}">
17           <p:ajax event="change" update="@this" />
18           <f:selectItems value="#{simuladorBean.avaliacaoVO.
19             pergunta.listaResposta}" var="resposta"
20             itemLabel="#{resposta.descricao}" itemValue="#{resposta}" />
21
22         </p:selectOneRadio>
23       </h:panelGrid>
24     </p:fieldset>
25
26   </ui:insert>
27
28 </ui:composition>
```

Dando continuidade ao processo de criação das estruturas que apresentarão as respostas de acordo com o tipo de questão, criaremos agora a página que apresentará as respostas em formato de múltiplas escolhas. Este bloco de código nos possibilitará selecionar mais de uma resposta para determinada pergunta, diferentemente do bloco anterior, onde é possível selecionar apenas uma resposta. Para isso, dentro da pasta `tipos_questoes`, crie o arquivo `multipla.xhtml` e modifique o código gerado pelo Eclipse para corresponder ao da **Listagem 9**.

A seguir destacamos o principal trecho de código desse arquivo:

• **Linha 13:** Com o componente `<p:selectManyCheckbox />` do PrimeFaces, temos a possibilidade de marcar múltiplas respostas para uma determinada pergunta. Além disso, da mesma forma que fizemos na **Listagem 8**, devemos informar através do atributo `converter` qual será o conversor deste componente, já que estamos trabalhando com objetos complexos do tipo **Resposta**.

Listagem 9. Código da página multipla.xhtml.

```
01 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
02   xmlns:f="http://java.sun.com/jsf/core"
03   xmlns:p="http://primefaces.org/ui"
04   xmlns:h="http://java.sun.com/jsf/html"
05   xmlns:ui="http://java.sun.com/jsf/facelets">
06
07   <ui:insert name="respostaMultipla">
08
09   <p:fieldset style="margin-top:20px">
10     <h:panelGrid>
11       <p:outputLabel value="#{message['pagina.multipla.label']}"
12         style="font-weight:bold;" />
13
14       <p:selectManyCheckbox value="#{simuladorBean.
15         avaliacaoVO.pergunta.respostasSelecionadas}" layout="grid"
16         columns="2" converter="domainConverter"
17         disabled="#{!simuladorBean.iniciar}">
18         <p:ajax event="change" update="@this" />
19         <f:selectItems value="#{simuladorBean.avaliacaoVO.
20           pergunta.listaResposta}" var="resposta"
21             itemLabel="#{resposta.descricao}" itemValue="#{resposta}" />
22       </p:selectManyCheckbox>
23     </h:panelGrid>
24   </p:fieldset>
25
26 </ui:insert>
27
28 </ui:composition>
```

Para finalizar, criaremos a última estrutura para as questões, que se encarrega de apresentar as respostas no formato drag and drop. Deste modo, dentro da pasta `tipos_questoes`, crie o arquivo `arrasta_solta.xhtml` e modifique o código conforme o conteúdo da **Listagem 10**.

Os principais trechos de código são comentados a seguir.

- **Linhas 9 a 14:** Função JavaScript utilizada pelo componente drag and drop do PrimeFaces. Sem ela o componente não funcionaria corretamente, pois não seria possível selecionar um objeto e arrastá-lo de um local para outro. Além disso, essa função abstrai a implementação deste recurso para o desenvolvedor;
- **Linhas 17 a 29:** Painel onde é feita a listagem das respostas disponíveis;
- **Linhas 22 e 37:** O componente `<p:draggable />` do PrimeFaces é o responsável pelo efeito de arrastar. Quando utilizado neste exemplo, teremos a possibilidade de reposicionar uma determinada resposta para uma das áreas, disponíveis ou selecionadas;
- **Linhas 33 a 44:** Painel que armazenará as respostas selecionadas pelo usuário. Para selecionar uma resposta, deve-se posicionar a seta do mouse no símbolo que a representa;
- **Linhas 47 a 52:** Além desta implementação possibilitar a retirada de objetos da área de disponíveis e movê-los para a área de selecionados, temos também a possibilidade de os retornar para sua área de origem. Isto é possível por causa da utilização do componente `<p:droppable />`.

Criando a página de timeout

Após a criação das estruturas que apresentarão as respostas em formato de marcação simples ou marcações múltiplas, implementaremos o timeout da aplicação, uma importante funcionalidade

Listagem 10. Código da página arrasta_solta.xhtml.

```

01 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
02   xmlns:f="http://java.sun.com/jsf/core"
03   xmlns:p="http://primefaces.org/ui"
04   xmlns:h="http://java.sun.com/jsf/html"
05   xmlns:ui="http://java.sun.com/jsf/facelets">
06
07   <ui:insert name="respostaArrastaSolta">
08
09     <script type="text/javascript">
10       function handleDrop(event, ui) {
11         var droppedQuestion = ui.draggable;
12         droppedQuestion.fadeOut('fast');
13       }
14     </script>
15
16   <p:fieldset id="fieldRespostasDisponiveis"
17     legend="#{message['pagina.arrasta.solta.field.disponivel']}"
18     style="margin-top:20px">
19     <p:outputPanel id="putArea">
20       <p:outputLabel value="#{message['pagina.arrasta.solta.label.iniciar']}"
21         rendered="#{!simuladorBean.iniciar}" style="font-weight:bold;" />
22       <p:dataTable id="respostasDisponiveis"
23         var="resposta" value="#{simuladorBean.avaliacaoVO.pergunta.listaResposta}"
24         emptyMessage="#{message['pagina.arrasta.solta.label.disponivel']}">
25         <p:column style="width:20px">
26           <h:outputText id="dragIcon" styleClass="ui-icon ui-icon-disk"/>
27           <p:draggable for="dragIcon" revert="true" helper="clone"
28             disabled="#{!simuladorBean.iniciar}" />
29         </p:column>
30
31         <p:column headerText="#{message['pagina.arrasta.solta.label.resposta']}">
32           <p:outputLabel value="#{resposta.descricao}" />
33         </p:column>
34       </p:dataTable>
35     </p:outputPanel>
36   </p:fieldset>
37
38   <p:fieldset id="fieldRespostasSelecionadas"
39     legend="#{message['pagina.arrasta.solta.field.selecionada']}"
40     style="margin-top:20px">
41     <p:outputPanel id="dropArea">
42       <p:datatable id="respostasSelecionadas" var="resposta"
43         value="#{simuladorBean.avaliacaoVO.pergunta.respostasSelecionadas}"
44         emptyMessage="#{message['pagina.arrasta.solta.label.selecionada']}">
45         <p:column style="width:20px">
46           <h:outputText id="dragIcon" styleClass="ui-icon ui-icon-trash"/>
47           <p:draggable for="dragIcon" revert="true" helper="clone"
48             disabled="#{!simuladorBean.iniciar}" />
49         </p:column>
50       </p:datatable>
51     </p:outputPanel>
52   </p:fieldset>
53
54 </ui:insert>
55
56 </ui:composition>
```

para o simulador. Para isso, utilizaremos uma página XHTML para apresentar uma mensagem de término dos testes sempre que o tempo da avaliação expirar. Ainda nesta parte, demonstraremos o mecanismo de redirecionamento do JSF.

Para implementar a página que exibirá as informações de término dos testes, crie a pasta *avaliacao* dentro de *src/main/webapp*, crie o arquivo *timeout.xhtml* dentro desta pasta e o modifique para que fique semelhante ao da **Listagem 11**.

A seguir analisamos os principais trechos desse código:

- Linha 6:** Note que utilizamos nosso template. Desta forma esta página herdará todos os trechos de código reutilizáveis disponibilizados pelo mesmo, tais como menu e rodapé;
- Linha 10:** Utilizamos a tag *<p:outputLabel />* para exibir a mensagem de timeout proveniente do arquivo de mensagens.

Criando o ManagedBean e a página do simulador

Chegou o momento de criarmos a página que executará as avaliações do simulador. Para isso, iniciaremos com a criação de um JavaBean, que nada mais é do que uma classe Java tradicional contendo atributos com seus respectivos métodos de acesso e também um construtor padrão, obrigatório. Para que o JSF a

Listagem 11. Código da página timeout.xhtml.

```

01 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
02   xmlns:f="http://java.sun.com/jsf/core"
03   xmlns:p="http://primefaces.org/ui"
04   xmlns:h="http://java.sun.com/jsf/html"
05   xmlns:ui="http://java.sun.com/jsf/facelets"
06   template="/template/main.xhtml">
07
08   <ui:define name="body">
09     <div id="content">
10       <p:outputLabel id="timeout" value="#{message['pagina.timeout']}" />
11     </div>
12   </ui:define>
13
14 </ui:composition>
```

reconheça e funcione adequadamente, precisamos configurá-la, e uma das formas de fazer isso é através de anotações.

Após anotá-la adequadamente, como demonstrado na **Listagem 12**, tornamos a classe um ManagedBean, um objeto gerenciado pelo container e utilizado pelo framework JSF. Desta forma, alcançamos a separação entre a camada de negócios e a de apresentação ao qual o framework se propõe.

A declaração de um ManagedBean pode ser feita tanto no arquivo *faces-config* como através do uso de anotações. Em versões anteriores, essas configurações eram efetuadas diretamente no *faces-config*. Neste exemplo, no entanto, utilizaremos anotações, uma opção mais prática e que possibilita um aumento de produtividade. Para criar nosso ManagedBean, primeiramente vamos criar um pacote, de nome **br.com.javamagazine.simulador.view**, e depois uma classe Java, de nome **SimuladorMB**. Seu código-fonte é apresentado na **Listagem 12**.

Listagem 12. Código da classe SimuladorMB.

```

01 package br.com.javamagazine.simulador.view;
02 // imports omitidos;
03
04 @ViewScoped
05 @ManagedBean(name="simuladorBean")
06 public class SimuladorMB implements Serializable {
07
08     private static final long serialVersionUID = 366271030208991605L;
09     private int indice;
10     private boolean iniciar;
11     private AvaliacaoBC avaliacaoBC;
12     private AvaliacaoVO avaliacaoVO;
13
14     @PostConstruct
15     public void init() {
16         this.avaliacaoBC = new AvaliacaoBCImpl();
17         this.avaliacaoVO = new AvaliacaoVO();
18     }
19
20     public void carregarAvaliacao(ValueChangeEvent event) {
21         if (event.getNewValue() != null) {
22             getAvaliacaoVO().setAvaliacao((Avaliacao)event.getNewValue());
23             getAvaliacaoVO().setPergunta(getAvaliacaoVO().getAvaliacao()
24                                         .getListaPergunta().get(Integer.valueOf(ConstantesEnum.
25                                         PRIMEIRA_PERGUNTA.getValor().toString())));
26             getAvaliacaoVO().setTempoAvaliacao(getAvaliacaoVO()
27                                         .getAvaliacao().getTempo());
28         } else {
29             getAvaliacaoVO().setAvaliacao(new Avaliacao());
30         }
31     }
32
33     public void onRetirarResposta(DragDropEvent ddEvent) {
34         Resposta resposta = ((Resposta) ddEvent.getData());
35         getAvaliacaoVO().getPergunta().getRespostasSelecionadas().add(resposta);
36         getAvaliacaoVO().getPergunta().getListaResposta().remove(resposta);
37     }
38
39     public void onRetornarResposta(DragDropEvent ddEvent) {
40         Resposta resposta = ((Resposta) ddEvent.getData());
41         getAvaliacaoVO().getPergunta().getListaResposta().add(resposta);
42         getAvaliacaoVO().getPergunta().getRespostasSelecionadas().remove(resposta);
43     }
44
45     public void avancar() {
46         Integer tam = getAvaliacaoVO().getAvaliacao().getListaPergunta().size() - 1;
47         getAvaliacaoVO().setPergunta(getAvaliacaoVO().getAvaliacao()
48                                         .getListaPergunta().get(getIndice() < tam ? ++this.indice : getIndice()));
49     }
50
51     public boolean desabilitarAvancar() {
52         return !getAvaliacaoVO().getAvaliacao().getListaPergunta().isEmpty()
53             || (getIndice() == getAvaliacaoVO().getAvaliacao().getListaPergunta().size() - 1)
54             || !isIniciar() : false;
55     }
56
57     public void decrementarTempo() {
58         if (!isIniciar()) {
59             Long tim = this.avaliacaoBC.decrementarTempo(getAvaliacaoVO()
60                                         .getTempoAvaliacao());
61             getAvaliacaoVO().setTempoAvaliacao(tim);
62             Calendar cal = Calendar.getInstance();
63             cal.setMillisecond(tim);
64             setIniciar(Cronometro.fimTeste(cal) ? false : isIniciar());
65         }
66     }
67
68     public void finalizar() {
69         try {
70             FacesContext.getCurrentInstance().getExternalContext().getFlash()
71                 .put(ConstantesEnum.CHAVE_FLASH.getValor().toString(), this.avaliacaoVO);
72             FacesContext.getCurrentInstance().getExternalContext()
73                 .dispatch(ConstantesEnum.CAMINHO_RESULTADO.getValor().toString());
74         } catch (IOException e) {
75             e.printStackTrace();
76         }
77     }
78
79     public void timeout() {
80         try {
81             FacesContext.getCurrentInstance().getExternalContext()
82                 .redirect(ConstantesEnum.CAMINHO_TIMEOUT.getValor().toString());
83         } catch (IOException e) {
84             e.printStackTrace();
85         }
86
87     public String getTempoFormatado() {
88         return Cronometro.formatarTempoAvaliacao(getAvaliacaoVO()
89                                         .getTempoAvaliacao());
90
91     public List<Avaliacao> getListaAvaliacao() {
92         return this.avaliacaoBC.listar();
93
94     //getters e setters omitidos...
95 }
```

As principais linhas desse código são explicadas a seguir:

- **Linha 4:** Utilizamos a anotação **@ViewScoped** neste ManagedBean para determinar que ele será atribuído ao escopo de visão, um escopo maior que o de requisição e menor que o de sessão. As instâncias deste ManagedBean serão mantidas enquanto o usuário estiver navegando na página que o referencia (*simulador.xhtml* neste caso);
- **Linha 5:** A anotação **@ManagedBean(name="simuladorBean")** faz com que o JSF reconheça esta classe como um ManagedBean.

Desta forma, poderemos vinculá-la a uma página XHTML e utilizar os recursos do JSF que serão necessários ao funcionamento do simulador;

- **Linha 9:** Criamos a variável `índice`, para que seja possível controlar a apresentação das perguntas de forma sequencial, ou seja, com ela teremos a possibilidade de navegar pelos índices da lista de perguntas da avaliação de modo que o usuário terá a capacidade de controlar essa navegação, avançando para a próxima pergunta ou retornando para a anterior;

- **Linha 14:** A anotação `@PostConstruct` possibilita que este método seja invocado antes que a classe seja disponibilizada como um serviço. Assim, podemos, por exemplo, inicializar objetos que serão utilizados durante o seu fluxo de execução. Poderíamos ter feito isso no próprio construtor, mas optamos por fazê-lo neste local para demonstrar uma das opções de adotar este recurso do JSF;

- **Linhas 20 a 28:** Método de callback do JSF, disparado sempre que o combobox de avaliações for selecionado. A sua principal atribuição é alimentar um objeto VO que será utilizado no decorrer do quiz. Este objeto será alimentado com informações da própria avaliação, tais como o nome da avaliação, o total de questões, o enunciado da pergunta e o tempo total de duração do teste;

- **Linhas 30 a 40:** Métodos de callback utilizados pelo componente drag and drop do PrimeFaces. Com eles, possibilitamos trafegar respostas entre as listas de respostas disponíveis e selecionadas. Por exemplo, quando o método `onRetirarResposta()` é disparado, adicionamos uma resposta na lista de respostas selecionadas e a retiramos da lista de respostas disponíveis. Por sua vez, quando o método `onRetornarResposta()` é acionado, fazemos o contrário. Adicionamos a resposta na lista de respostas disponíveis e a retiramos da lista de respostas selecionadas;

- **Linhas 42 a 45:** Como dito anteriormente, teremos a possibilidade de passar o controle de navegação pelas perguntas ao usuário. Internamente, fazemos isso através da navegação pelos índices da lista de perguntas, ou seja, sempre que o usuário clicar no botão avançar, este método é executado e como seu próprio nome sugere, avançamos no índice da lista de perguntas, o que nos dá a possibilidade de carregar informações das próximas questões;

- **Linhas 47 a 49:** Este método faz o oposto do anterior, possibilitando a navegação de forma regressiva pelo índice da lista de perguntas;

- **Linhas 51 a 53:** Implementamos um controle ao avançar pelos índices da lista de perguntas, de forma a evitar erros do tipo `ArrayIndexOutOfBoundsException`, ocasionado pela tentativa de acesso a índices inexistentes;

- **Linhas 55 a 67:** Este trecho de código controla o cronômetro regressivo da avaliação, de modo que o tempo seja decrementado a cada segundo até que ele se esgote. Neste caso, a avaliação é finalizada e o usuário redirecionado para uma página de timeout;

- **Linhas 69 a 76:** Utilizamos um recurso muito interessante do JSF 2.0, o escopo Flash, que serve para transportar informações entre ManagedBeans. Neste exemplo, adicionamos nosso objeto VO no escopo flash do JSF carregado com todas as informações

do simulado e o transportamos para outro ManagedBean, onde será utilizado para avaliar o resultado do teste;

- **Linhas 78 a 84:** O método `timeout()` é executado ao término do tempo do teste e faz um redirecionamento para página `timeout.xhtml`;

- **Linhas 86 a 88:** Este trecho tem a função de retornar o tempo em formato apropriado (hh:MM:ss) a ser exibido na tela;

- **Linhas 90 a 92:** No início da aplicação, efetuamos a carga automática dos dados (como vimos na parte 2 deste curso) e com isso cadastramos algumas avaliações no banco de dados. Ao iniciar a execução desta classe, recuperaremos todas as avaliações cadastradas por intermédio do Hibernate e as listamos em um combobox para que o usuário seja capaz de selecionar a que deseja executar.

Nosso próximo passo será codificar a página que fará uso do ManagedBean criado na listagem anterior. Para isso, crie o arquivo `simulador.xhtml` dentro da pasta `avaliacao` e modifique o código gerado pelo Eclipse para que fique igual ao da [Listagem 13](#).

Vejamos os trechos de maior destaque desse código:

- **Linha 16:** Utilizamos a tag `<p:selectOneMenu />` para listar as avaliações cadastradas para o teste;

- **Linhas 26 a 33:** Painel de navegação do simulador, implementado através de botões que executam funções como iniciar, voltar, avançar e finalizar;

- **Linhas 35 a 66:** Painel de visualização do simulado. Nele apresentamos informações como o nome da avaliação, a dificuldade da pergunta, o tipo da questão, o tema, o número da questão atual, o tempo restante do simulado, entre outras;

- **Linhas 68 a 75:** Painel de visualização das perguntas. Nele exibimos o enunciado da pergunta;

- **Linhas 77 a 91:** Painel de exibição das respostas. Nele renderizamos as respostas de acordo com o tipo da questão, como mencionado anteriormente. As respostas podem ser exibidas na forma de marcações simples, marcações múltiplas ou arrastar e soltar.

Criando o ManagedBean e a página de resultado das avaliações

Agora que concluímos a implementação do simulador, nosso próximo passo será construir a página de resultados, onde serão exibidas informações como taxas de erro e acerto, tempo total do teste, entre outras. O funcionamento desta parte do simulador acontece da seguinte maneira: assim que a classe `ResultadoAvaliacaoMB` inicia sua execução, extraímos do escopo flash o VO adicionado pelo ManagedBean `SimuladorMB` que carrega as informações relacionadas ao teste. Em seguida, efetuamos a contabilização das respostas certas e erradas. Concluída esta etapa, o sistema apresenta um resumo informativo com os dados provenientes da avaliação e uma opção para imprimir um relatório contendo o histórico de todos os testes executados pelo simulador.

Para que as funcionalidades de avaliação e impressão de resultados sejam adicionadas nesta aplicação, daremos continuidade ao desenvolvimento da última parte do nosso simulador. Sendo assim, crie uma classe com o nome `ResultadoAvaliacaoMB` dentro do pacote `br.com.javamagazine.simulador.view`. Seu código é apresentado na [Listagem 14](#).

Listagem 13. Código da página simulador.xhtml

```
01 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
02     xmlns:f="http://java.sun.com/jsf/core"
03     xmlns:p="http://primefaces.org/ui"
04     xmlns:sh="http://java.sun.com/jsf/html"
05     xmlns:ui="http://java.sun.com/jsf/facelets"
06     template="/template/main.xhtml">
07
08     <ui:define name="body">
09         <div id="content">
10             <h:form id="form" prependId="true">
11                 <p:outputPanel>
12                     <p:fieldset>
13                         <h:panelGrid columns="2">
14                             <p:outputLabel value="Escolha a Avaliação:"/>
15                             <p:outputPanel id="panelAvaliacao">
16                                 <p:selectOneMenu style="width:245px"
17                                     converter="domainConverter" valueChangeListener=
18                                     "#{simuladorBean.carregarAvaliacao}"
19                                     disabled="#{simuladorBean.iniciar}">
20                                     <p:ajax event="change" update="aceos resumo panelPergunta
21                                         tipoQuestao"/>
22                                     <f:selectItem itemLabel="Selecione..." itemValue="" />
23                                     <f:selectItems value="#{simuladorBean.listaAvaliacao}"
24                                         var="avaliacao" itemLabel="#{avaliacao.descricao}"
25                                         itemValue="#{avaliacao}"/>
26                                     </p:selectOneMenu>
27                                 </p:outputPanel>
28                         </h:panelGrid>
29                     </p:fieldset>
30                     </p:outputPanel>
31
32                     <p:outputPanel id="aceos" style="text-align:center; margin-top:10px">
33                         <p:fieldset rendered="#{simuladorBean.avaliacaoVO.avaliacao.id ne null}">
34                             <p:commandButton id="iniciar" value="Iniciar" action=
35                                 "#{simuladorBean.setIniciar(true)}" update="panelAvaliacao aceos
36                                 resumo panelPergunta panelQuestoes"
37                                 disabled="#{simuladorBean.iniciar}" />
38                             <p:commandButton id="anterior" value="Anterior"
39                                 action="#{simuladorBean.retornar}" update="anterior proximo
40                                 resumo panelPergunta panelQuestoes" disabled="#{simuladorBean.
41                                     indice eq 0 or !simuladorBean.iniciar}" />
42                             <p:commandButton id="proximo" value="Próximo"
43                                 action="#{simuladorBean.avancar}" update="proximo anterior
44                                 resumo panelPergunta panelQuestoes"
45                                 disabled="#{simuladorBean.desabilitarAvancar()}" />
46                             <p:commandButton id="finalizar" value="Finalizar"
47                                 action="#{simuladorBean.finalizar}" ajax="false"
48                                 update="aceos resumo panelPergunta panelQuestoes"
49                                 disabled="#{!simuladorBean.iniciar}" />
50                         </p:fieldset>
51                     </p:outputPanel>
52
53                     <p:outputPanel id="resumo" style="margin-top:10px">
54                         <p:fieldset rendered="#{simuladorBean.avaliacaoVO.avaliacao.id ne null}">
55                             <h:panelGrid columns="8">
56                                 <p:outputLabel value="Nome:" style="font-weight:bold;" />
57                                 <p:outputLabel value="#{simuladorBean.avaliacaoVO.avaliacao.
58                                     descricao}" />
59                                 <p:outputLabel value="Dificuldade:" style="font-weight:bold;" />
60                                 <p:outputLabel value="#{simuladorBean.avaliacaoVO.
61                                     pergunta.nivel.descricao}" />
62                                 <p:outputLabel value="Tipo Questão:" style="font-weight:bold;" />
63                                 <p:outputLabel value="#{simuladorBean.avaliacaoVO.pergunta.
64                                     tipoQuestao.descricao}" />
65                             </h:panelGrid>
66                         </p:fieldset>
67                     </p:outputPanel>
68
69                     <p:outputPanel id="panelPergunta" style="margin-top:10px">
70                         <p:fieldset rendered="#{simuladorBean.avaliacaoVO.
71                             avaliacao.id ne null}">
72                             <h:panelGrid columns="2">
73                                 <p:outputLabel value="Pergunta:" style="font-weight:bold;" />
74                                 <p:outputLabel value="#{simuladorBean.avaliacaoVO.pergunta.
75                                     descricao}" />
76                             </h:panelGrid>
77                         </p:fieldset>
78                     </p:outputPanel>
79
80                     <p:outputPanel id="tipoQuestao" style="margin-top:10px">
81                         <p:outputPanel id="panelQuestoes"
82                             rendered="#{simuladorBean.avaliacaoVO.avaliacao.id ne null}">
83                             <p:outputPanel id="panelSimples"
84                                 rendered="#{simuladorBean.avaliacaoVO.
85                                     pergunta.tipoQuestao.codigo eq 'MS'}">
86                                 <ui:include src="/tipos_questoes/simples.xhtml" />
87                             </p:outputPanel>
88
89                             <p:outputPanel id="panelMultiplo" rendered="#{simuladorBean.
90                                 avaliacaoVO.pergunta.tipoQuestao.codigo eq 'MM'}">
91                                 <ui:include src="/tipos_questoes/multipla.xhtml" />
92                             </p:outputPanel>
93
94                         </div>
95                     </ui:define>
96                 </ui:composition>
```

Os principais pontos desta listagem estão nas seguintes linhas, comentadas em seguida:

- **Linhas 14 a 19:** Recuperamos o estado do objeto VO recebido através do escopo flash e o atribuímos à variável de classe **simuladorVO**. Desta forma será possível acessar as informações do teste executado pelo usuário;
- **Linhas 21 a 38:** Neste método, **avaliarSimulado()**, verificamos a quantidade de respostas corretas e erradas;

- **Linhas 73 a 75:** Como seu próprio nome diz, o método **calcularPorcentagem()** realiza o cálculo percentual com base na quantidade de erros e acertos do estudante;
- **Linhas 77 a 81:** Através do método **prepararRelatorio()** realizamos a persistência de um objeto do tipo **ResultadoAvaliacao**. Com isso temos a possibilidade de exibir, através do relatório, o histórico geral das avaliações efetuadas no simulador e também apresentar percentuais gerais de acertos e erros;

Listagem 14. Código da classe ResultadoAvaliacaoMB.

```

01 package br.com.javamagazine.simulador.view;
02 // imports omitidos;
03 @ViewScoped
04 @ManagedBean(name="resultadoAvaliacaoBean")
05 public class ResultadoAvaliacaoMB implements Serializable {
06     private static final long serialVersionUID = 366271030208991605L;
07     private PerguntaBC perguntaBC;
08     private ResultadoAvaliacaoBC resultadoAvaliacaoBC;
09     private AvaliacaoVO simuladorVO;
10    private int totalCorretas;
11    private int totalErradas;
12    private boolean imprimir;
13
14    @PostConstruct
15    public void init() {
16        setSimuladorVO((AvaliacaoVO) FacesContext.getCurrentInstance()
17            .getExternalContext().getFlash().get(ConstantesEnum.CHAVE_FLASH
18            .getValor().toString()));
19        this.perguntaBC = new PerguntaBCImpl();
20        this.resultadoAvaliacaoBC = new ResultadoAvaliacaoBCImpl();
21    }
22
23    public void avaliarSimulado() {
24        setTotalCorretas(0);
25        setTotalErradas(0);
26        for (Pergunta pergunta : this.getSimuladorVO().getAvaliacao()
27            .getListaPergunta()) {
28            switch (pergunta.getTipoQuestao()) {
29                case MARCAR_SIMPLES:
30                    this.contabilizarQuestaoSimples(pergunta);
31                    break;
32                case MARCAR_MULTIPLAS:
33                    pergunta.setContemRespostasErradas
34                        (this.contabilizarQuestaoMultipla(pergunta, pergunta));
35                    break;
36            }
37            setImprimir(true);
38        }
39
40        private void contabilizarQuestaoSimples(Pergunta pergunta) {
41            setTotalCorretas(pergunta.getRespostaSimples() != null && pergunta.
42                getRespostaSimples().isCorreta() ? getTotalCorretas()
43                    + 1 : getTotalCorretas());
44
45        private boolean contabilizarQuestaoMultipla
46            (Pergunta perguntaOriginal, Pergunta perguntaRespondida) {
47            boolean contemRespostasErradas = false;
48            if (this.obterQuantidadeRespostasCorretas(perguntaOriginal
49                .getListaResposta()) == perguntaRespondida.getRespostasSelecionadas()
50                    .size()) {
51                boolean correta = false;
52                for (Resposta resposta : perguntaRespondida.getRespostasSelecionadas()) {
53                    correta = resposta.isCorreta() ? true : false;
54                    if (!correta) {
55                        contemRespostasErradas = true;
56                    }
57                }
58            }
59            setTotalCorretas(correta ? getTotalCorretas() + 1 : getTotalCorretas());
60            setTotalErradas(correta ? getTotalErradas() : getTotalErradas() + 1);
61        } else {
62            setTotalErradas(getTotalErradas() + 1);
63            contemRespostasErradas = true;
64        }
65        return contemRespostasErradas;
66    }
67
68    private int obterQuantidadeRespostasCorretas(List<Resposta> listaResposta) {
69        int corretas = 0;
70        for (Resposta resposta : listaResposta) {
71            corretas = resposta.isCorreta() ? corretas + 1 : corretas;
72        }
73        return corretas;
74    }
75
76    public Double calcularPorcentagem() {
77        return (double)getTotalCorretas() / (double)getResultadoAvaliacao()
78            .getAvaliacao().getListaPergunta().size() * 100;
79    }
80
81    public void prepararRelatorio() {
82        ResultadoAvaliacao resultadoAvaliacao = new ResultadoAvaliacao
83            (getSimuladorVO().getAvaliacao(), new Date(), calcularPorcentagem(),
84            getTotalCorretas(), getTotalErradas());
85        this.getResultadoAvaliacaoBC().salvar(resultadoAvaliacao);
86        setImprimir(false);
87    }
88
89    public void imprimir() throws Exception {
90        Relatorio relatorio = new Relatorio(new JRDataSourcePagingacao
91            (ResultadoAvaliacao.LISTAR_TODOS), new HashMap<String, Object>()
92            {private static final long serialVersionUID = -1947818225327501326L;
93                {put(ConstantesEnum.CHAVE_TOTAL_CORRETAS.getValor().
94                    toString(), getTotalCorretas());
95                put(ConstantesEnum.CHAVE_TOTAL_ERRADAS.getValor().
96                    toString(), getTotalErradas());
97                put(ConstantesEnum.CHAVE_CABECALHO_RELATORIO
98                    .getValor().toString(), ConstantesEnum.TITULO_RELATORIO
99                    .getValor().toString());
100               }, ConstantesEnum.NOME_RELATORIO.getValor().toString());
101           new GeradorRelatorio().gerarRelatorio(relatorio);
102       }
103   }
104
105   // getters and setters omitidos
106 }
```

- Linhas 83 a 90:** A impressão do relatório é realizada por intermédio do método **imprimir()**. Para que a impressão funcione como esperado, devemos preparar as estruturas necessárias para instanciar um objeto do tipo **Relatorio**, como um objeto do tipo **JRDataSource**, um objeto do tipo **HashMap**, que conterá os parâmetros necessários para o relatório, como o total de respostas certas e erradas e, finalmente, um objeto do tipo **String**, contendo o nome do relatório.

Nota

Para que a impressão do relatório funcione adequadamente, precisamos adicionar o arquivo Jasper do relatório no classpath da aplicação. Para isso, crie uma pasta de nome “relatorio” e adicione o arquivo RelatorioAvaliacao.jasper, que pode ser baixado diretamente na página desta edição. Realizada essa etapa, precisamos criar em relatorio uma nova pasta, de nome “virtual”. Ela será necessária para que nosso mecanismo de paginação implementado na parte um deste artigo funcione adequadamente.

Para finalizar a implementação do simulador quiz, nosso próximo e último passo é codificar a página que acessará o ManagedBean de resultados. Desta forma, crie o arquivo *resultado.xhtml* e atualize seu código com o apresentado nas **Listagens 15 e 16**, que resultam na janela exposta na **Figura 4**.

Dentre os pontos desta listagem, vale destacar:

- Linhas 7 a 11:** Declaramos uma função JavaScript para acionar a impressão do relatório de forma automática, após persistir o resultado da avaliação na base de dados;

- Linhas 13 a 39:** Painel para apresentação das informações relacionadas ao simulado, como o tema, a quantidade de questões, a data em que a avaliação foi efetuada, entre outros;

- Linhas 40 a 46:** Painel que exibe os botões de ação *Avaliar* e *Imprimir*. Na primeira opção executamos a avaliação das respostas, contabilizando os erros e acertos. Já no segundo caso efetuamos a persistência do resultado da avaliação e acionamos a impressão do relatório via JavaScript. Note que poderíamos ter utilizado uma solução envolvendo o componente **<p:commandButton />** do PrimeFaces, combinando os atributos **actionListener** e **action**, chamando, respectivamente, os métodos de persistência e impressão do ManagedBean. No entanto, optamos pela solução via JavaScript para apresentar uma maneira alternativa de combinar ações entre o framework JSF e esta linguagem que vem ganhando bastante espaço na construção de interfaces responsivas;

- Linhas 47 a 70:** Utilizamos uma forma tradicional para listagem de dados através do componente **<p:dataList />**. Com ele apresentamos todas as perguntas e suas respectivas respostas, exibindo também o texto “*Errado*” para as respostas erradas e “*Certo*” para as respostas corretas. Para destacar ainda mais, aplicamos um

Listagem 15. Código da página *resultado.xhtml* – Parte 1.

```

01 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
02   xmlns:f="http://java.sun.com/jsf/core"
03   xmlns:p="http://primefaces.org/ui"
04   xmlns:h="http://java.sun.com/jsf/html"
05   xmlns:ui="http://java.sun.com/jsf/facelets"
06   template="/template/main.xhtml">
07
08 <ui:define name="body">
09 <div id="content">
10   <script type="text/javascript">
11     function imprimir(){
12       document.getElementById('form:imprimir').click();
13     }
14   </script>
15   <h:form id="form" preprendId="true">
16     <p:outputPanel style="margin-top:10px">
17       <p:fieldset>
18         <h:panelGrid columns="14">
19           <p:outputLabel value="Nome:" style="font-weight:bold;" />
20           <p:outputLabel value="#{resultadoAvaliacaoBean.simuladorVO
21             .avaliacao.descricao}" />
22           <p:outputLabel value="Tema:" style="font-weight:bold;" />
23           <p:outputLabel value="#{resultadoAvaliacaoBean.simuladorVO
24             .avaliacao.tema.descricao}" />
25           <p:outputLabel value="Questões:" style="font-weight:bold;" />
26           <p:outputLabel value="#{resultadoAvaliacaoBean.simuladorVO
27             .avaliacao.listaPergunta.size()}" />
28           <p:outputLabel value="Data:" style="font-weight:bold;" />
29           <p:outputLabel value="#{resultadoAvaliacaoBean.simuladorVO
30             .dataAvaliacao}" />
31           <p:outputLabel value="Tempo Total:" style="font-weight:bold;" />
32
33 <h:panelGroup>
34   <h:panelGrid columns="2">
35     <p:outputLabel value="#{resultadoAvaliacaoBean
36       .simuladorVO.avaliacao.porcentagemAcerto}" />
37     <f:convertNumber pattern="#000.00"/>
38   </h:panelGrid>
39 </h:panelGroup>
40 <h:panelGrid>
41   <p:outputPanel id="acoes" style="text-align:center; margin-top:10px">
42     <p:fieldset>
43       <p:commandButton value="Avaliar" update="acoes detalhes
44         resultado" actionListener="#{resultadoAvaliacaoBean
45           .avaliarSimulado()}" rendered="#{!resultadoAvaliacaoBean
46             .imprimir()}" />
47       <p:commandButton value="Imprimir" oncomplete="imprimir();"
48         actionListener="#{resultadoAvaliacaoBean.prepararRelatorio()}"
49         rendered="#{resultadoAvaliacaoBean.imprimir()}" />
50       <h:commandButton id="imprimir" action="#{resultadoAvaliacao
51         Bean.imprimir()}" style="display:none;" />
52     </p:fieldset>
53   </p:outputPanel>
54 
```

Relatórios avançados com Hibernate, JasperReports e PrimeFaces – Parte 3

Listagem 16. Código da página resultado.xhtml – Parte 2.

```
47      <p:outputPanel style="margin-top:10px">
48          <p:fieldset legend="Resumo">
49              <h:panelGrid id="detalhes" columns="2">
50                  <p: dataList value="#{resultadoAvaliacaoBean.simuladorVO.
51                      .avaliacao.listaPergunta}" var="pergunta" type="ordered">
52                      <p:outputPanel rendered="#{pergunta.tipoQuestao.codigo
53                          eq 'MS'}">
54                          <p:outputLabel value="P: #{pergunta.descricao}" />
55                          <p:outputLabel value="#{pergunta.respostaSimples.correta ?
56                              'Certo' : 'Errado' }" rendered="#{resultadoAvaliacaoBean.
57                                  .imprimir}" style="color:#{pergunta.respostaSimples.correta ?
58                                      'blue' : 'red' }" />
59                          <p:outputLabel value="R: #{pergunta.respostaSimples
60                              .descricao}" style="font-weight:bold;" />
61                      </p:outputPanel>
62                      <p:outputPanel rendered="#{pergunta.tipoQuestao
63                          .codigo eq 'MM' or pergunta.tipoQuestao.codigo eq 'AS'}">
64                          <p:outputLabel value="P: #{pergunta.descricao}" />
65                          <p:outputLabel value="#{pergunta.contemRespostas
66                              Erradas ? 'Errado' : 'Certo' }" rendered="#{resultadoAvaliacao
67                                  Bean.imprimir}" style="color:#{pergunta.contemRespostas
68                                      Erradas ? 'red' : 'blue' }" />
69                          <ui:repeat var="resposta" value="#{pergunta.respostas
70                              Selecionadas}">
71                              <p:outputLabel value="R: #{resposta.descricao}"
72                                  style="font-weight:bold;" />
73                      </ui:repeat>
74                      <h:panelGroup rendered="#{pergunta.respostasSelecionadas
75                          .isEmpty()}">
76                          <p:outputLabel value="Errado" style="color:red"
77                              rendered="#{resultadoAvaliacaoBean.imprimir}" />
78                          <p:outputLabel value="R:" style="font-weight:bold;" />
79                      </h:panelGroup>
80                  </p: dataList>
81              </h:panelGrid>
82          </p:outputPanel>
83          <p:outputPanel id="resultado" style="margin-top:10px">
84              <p:fieldset rendered="#{resultadoAvaliacaoBean.imprimir}">
85                  <h:panelGrid id="resumo" columns="2">
86                      <p:outputPanel>
87                          <p:outputLabel value="Porcentagem" style="font-weight:bold;" />
88                          <h:panelGroup>
89                              <h:panelGrid columns="2">
90                                  <p:outputLabel value="#{resultadoAvaliacaoBean.
91                                      calcularPorcentagem()}">
92                                      <f:convertNumber pattern="#000.00"/>
93                                  </p:outputLabel>
94                                  <p:outputLabel value="%" style="font-weight:bold;" />
95                              </h:panelGrid>
96                          </h:panelGroup>
97                          <p:outputLabel value="Status" style="font-weight:bold;" />
98                          <p:outputLabel value="#{resultadoAvaliacaoBean.
99                              calcularPorcentagem() ge resultadoAvaliacaoBean.
100                             simuladorVO.avaliacao.porcentagemAcerto ? 'Aprovado'
101                                 : 'Reprovado' }" />
102                          <p:outputLabel value="Total Acertos" style="font-weight:bold;" />
103                          <p:outputLabel value="#{resultadoAvaliacaoBean.
104                              .totalCorretas}" />
105                          <p:outputLabel value="Total Erros" style="font-weight:bold;" />
106                          <p:outputLabel value="#{resultadoAvaliacaoBean.totalErradas}" />
107                      </h:panelGrid>
108                  </p:outputPanel>
109              </h:panelGrid>
110          </p:outputPanel>
111      </p:fieldset>
112  </ui:define>
113  </ui:composition>
```

Figura 4. Tela de relatórios do simulador

estilo vermelho para as respostas erradas e azul para as corretas. Repare que aproveitamos a oportunidade para demonstrar o uso do operador ternário dentro das tags do JSF, sendo este de grande valor para o desenvolvedor ao se deparar com situações em que é necessário apresentar informações ou aplicar estilos diferentes de acordo com alguma condição;

- **Linhas 71 a 95:** Finalmente, apresentamos as informações resumidas do resultado da avaliação, como a porcentagem de acerto e o total de respostas corretas e erradas.

A partir de agora teremos um simulador de quiz funcionando e capaz de receber mudanças e novas funcionalidades sem grandes dificuldades. Para isso, ao longo de três artigos, exploramos recursos presentes no dia a dia de grande parte dos desenvolvedores e apresentamos como os explorar com boas práticas.

Na era da informação, as mudanças se tornaram um requisito presente em qualquer etapa do ciclo de desenvolvimento do software. Desta forma, precisamos projetar aplicativos que estejam aptos a receber modificações

do modo menos impactante possível em seu código e arquitetura. Para isso, o uso de padrões de projeto como MVC e DAO se tornam ainda mais importantes.

Por fim, vale frisar que a capacidade que uma aplicação tem de sofrer alterações com eficiência contribui para sua qualidade. Outra forma de classificar a qualidade de um software é a escolha mais adequada dos frameworks que utilizamos para seu desenvolvimento. Nesta proposta, a adoção de soluções como Hibernate e JSF podem ser um grande diferencial de acordo com o contexto do projeto.

Autor



Marcos Vinicios Turisco Dória

É Analista de Sistemas Java, certificado em OCJA, OCJP e OCWCD, Graduado em Ciência da Computação pela Universidade de Fortaleza - UNIFOR, trabalha na Secretaria de Finanças de Fortaleza – SEFIN e desenvolve há cerca de seis anos.



Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Links:

Página de download do iReport.

<http://community.jaspersoft.com/project/ireport-designer/releases>

Página de download do Eclipse Luna.

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

Página de download do Tomcat.

<http://tomcat.apache.org/download-70.cgi>

Página de download do MySQL.

<https://www.mysql.com/downloads/>

JSF Reference Documentation.

<http://www.javaserverfaces.org/documentation>

PrimeFaces Reference Documentation.

<http://www.primefaces.org/documentation>

PrimeFaces Components.

<http://www.primefaces.org/showcase/>



Requisitos e Casos de Uso x User Story

Uma abordagem comparativa das técnicas de levantamento de requisitos tradicionais e ágeis

Em algum momento na carreira de um analista de negócios, analista de sistemas ou desenvolvedor, eles acabam se deparando com alguns dilemas ou com a dificuldade em escolher ou entender as diferentes técnicas para registrar os pedidos dos usuários. Independente se você pratica uma metodologia de desenvolvimento de software mais formal ou mais ágil, a qualidade do que se entrega é fundamental. Porém, o que é qualidade senão entregar o que foi solicitado e especificado para atender uma necessidade ou resolver um problema do usuário.

Pensando nisso, não podemos evitar de ter que adotar umas práticas de descoberta e registro dessas solicitações e especificações para nos guiar durante o ciclo de software e para garantir que estamos entregando exatamente o que foi acordado entre duas partes, ainda que, obviamente, mudanças possam ocorrer, pois claramente as solicitações e especificações do software vão se tornando mais próximas da solução à medida que avançamos no tempo em um projeto de software.

É natural identificarmos mudanças ou adequações necessárias tanto para que o software funcione conforme esperado quanto para que o nosso usuário saia satisfeito com o produto que recebeu. Agora imagine não ter nenhuma base de referência e deixar isso somente pelo que está na cabeça do desenvolvedor, que pode ter entendido uma coisa completamente diferente do que o usuário está pensando. Uma origem comum dos problemas citados são acordos que usam somente a comunicação verbal e que muitas vezes são falhos.

Ao contrário do que pensamos, a comunicação verbal possui muitas falhas devido ao fato de ter como guia regras de moral ou polidez que impedem um pensamento analítico e estruturado.

Veja, por exemplo, quando tentamos falar com uma pessoa procurando ser extremamente explícitos e diretos no que queremos dizer e como pode soar estranho, quando não, mal-educado: “Preciso que você identifique

Fique por dentro

Conhecer as diferentes técnicas de especificação de requisitos utilizadas atualmente é crucial para sobreviver em um mercado de TI que frequentemente vem exigindo profissionais capacitados e que saibam trabalhar ora em ambientes formais e sob regimento de normas e padrões rigorosos, ora em projetos que usam modelos de metodologias de desenvolvimento ágeis e flexíveis, mas sempre sem perder a qualidade. Esse artigo procura abranger as técnicas mais conhecidas para identificação e registro de requisitos (Especificação de Requisitos de Software, Casos de Uso e User Stories), buscando a essência do assunto e mostrando as diferenças e semelhanças em cada técnica. Ele não tem a intenção de definir uma técnica em detrimento de outra, mas pelo contrário, permitir que o leitor consiga, por si só, aplicar a melhor técnica para o projeto ou processo de desenvolvimento de software mais apropriado, com embasamento teórico e exemplos de aplicação.

os pedidos dos clientes para que você possa registrá-los e garantir que você não esquecerá do que foi solicitado pelo cliente e que as demais pessoas entendam o que você está fazendo”.

É óbvio que não falamos dessa maneira, por educação e também para soar menos autoritário e mais amigável. Por outro lado, esse excesso de polidez e até a própria estrutura da língua exige um texto que muitas vezes pode-se tornar ambíguo.

Veja no exemplo a seguir como o texto pode soar estranho: “Maria devolveu a José o seu livro”. De quem era o livro? De Maria ou de José.

Observe mais alguns exemplos de ambiguidade:

- “Roberto e Cláudia vão se divorciar”. Um vai se divorciar do outro ou ambos vão se divorciar dos respectivos cônjuges?
- “Eles viram os animais quando estavam presos”. Quem estava preso? “Eles” ou os animais?
- “Frederico encontrou Godofredo na rua e lhe disse que seu primo estava em casa”. O primo de qual dos dois está em casa?

O que acontece em geral é que recebemos instruções ou pedidos de maneira muito informal, sucinta e sem detalhes suficientes para que o entendimento seja o mesmo entre os envolvidos, levando a solicitações com detalhes omitidos ou subentendidos.

Essas omissões podem ter diversos motivos, a saber:

1. O usuário não quer transparecer que não sabe o detalhe;
2. O usuário espera ser questionado quanto ao detalhe, pois acredita que não é sua responsabilidade ter que identificá-lo;
3. O usuário pensa que o desenvolvedor ou analista já conhece o detalhe porque supõe que o mesmo tenha experiência e conhecimento do produto ou do negócio em questão.

Sendo assim, registrar os pedidos e especificações do software que iremos desenvolver passa a ser essencial para evitarmos essas e outras armadilhas de entendimento.

Porém, nada é tão simples. Devido à variedade de técnicas e práticas atuais, questões como “Será que as histórias de usuário (User Stories) podem capturar o comportamento do produto a ser desenvolvido?”, “Requisitos e Casos de Uso são redundantes?” ou “Como posso saber qual a melhor técnica para o meu projeto?” podem surgir e, por consequência, podemos nos sentir perdidos com a vasta literatura disponível para cada técnica, mas poucos debates ou tentativas de contextualizar cada uma delas para que seja possível esclarecer essas e outras questões.

As três técnicas apresentadas aqui as vezes soam sobrepostas ou até mesmo contrárias. Mas se entendermos bem o que são e para que servem, podemos facilmente identificar qual prática melhor se adequa a qual objetivo.

Para isso, vamos entender o que é uma Especificação de Requisito, para que ela serve, qual a sua relação com os Casos de Uso e qual é o seu lugar junto às Histórias de Usuário (User Stories).

Especificação de Requisitos

Na Engenharia de Software, a especificação de requisitos é a principal técnica para obter qualidade, ou seja, atender o que foi solicitado pelo cliente. Os requisitos são o princípio e a base para tudo o que está relacionado ao ciclo de vida do desenvolvimento do projeto. Do início ao fim do desenvolvimento, são eles que vão: definir o que foi solicitado; guiar a construção do sistema em questão; e servir de base para o acompanhamento, validação e verificação do produto entregue.

Como representação formal do que é um requisito, a norma IEEE Std 830-1998 estabelece que: “... é a especificação de produto de software, programa ou conjunto de programas em particular que executam funções específicas em um ambiente específico.”.

Em relação à definição do que é qualidade, vejamos a especificação formal da NBR ISO 9000:2005: “... é o grau no qual um conjunto de características satisfaz a requisitos”.

O IEEE Std 830-1998 define detalhadamente quais são as características de uma boa especificação de requisitos e quais os tipos e categorias dos requisitos de software existentes. Essas características são:

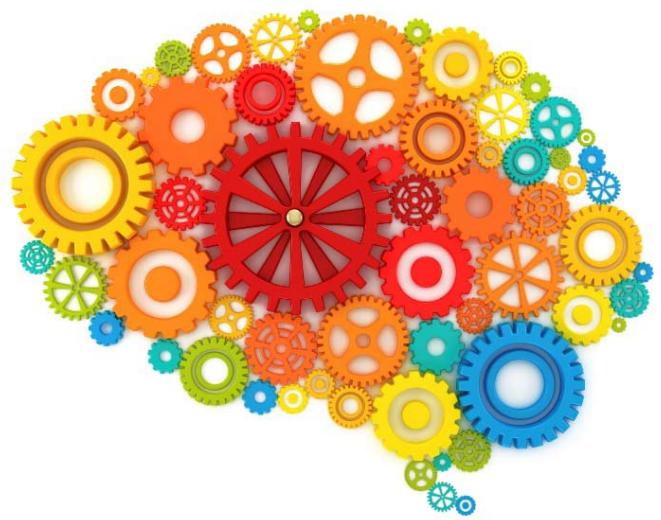
- **Correto:** a especificação dos requisitos deve exprimir corretamente tudo o que o software deve fazer ou qual capacidade o software deve ter;
- **Não ambíguo:** o requisito não dá margem para mais de uma interpretação;
- **Completo:** todos os requisitos exprimem a capacidade ou necessidade por completo;
- **Consistente:** um requisito não é contraditório com outro;
- **Classificado por importância e risco:** possui um valor atribuído que indique a importância para o negócio e um valor para indicar o risco de o requisito mudar;
- **Verificável:** é possível testar o requisito;
- **Modificável:** a especificação de requisitos permite ser alterada de maneira completa, fácil e consistente;
- **Rastreável:** possui um mecanismo de rastrear todos os documentos e artefatos produzidos a partir do requisito.

Na prática, porém, não há muitas orientações de como escrever um bom requisito de software. Alguns exemplos de escrita de requisitos de software expressam os requisitos como sentenças simples utilizando verbos como **deve** para requisitos obrigatórios e **deveria** para requisitos opcionais.

Vejamos alguns exemplos de requisitos de software:

1. O sistema deve apresentar um alerta visual em verde ao término do processamento para sinalizar ao usuário que o processamento foi concluído;
2. O sistema deveria criar um registro de log em um arquivo contendo a data da execução e o resultado do processamento para fins de auditoria.

Ainda que a descrição de um requisito de software possua todos os atributos definidos pelo padrão IEEE Std 830-1998, fica difícil indicar dentro da sentença quais partes evidenciam essas propriedades. De uma maneira mais simples, como sabemos se o texto do requisito está bem escrito?



Requisitos e Casos de Uso x User Story

Para garantir uma definição estruturada, e que nos permita identificar todas as propriedades de um requisito, seria necessário especificar um padrão de escrita no formato de um formulário para preenchimento dos devidos atributos. Um exemplo desse tipo de escrita de requisitos costuma ter as seguintes informações e é apresentado a seguir:

- **Número:** uma numeração sequencial que identifica unicamente o requisito e permite fazer uma rastreabilidade para outros requisitos e demais artefatos do desenvolvimento;
- **Título:** um título que resuma o conteúdo do requisito;
- **Importância:** indica qual a importância para o negócio ou o quanto o objetivo do negócio será atingido por esse requisito. Pode ser uma escala subjetiva como alta, média ou baixa ou uma escala mais precisa que indique de forma quantificada o quanto esse requisito é importante para o negócio;
- **Complexidade:** indica a estimativa de complexidade do requisito da perspectiva técnica;
- **Risco:** indica o nível de risco de o requisito mudar;
- **Beneficiário:** solicitante ou usuário final que será beneficiado pelo recurso que o software deve prover;
- **Cenário atual:** uma pequena descrição do problema que o usuário possui e que precisa ser resolvido pelo software;
- **Objetivo ou Benefício:** o motivo ou motivos pelos quais o recurso foi solicitado que justifica sua existência;
- **Solução proposta:** o recurso que o software deve possuir para poder resolver o problema do usuário.

Exemplo de Requisito de Software no estilo formulário

Número: 01025

Título: Informar o protocolo de atendimento.

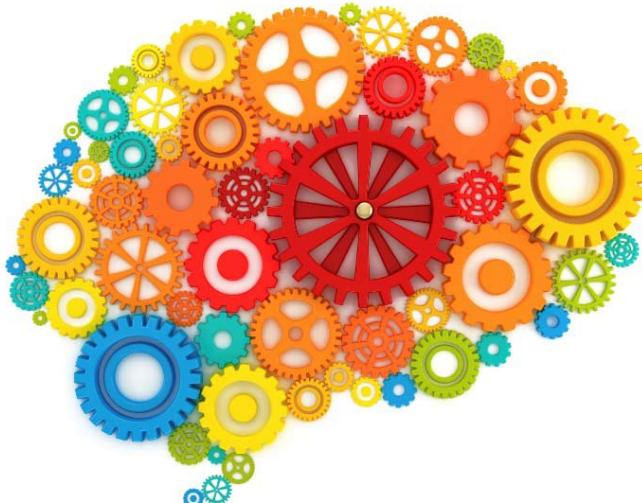
Importância: Alta.

Complexidade: Baixa.

Risco: Médio.

Beneficiário: Analista do Service Desk.

Cenário atual: Atualmente o sistema Zeus não apresenta o



número do protocolo de atendimento gerado no início do atendimento ao cliente para o usuário.

Objetivo ou Benefício: O Analista do Service Desk necessita visualizar o protocolo de atendimento para informar para o cliente a qualquer momento durante a ligação caso o cliente solicite o protocolo novamente.

Solução proposta: O sistema deve exibir o número do protocolo de atendimento de forma destacada na posição superior da tela de atendimento do Service Desk.

Comparando as duas abordagens percebemos que a primeira é sucinta e mais simples, porém sem diversas informações que nos garantem facilmente as propriedades de um bom requisito. Já na segunda abordagem temos todos os detalhes e motivadores para entender melhor o que foi solicitado, porém em uma forma muito mais trabalhosa.

Requisitos devem ser descritos em linguagem natural, divididos geralmente em:

- **Requisitos Funcionais:** Os requisitos funcionais descrevem as diversas funções que Clientes e Usuários necessitam que o software atenda. Eles definem as funcionalidades desejadas no software. A especificação de um requisito funcional deve determinar o que se espera que o software faça, sem a preocupação de como ele faz;

- **Requisitos Não-funcionais:** São aqueles que restringem o comportamento do software, ou que determinam qualidades e capacidades do produto. Definem qualidades gerais sobre o sistema e estão relacionados com restrições sobre como os requisitos dos usuários devem ser satisfeitos (custo, prazo, tecnologia, etc.). São subjetivos, sujeitos a diferentes interpretações e perspectivas, relativos, pois dependem de cada sistema, e interativos, pois interagem entre si, afetando em maior ou menor intensidade os demais requisitos.

Os requisitos são uma ótima base para priorizar, estimar, acompanhar, controlar, gerenciar e validar o software a ser desenvolvido e podem ainda ser rastreados para requisitos de usuário (ou requisitos que indicam qual o problema operacional ele quer resolver) e requisitos de negócio (requisitos que exprimem qual o problema de negócio o software irá resolver). Geralmente usa-se ferramentas de gestão de requisitos para registrá-los ou, na ausência de ferramentas mais específicas, templates em Word ou Excel (vide Figura 1).

No entanto, ainda ficam pendentes algumas questões:

- Qual o contexto de uso do requisito, isto é, como podemos entender como a função solicitada será usada? A descrição do requisito não demonstra o uso da função solicitada;
- Em que momento, dentro do software, os requisitos (ou seja, o que foi solicitado) devem ser acionados ou exibidos? Ou, como a função deve ser ativada? A descrição do requisito não informa onde nem quando o requisito será utilizado ou invocado dentro da aplicação;
- Quais situações alternativas ou cenários de exceção? Quais são as situações de uso da função que podem gerar um desvio

Dados do Projeto			
Nome do Projeto:	[Informar o nome do projeto. Exemplo: Marketplace Reverso]		
Número da Demanda:	[Informar o numero da demanda que deu origem ao documento]		
Área de Negócio Envolvidas/Impactadas			
Área de Negócio	Envolvido	Envolvimento	
[informar o nome da área ou departamento de negócio envolvida]	[informar o nome do envolvido pela area de negócio]	[informar qual o papel do envolvido no projeto. Ex. Solicitante, Sponsor, Impactado, Informado, Consultado, Aprovador, etc.]	
Requisitos do Usuário (Desejo)			
Número do Requisito	Nome do Requisito	Descrição	Prioridade*
[Número do requisito de usuário no padrão RFXXX.]	[Nome do requisito em linguagem de negócios.]	[Descrever quais funções foram solicitadas, qual alteração no comportamento do sistema é desejado pelo solicitante. Utilizar o padrão de escrita de estória do usuário. Como um <Beneficiário> posso/gostaria/devo <função> para/de <resultado para o negócio>. Exemplo: Como um Cliente gostaria de Visualizar os produtos de uma categoria para Incluir no carrinho de compras.]	[informar a ordem de prioridade.]

* 1 = Imprescindível; 2 = Importante; 3 = Pouco Importante; 4 = Desejável

Figura 1. Exemplo de template de Requisito de Software

condicional ou um tratamento de uma adversidade? O texto do requisito não trata alternativas e exceções.

Ou seja, a especificação dos requisitos em si ainda parece ser insuficiente para identificarmos tudo o que o software deve fazer. Para preencher essa lacuna, os casos de uso seriam um ótimo artefato para poder capturar o que estaria faltando nos requisitos.

Casos de Uso

Os casos de uso permitem criar um contexto para os requisitos, isto é, descrevê-los de maneira que fique fácil entender como ele será usado, em quais situações e qual o comportamento esperado pelo sistema quando um ou mais requisitos estão em uso.

Para entender melhor, um diagrama de Casos de Uso da UML mostra como o Requisito de Software é atendido. Na Figura 2 usamos esse diagrama para demonstrar que o pedido do cliente “Informar o protocolo de atendimento” será atendido por uma funcionalidade do sistema chamada Atender Cliente, ou seja, ele transforma o desejo ou pedido do usuário em algo mais concreto para podermos analisar de que maneira esse requisito seria utilizado.

O diagrama de casos de uso é um desenho que permite identificar os atores (entidades externas que utilizam o sistema) e todos os casos de uso (ou funcionalidades) envolvidos no escopo do projeto do software que estamos construindo.

Utilizar um diagrama que demonstra quais requisitos cada caso de uso está atendendo facilita entender se estamos entregando todo o escopo do projeto. Para complementar o detalhe faltante dos requisitos, esse artefato ainda possui uma especificação estruturada (fluxos principais, fluxos alternativos e fluxos de exceção). Essa especificação permite descrever como usar o requisito e, assim, descobrir novas situações de uso, como tratar erros, desvios

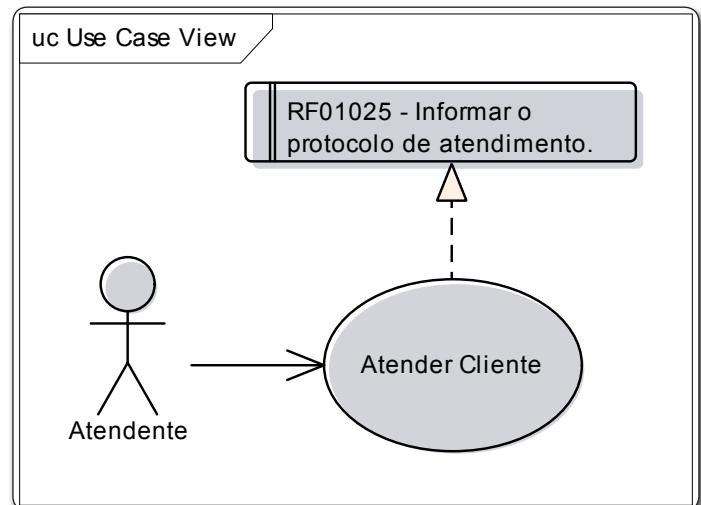
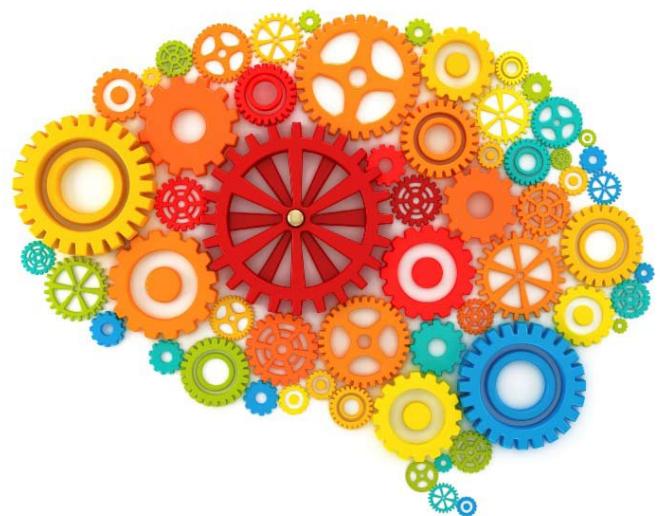


Figura 2. Diagrama de Caso de Uso com Rastreabilidade para Requisitos



e situações adversas que podem surgir quando o usuário estiver interagindo com o sistema.

No exemplo a seguir temos uma indicação de quais situações o requisito “Informar o protocolo de atendimento” pode ser utilizado:

Caso de Uso – Atender Cliente

Fluxo Principal – Realizar Primeiro Atendimento

1. O atendente inicia o atendimento;
 2. O sistema exibe o protocolo de atendimento (Req. 01025);
 3. O atendente informa os dados do cliente (RG, CPF e nome);
 4. O sistema localiza e valida os dados do cliente;
 5. O sistema apresenta as informações detalhadas (contrato, endereço, etc.) do cliente na tela de atendimento;
 6. O atendente informa o protocolo de atendimento (Req. 01025).
- N ... Continuação do caso de uso

Fluxo Alternativo – Solicitar Protocolo de Atendimento

1. A qualquer momento o cliente pode solicitar novamente o protocolo do chamado vigente durante o atendimento.
2. O atendente deve informar ao cliente o protocolo exibido na tela de atendimento (Req. 01025).

É possível observar neste exemplo que o caso de uso permite identificar em quais passos o requisito é referenciado e também quando são utilizados. Além disso, permite especificar de forma detalhada como a função incluída/alterada no sistema para atender ao requisito será usada.

Comparando o exemplo do formulário de requisito com o caso de uso, temos uma visão de que ambas as técnicas são complementares. Os requisitos expressam de maneira clara e objetiva o que o usuário solicitou, de forma que rapidamente podemos validar nosso entendimento com ele sobre o pedido, ao passo que o caso de uso diz como o pedido será implementado.



Ao criarmos o caso de uso colocamos os requisitos em um contexto de uso, detalhando como esse requisito se encaixa nas funções do sistema, qual sua localização e como ele pode ser utilizado, o que complementa e valida o pedido do requisito de software. Em consequência disso, temos um artefato que nos proporciona uma validação mais detalhada do nosso entendimento junto ao usuário solicitante, além de permitir a construção de casos de testes e servir de guia para demais artefatos de modelagem e codificação e finalmente o manual do usuário.

Os fluxos alternativos e fluxos de exceção em um caso de uso permitem ainda que possamos imaginar situações adversas ou alternativas de uso do requisito e, assim, ter uma visão mais completa de como ele pode ser usado. Ademais, o caso de uso também adota uma linguagem não técnica e permite uma comunicação mais clara com o usuário solicitante, gerentes de projeto, equipe de desenvolvimento e demais envolvidos.

Existe ainda um pequeno gap de refinamento que seriam os detalhes da interface do usuário. Para validar questões como a apresentação, o layout, a identidade visual, o posicionamento e formatação de campos e controles, o caso de uso não é o artefato mais apropriado. Para isso usamos um Wireframe, ou protótipo estático para questões de layout (vide **Figura 3**).

Com o uso de Wireframes e Protótipos Estáticos ou Navegáveis podemos deixar os casos de uso e os requisitos livres de detalhes da interface do usuário. Isso permite que a necessidade ou capacidade solicitada pelo usuário seja pura e independente de tecnologia ou interface. A grande vantagem nisso é primeiro poder entender bem qual o problema do negócio do usuário e segundo permitir uma flexibilidade de soluções de interface que podem ser capturadas separadamente sem impactar nos casos de uso e requisitos que podem ficar desacoplados da solução da interface do usuário.

Como vimos, os requisitos sozinhos não fornecem detalhamento suficiente para a construção de uma solução, deixando de fora como a solução será utilizada, quais são as alternativas de uso, quais os tratamentos de exceção e questões de interface do usuário, o que nos leva a recorrer a outros artefatos para podermos detalhar ou refinar melhor o entendimento do que foi solicitado.

User Stories

Histórias de usuário são tipicamente utilizadas em processos de desenvolvimento de softwares ágeis, visando estabelecer uma maneira simples de capturar os desejos dos usuários, rapidamente transcrevê-los e utilizá-los durante o ciclo de vida como base para a construção do software.

Assim como nas especificações de requisitos, a preocupação ou o foco é na qualidade, ou seja, entregar o software conforme o pedido, além de servir de referência para o ciclo de desenvolvimento do que será construído. Porém, a visão dos processos ágeis é que um certo grau de incerteza é permitido, evitando dispensar muito tempo do desenvolvimento em detalhar demais os requisitos, já que esses vão mudar.

As histórias de usuário devem ser escritas em sentenças breves que capturam a **necessidade** do usuário através de funcionalidades, o problema ou **objetivo** que ele quer atingir com essa funcionalidade e quem se beneficia com ela, através de um modelo de escrita padrão.

Em geral, o padrão de escrita de história segue o modelo “Como um <papel> eu posso/gostaria/devo <função> para/de <resultado esperado para o negócio>”. Exemplo: Como um Cliente eu gostaria de **visualizar os planos existentes para decidir qual plano devo comprar.**

A ideia original é que as histórias possam caber em cartões flash ou cartões de biblioteca. Aqueles cartões de papel mais duro e pautado usado em bibliotecas para indexar os livros. A história deve ser escrita na parte da frente do cartão e na parte de traz deve-se escrever os testes ou condições nas quais a história deve ser submetida. Por exemplo: “Testar com cartão MasterCard, Visa e American Express. Testar com CPFs válidos e inválidos. Testar com data de vencimento do cartão expirado”.

Vale ressaltar que outras informações importantes também podem estar presentes no cartão, como a estimativa em pontos de história ou prioridade. Veja um exemplo na Figura 4.

Além das definições anteriores, também é necessário entender quais são os atributos de uma boa história de usuário. Neste contexto, a técnica INVEST, que determina uma forma de escrita de histórias, pode ser utilizada para guiar e validar a qualidade destas. Segundo a INVEST, uma história deve ser:

- Independente: a história deve ser completa o suficiente para permitir sua construção sem depender de outra história;
- Negociável: a história deve ser pequena o suficiente para permitir a negociação com o usuário. Uma história complexa demais pode estar composta de diversas histórias menores e isso dificulta em negociar prazos ou incluir a história no Sprint por ser grande demais;
- Valorizada pelo usuário: a história deve entregar algo de valor para o usuário e não para o desenvolvedor, ou seja, deve exprimir funções que o usuário final ou o comprador do software entendem como algo de valor para seu uso ou negócio;
- Estimável: a história deve permitir que os desenvolvedores possam realizar estimativas de tamanho ou esforço. Histórias grandes demais ou que dificultam a identificação de quais tarefas de desenvolvimento serão necessárias para implementá-la devem ser melhor detalhadas ou quebradas em histórias menores para permitir uma estimativa;
- Simples: a história deve ser simples, sem muitos detalhes. O tamanho da história influencia na dificuldade em estimá-la. Nem

O wireframe mostra a interface de usuário para atender um cliente. No topo, uma barra azul com o título "Atender Cliente". Abaixo, uma barra cinza com o texto "Data: 08/07/2015" e "Protocolo: 010002585641". Um campo para "RG" com o valor "00.000.000-X" e um campo para "CPF" com o valor "000.000.000-00". Um botão "Pesquisar". A seção "Dados do Cliente" contém o nome "Livia Silva Cunha", endereço "Rua Walter Macheroni, 71 Jaboticabal SP 14875466", data de nascimento "Birthday February 17, 1983" e idade "Age 32 years old". A direita, uma caixa amarela com a seguinte legenda: "O protocolo é gerado ao inicio de cada atendimento e apresentado na posição indicada ao lado."

Figura 3. Wireframe da Tela de Atendimento

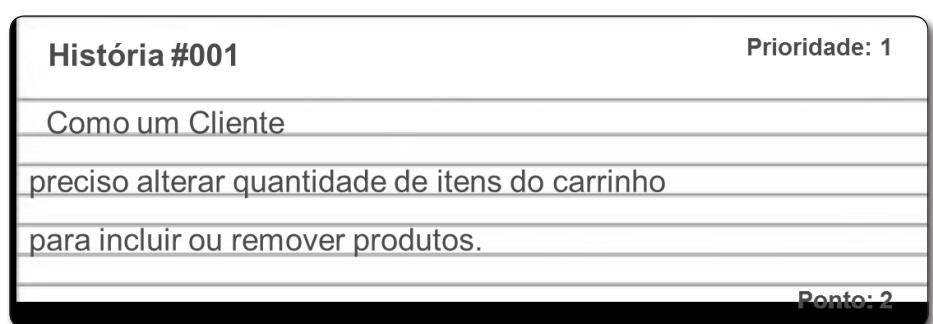


Figura 4. Exemplo de cartão de história do usuário

todo detalhe deve estar incluso em uma história, pois ela deve ser apenas um lembrete do que será construído. Detalhes de regras ou de condições específicas de como a história deve funcionar podem ser expressos como testes na parte de traz da história;

- Testável: a história deve ser testável, ou seja, indicar em quais condições deve ser testada.



Requisitos e Casos de Uso x User Story

Em geral uma história de usuário deve ser simples o suficiente para permitir a construção da funcionalidade, servindo como um lembrete do que foi acordado e negociado, porém sem detalhar excessivamente como a função será usada. Esse detalhe, segundo os processos ágeis, é capturado através de diálogos com o solicitante. Como os processos ágeis preveem ciclos menores para validação com o usuário, este pode dar o feedback e o detalhe necessário através da validação ao final de algumas sprints.

Histórias de usuário, assim como requisitos e casos de uso, não detalham questões de interface com o usuário. Quando o projeto está definindo um sistema totalmente novo e onde todas as discussões com o usuário podem gerar muitas dúvidas ao ponto de comprometer o tempo da validação, talvez seja a hora de adotar também um protótipo ou wireframe, que rapidamente pode contextualizar esses detalhes e permitir seguir com o desenvolvimento no ritmo apropriado.

Algumas recomendações importantes na hora de escrever uma boa história de usuário são:

- Informe um papel na história. Por exemplo, em vez de escrever “Um usuário pode submeter um currículo”, escreva “Um candidato a uma vaga pode submeter um currículo”;
- Escreva para um usuário. Em alguns casos pode ficar subentendido se o usuário pode manipular informações suas ou de outros

usuários. Por isso, escreva histórias de maneira que fique explícito o que um usuário pode fazer em relação aos outros. Exemplo: “Um candidato a uma vaga pode excluir somente o seu currículo”;

- Escrever na voz ativa. Em vez de escrever “Um currículo pode ser submetido pelo candidato”, escreva “Um candidato pode submeter um currículo”;
- Idealmente as histórias devem ser escritas pelos usuários e não pelos desenvolvedores.

Compreendendo as diferenças

Conforme demonstrado, Requisitos e Casos de Uso assumem papéis diferentes e complementares. O requisito de software deve ser sucinto e claro o suficiente para exprimir uma solicitação de uma capacidade que o sistema deve ter ou uma função que ele deve executar. O requisito não detalha como usar essa capacidade ou função. Para sanar essa lacuna, o caso de uso entra como uma solução para tentar capturar o detalhe faltante colocando o requisito em um contexto de uso, permitindo aos desenvolvedores e usuários, a partir de uma língua comum e não técnica, entender e acordar o que deve ser construído. Para refinar ainda mais o entendimento, outros artefatos complementares podem existir como, por exemplo, wireframes e protótipos de tela.

Já as histórias de usuários possuem um foco muito semelhante ao do requisito: servir de base para o que deve ser construído.

Técnica	Para que usar	Quando usar	Quando não usar	Risco de não usar
Especificação de Requisitos de Software	Para identificar os pedidos de funções ou capacidades do software a ser construído. Para permitir estimar o esforço do desenvolvimento. Para permitir a priorização e negociação do que será entregue. Para servir de base para o acompanhamento da evolução e andamento do projeto de software.	Quando o projeto requer um nível de formalismo mais alto nos acordos com os usuários devido à dificuldade ou risco de comprometimento do usuário no projeto. Quando o usuário possui alto grau de conhecimento dos requisitos e sabe exatamente o que o software precisa fazer.	Quando o detalhamento ou formalismo excessivo pode gerar aumento de esforço de desenvolvimento em projetos onde os ciclos de validação são menores para comportar a imprevisibilidade ou chance de mudança do requisito.	Não usar essa técnica implica em não ter o requisito correto ou completo suficiente para uma maior precisão do que será construído.
Casos de Uso	Para permitir detalhar e refinar os requisitos. Para colocar os requisitos em um contexto que permite entender melhor como o usuário enxerga o uso do requisito.	Quando o projeto precisa de maior detalhamento do requisito.	Quando o projeto não pode investir num esforço maior de detalhamento de requisitos devido à imprevisibilidade e risco do requisito mudar constantemente.	Não fazer caso de uso implica em não entender como o requisito de software será usado e gera o risco de entregar uma função que não resolva o problema do usuário.
User Stories	Para servir de lembrete do que foi pedido. Para servir de base para a estimativa. Para possibilitar a negociação e acomodação do requisito de software em ciclos menores de validação.	Quando o projeto não comporta esforço demais de detalhamento de requisito porque possui ciclos pequenos de validação. Quando a variabilidade e o risco de mudança do requisito são muito altos. Quando o envolvimento do usuário e validação do requisito é feito com maior frequência, permitindo colher o feedback e rapidamente ajustar o requisito no próximo ciclo.	Quando o projeto requer um nível de formalismo mais alto nos acordos com os usuários devido à dificuldade ou risco de comprometimento do usuário no projeto.	Não usar implica no risco de não possuir uma base para estimar, conduzir e acompanhar o desenvolvimento e não conseguir comparar o que foi construído contra o que foi solicitado.

Tabela 1. Comparação das técnicas

No entanto, para permitir maior facilidade de negociação e comportar inevitáveis mudanças, não detalha demasiadamente como usar a função solicitada. Esse detalhamento deve ser negociado verbalmente e não capturado em artefatos, tornando o processo mais rápido. É claro que para isso deve-se seguir um processo de desenvolvimento ágil como Scrum, que prevê ciclos menores de entrega e validação, os Sprints. Sem isso, esse detalhe fica perdido e possivelmente se tornará um defeito quando o software estiver totalmente construído.

De maneira mais simples, Requisitos e Casos de Uso são apropriados para processos ou estruturas de empresas onde o nível de formalidade é mais alto devido a características de negócio ou da empresa em questão.

Para facilitar o entendimento, a **Tabela 1** ilustra como e quando utilizar cada técnica.

É importante compreender o uso das técnicas não pensando em qual opção é melhor, mas sim entendendo que cada uma tem sua aplicabilidade em contextos e projetos sob condições específicas.

Qualquer processo de desenvolvimento de software busca entregar produtos com qualidade. Porém, quais são os critérios de qualidade? Sem definir sob quais critérios o software será avaliado, não é possível apontar se o software possui qualidade. Por isso requisitos ou histórias do usuário são importantes para nos ajudar a descobrir e definir esses critérios junto do usuário. Conhecer e entender essas técnicas aumenta a habilidade do profissional de TI envolvido no desenvolvimento, tornando-o mais valioso no mercado por melhor se adaptar aos diversos cenários de projetos e metodologias utilizadas atualmente.

Poder comparar as técnicas, suas características e aplicabilidade facilita na hora da adoção e utilização de uma delas com segurança, pois podemos saber quais problemas essas técnicas nos ajudam a resolver, quando melhor utilizar uma ou outra e qual o seu valor dentro do processo de desenvolvimento.

Autor



José Compadre Junior

jose.compadre@oatsolutions.com.br

Profissional com quinze anos de experiência em TI com certificação em ITIL V3 Foundations, atuando desde o desenvolvimento de sistemas, levantamento de requisitos e processos de negócio até o desenvolvimento, implantação e suporte em metodologias de desenvolvimento de sistemas e processos de qualidade. É formado Bacharel em Administração de Empresas com Ênfase em Sistemas de Informação na FIAP. Atualmente atua na OAT realizando serviços de treinamento e consultoria em Engenharia e Qualidade de Software com apoio de ferramentas. Possui conhecimentos sólidos em Mapeamento de Processos de Negócio com notação BPMN, Engenharia de Software, UML, RUP, ITIL, Cobit, CMMI e Práticas de Gestão de Projetos segundo o PMBOK.



Links:

Associação Brasileira de Normas Técnicas. (2005). ABNT NBR ISO 9000:2005. ABNT.

Bittner, K., & Spence, I. (2002). Use Case Modeling. Addison Wesley.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). The Unified Modeling Language User Guide. Addison-Wesley Professional.

Cohn, M. (2004). User Stories Applied: For Agile Software Development. Addison Wesley.

Jacobson, I., Spence, I., & Bittner, K. (Dezembro de 2011). USE-CASE 2.0 - The Guide to Succeeding with Use Cases. Estados Unidos.

Rubin, K. S. (2013). Essential Scrum - A Practical Guide To The Most Popular Agile Process. Addison Wesley.

Software Engineering Standards Committee of the IEEE Computer Society. (1998). IEEE Std 830-1998 - Recommended Practice for Software Requirements Specifications. New York: The Institute of Electrical and Electronics Engineers, Inc.

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



A Gestão de Mudanças e o Teste de Software

Sob a perspectiva do Teste de Software, saiba como a gestão de mudanças pode auxiliar no direcionamento dos testes

A pressão para desenvolver novos softwares ou melhorar os já existentes vem aumentando consideravelmente nos últimos anos. Neste cenário, o impacto negativo que uma grande falha pode gerar tem recebido ainda mais atenção/importância por parte das organizações. Em função disso, identificar as mudanças que foram implementadas, seja em função de uma nova funcionalidade ou mesmo uma linha de código modificada para atender às solicitações dos usuários ou corrigir um erro, tem grande relevância.

Para viabilizar o controle dessas alterações, torna-se essencial a Gerência de Configuração de Software, um conjunto de atividades de apoio ao ciclo de vida do desenvolvimento que permite a absorção controlada das mudanças inerentes à construção do software, mantendo a estabilidade durante a evolução do projeto.

A Gerência de Configuração é dividida em três atividades, analisadas a seguir:

• **Controle de Versão:** tem como objetivo apoiar as atividades de controle de mudança e integração contínua, fornecendo uma gama de serviços como, por exemplo:

- Identificação, armazenamento e gerenciamento dos itens de configuração e de suas versões durante todo o ciclo de vida do software;
- Histórico de todas as alterações efetuadas nos itens de configuração;
- Criação de rótulos e ramificações no projeto;
- Possibilidade de recuperar uma configuração específica mais antiga do software.

• **Integração Contínua:** tem como objetivo garantir que as mudanças ao longo do projeto sejam desenvolvidas, testadas e relatadas tão logo quanto possível depois de serem introduzidas. Além disso, a integração contínua também permite:

Fique por dentro

As mudanças realizadas ao longo do processo de desenvolvimento de software são inevitáveis, e caso não sejam gerenciadas, podem causar um grande impacto no projeto. Com base nisso, o objetivo desse artigo é mostrar que através da Gestão de Mudanças os riscos podem ser reduzidos e assim, a alteração nos requisitos, no código fonte e o planejamento dos testes tornam-se direcionados e mais objetivos.

- Integrar com frequência as alterações no código, permitindo acompanhar a evolução da aplicação;
- Propicia a utilização de um processo automatizado;
- Possibilita que cada integração seja verificada visando detectar erros de integração o quanto antes;
- Permite a notificação de toda a equipe quando um problema na integração é identificado.

• **Gestão de Mudanças:** fornece um serviço complementar ao oferecido pelo sistema de controle de versão. Tem como foco controlar os procedimentos pelos quais as mudanças de um ou mais itens de configuração são propostas, avaliadas, aceitas e aplicadas.

A partir disso, ao longo deste artigo serão apresentadas algumas características da Gestão de Mudanças, uma aresta da Gestão de Configuração que possibilita um maior controle, e consequentemente, mais qualidade à aplicação, ao longo de todo o processo de desenvolvimento do software sob a perspectiva do Teste de Software.

A Gestão de Mudanças

A Gestão de Mudanças envolve processos, ferramentas e técnicas para gerenciar os vários aspectos envolvidos nas alterações do software, tendo como objetivo facilitar que os resultados previstos sejam alcançados da maneira mais eficaz possível.

Segundo Pressman, em seu livro *Software Engineering: A Practitioner's Approach*, a Gestão de Mudanças é um conjunto de atividades projetadas para controlar as mudanças através da identificação dos produtos de trabalho que serão alterados, estabelecendo um relacionamento entre eles, definindo o mecanismo para o gerenciamento de diferentes versões destes produtos, controlando as mudanças impostas e auditando e relatando as mudanças realizadas.

Quando o enfoque é dado ao software, a Gestão de Mudanças geralmente engloba alguns motivos mais comuns, como a correção de erros, a implementação de melhorias, a criação de novas funcionalidades, alteração de escopo, todos eles mais detalhados ao longo do artigo.

Um fator importante a ser observado durante alguma mudança no software está relacionado ao custo e também ao impacto que devem ser analisados para avaliar quanto do software será afetado pela alteração proposta, bem como quanto poderá custar para desenvolver esta mudança.

Depois de realizada toda a análise relacionada à alteração da aplicação, o ciclo de desenvolvimento do software poderá incorporar as alterações propostas, em efeito cascata, modificando os requisitos, a aplicação e o teste planejado, refletindo assim as mudanças propostas, avaliadas e validadas, possibilitando uma mudança controlada, que não impacte negativamente no software como um todo.

Portanto, o gerenciamento adequado é essencial para o sucesso do projeto, visto que, caso não ocorra, podem surgir inconsistências entre os requisitos e os demais artefatos do projeto.

Um passo importante para agregar mais qualidade ao software desenvolvido é a capacidade de gerenciar as mudanças feitas na aplicação. Normalmente essas alterações abrangem diferentes aspectos, podendo causar um grande problema para o projeto quando não é possível determinar efetivamente a natureza e a localização das mesmas.

De uma maneira geral, a maioria das ferramentas voltadas para a Gestão de Mudanças tende a mostrar a localização dessas mudanças, destacando as funcionalidades afetadas. Por outro lado, geralmente não permite uma visão mais ampla do impacto.

A Gestão de Mudanças para o Teste de Software é de grande importância por diferentes e complementares razões, por exemplo, viabilizar que o ambiente de teste acompanhe adequadamente as mudanças do projeto, garantindo resultados controlados, através da execução dos testes em ambiente equivalente ao ambiente de produção.

O ideal é que somente depois que as modificações tenham sido pontualmente testadas, visando obter um resultado mais imediato sobre a qualidade do software após a alteração, sejam realizados testes adicionais que não estejam relacionados diretamente às funcionalidades que passaram por mudanças como, por exemplo, o Teste de Integração, o Teste de Performance e o Teste de Regressão.

O Teste de Integração deve ser realizado após serem testadas as unidades individualmente do software, verificando se as partes,

quando colocadas para trabalhar em conjunto, não conduzem a erros.

O Teste de Performance, por sua vez, poderá avaliar se o desempenho da aplicação não foi impactado após as alterações realizadas no software. Além disso, poderá verificar também se a funcionalidade desenvolvida irá responder ao esperado, ao simular uma determinada quantidade de usuários simultâneos, de acordo com a expectativa de utilização da aplicação em produção.

Já o Teste de Regressão, automatizado ou manual, deve ser executado sempre que uma nova versão de software for disponibilizada, ou até mesmo quando surgir a necessidade de executar um novo ciclo de testes. O objetivo desse teste é eliminar possíveis gargalos e estabelecer uma base para os testes de regressão futuros, além de evitar que novos defeitos sejam introduzidos na aplicação. Ou seja, visa tornar a aplicação estável de modo que o Processo de Teste definido possa prosseguir sem maiores problemas, e seja cuidadosamente controlado através das análises dos resultados apresentadas após a execução do teste.

O impacto das mudanças no software

Para atender à demanda contínua por evolução, todo software está sujeito a mudanças. Tanto durante o processo de desenvolvimento, quanto durante a manutenção de uma aplicação, há a necessidade de incorporar mudanças.

Essas mudanças, no entanto, podem causar vários impactos ao projeto. Deste modo, para avaliar o que pode acontecer com o software, algumas questões importantes podem auxiliar na análise, por exemplo:

- Quem será afetado pelas alterações?
- Em qual parte do código fonte da aplicação as mudanças precisam ser realizadas?
- Além do código fonte modificado, o que mais será afetado com a mudança?



- É viável realizar todas as mudanças?
- O ideal é que as mudanças sejam realizadas simultaneamente ou separadas?
- Em qual prazo toda alteração no software deverá ser realizada?

Através da Gestão de Mudanças é possível determinar claramente os itens que compõem o software, permitindo testar e certificar quanto à qualidade, desde que geridos corretamente, de todas as alterações realizadas na aplicação durante todo o ciclo de vida do projeto, viabilizando manter o registro de tudo o que está sendo testado. Deste modo, para o teste, a Gestão de Mudanças propiciará inclusive a rastreabilidade e o foco dos testes nas alterações sofridas pela aplicação.

Por exemplo, quando não existe a Gestão de Defeitos e um erro encontrado não é reportado, o problema poderá ser postergado ou até mesmo ir para produção sem o devido ajuste. Com a Gestão de Mudanças não é muito diferente, quando uma mudança é realizada na aplicação sem que seja devidamente controlada. O que ocorre é que a validação incompleta do software poderá possibilitar sérios danos à qualidade do mesmo, dependendo do impacto desta mudança.

Executar uma Gestão de Mudanças é uma tarefa que requer bastante clareza em sua definição, exigindo que o processo seja estipulado de acordo com a realidade da empresa. Para isso, classificar o impacto da alteração pode ser bem relevante, por exemplo:

- **Impacto Pequeno:** a mudança é relativamente trivial e apresenta um risco mínimo para o software;
- **Impacto Substancial:** a mudança exige um esforço significativo e poderia ter um substancial impacto na aplicação;
- **Impacto Grande:** a mudança pode impactar efetivamente funcionalidades críticas do projeto.

Através de uma definição correta de prioridade, será possível também especificar se a mudança em questão é primordial ou se pode esperar por um momento mais oportuno para implementação, por exemplo:

- **Prioridade Baixa:** a mudança pode aguardar um momento mais oportuno para ser implementada. A não implementação não impacta na liberação do software para produção;
- **Prioridade Normal:** quando é recomendável que a alteração seja desenvolvida assim que possível;
- **Prioridade Urgente:** quando a mudança normalmente está relacionada a corrigir problemas graves, que impactam diretamente na utilização do software na produção.

Os tipos de mudanças

Muitos são os tipos de solicitações que podem gerar uma mudança no software, e consequentemente no planejamento dos testes. De uma maneira macro, as estratégias de alterações na aplicação podem englobar as seguintes situações:

- **Manutenção de software:** ocorre quando mudanças são realizadas em resposta à alguma alteração nos requisitos ou correção de erros, porém, as funcionalidades principais do software se mantêm estáveis;
- **Evolução de arquitetura:** esse tipo de mudança é voltado para quando há alteração na arquitetura do software;
- **Reengenharia de software:** essa mudança ocorre quando nenhuma funcionalidade é adicionada ao software e é feita somente uma reorganização ou reestruturação na aplicação com o objetivo de facilitar futuras alterações.

As estratégias voltadas para as mudanças no software podem ser executadas em conjunto ou separadamente. A escolha vai depender da necessidade do projeto. Para o teste, assim como para o desenvolvimento, quanto mais mudanças envolvidas, maior a complexidade do planejamento e execução do escopo definido.

Para exemplificar os tipos de mudanças que podem ocorrer em um software, seguem algumas das razões mais comuns que levam às alterações do projeto:

- Implementação de melhorias;
- Alteração no escopo do projeto;
- Mudanças em leis;
- Correção de erros;
- Criação de novas funcionalidades;
- Adaptação de arquitetura;
- Fatores econômicos;
- Adequação de prazos;
- Diminuição dos riscos do projeto;
- Exigência dos usuários;
- Mudanças tecnológicas.

Além da definição da prioridade e do impacto da alteração no software, é possível qualificar também o tipo de modificação, como:

- **Simples:** voltada à correção de erros no código;



- **Mais extensas:** com o objetivo de corrigir os erros de projeto;
- **Significativas:** com a finalidade de corrigir erros de especificação ou acomodar novos requisitos.

Ainda segundo Pressman, as mudanças em um software podem ser definidas como corretiva, adaptativa, perfectiva e preventiva. Para o Teste de Software, cada uma dessas definições envolvem os seguintes direcionamentos para o planejamento dos testes:

- **Corretiva:** o teste é executado na funcionalidade modificada com o objetivo de avaliar se o problema reportado foi corrigido. Porém, é válido ressaltar que sem a execução conjunta do teste de regressão, não é possível assegurar que, após as alterações necessárias realizadas, outros problemas não foram inseridos;
- **Adaptativa:** nesse caso, o teste tem como objetivo garantir que um software se adaptará às mudanças no ambiente externo, por exemplo, alterações em regras de negócio, leis, mercado, mantendo o mesmo funcionamento;
- **Evolutiva ou Perfectiva:** diante das alterações que o software é submetido, sejam complexas, custosas ou estruturais, o teste voltado para a correção perfectiva visa avaliar se as alterações realizadas para aperfeiçoar a aplicação não impactaram no correto funcionamento das suas funcionalidades;
- **Preventiva:** testar a manutenção preventiva parte do pressuposto de identificar o que poderá gerar algum tipo de erro no software, visando relatar os possíveis problemas para que sejam tratados antes que venham a ocorrer efetivamente em produção. Deste modo, quanto maior o esforço empregado para tornar a aplicação suscetível de ser mantida, menor o custo de manutenção.

Ferramentas para a Gestão de Mudanças

Garantir qualidade ao software significa também minimizar os riscos e deixar o produto final com o menor número de erros possível. Para isso, a gestão de mudanças tem um papel fundamental de auxiliar no controle de todas as alterações realizadas no software.

As ferramentas comerciais geralmente apresentam mais funcionalidades e um maior grau de integração, se comparado com as ferramentas gratuitas. O problema é que o custo de licenciamento muitas vezes restringe, financeiramente, o seu uso, principalmente em pequenas e médias empresas.

Já as ferramentas open source tendem a possuir muitas vantagens, como o custo baixo ou nulo para aquisição, qualidade, segurança, independência e a possibilidade de adequação a necessidades específicas da empresa. Estas características tornam as ferramentas gratuitas consideravelmente relevantes.

O objetivo de uma ferramenta de gestão de mudanças é registrar as alterações levantadas para o projeto, bem como rastrear as modificações e definir métricas e critérios para avaliar o impacto que estas podem causar.

A seguir serão apresentadas algumas das ferramentas voltadas para a gestão de mudanças.

Mantis

Acessado através de um browser, o Mantis é um popular sistema de gerenciamento de defeitos que possui uma baixa curva de aprendizado, sendo fácil de instalar e também customizar.

É uma ferramenta gratuita, disponibilizada sob a licença GNU General Public License (GPL) e escrita em PHP. Como diferencial, possui suporte a diversos bancos de dados, entre eles: MySQL, SQL Server, PostgreSQL e DB2.

O Mantis está entre as cinco ferramentas open source voltadas para gestão de defeitos mais utilizadas, e certamente, sua ampla adoção está diretamente relacionada à usabilidade simples e intuitiva e a capacidade de gerenciar múltiplos projetos.

Algumas características relevantes do Mantis são:

- Possibilita o envio de notificações customizáveis por e-mail;
- Página principal personalizável para cada usuário;
- Permite a exportação de dados para planilhas.

Bugzilla

O Bugzilla é uma das ferramentas de acompanhamento de bugs mais populares entre as soluções open source disponíveis. Inicialmente, no entanto, foi criada para apoiar o desenvolvimento do browser Mozilla.

É uma opção que facilita o monitoramento das atividades dos usuários, e hoje em dia tem sido empregada em um grande número de projetos de software. Além disso, por ser construída sob a Licença Pública Mozilla, pode ser utilizada em diferentes organizações, independentemente do tamanho destas.

Algumas características relevantes do Bugzilla são:

- Possibilita notificações customizáveis por e-mail;
- Permite adicionar informações nos bugs registrados através de e-mail;
- Facilita o monitoramento das atividades dos usuários.

Scarab

O Scarab é um sistema de gestão de defeitos multiplataforma de fácil instalação, sendo totalmente personalizável através de um conjunto de páginas administrativas.

Essa ferramenta possibilita um eficiente controle de permissões de acesso aos projetos gerenciados por ela, além de notificar os usuários por e-mail a cada tarefa recebida.

Apesar de possuir uma documentação básica e pouco amigável, o Scarab possui vantagens bem interessantes para uma ferramenta open source como, por exemplo, a validação automática de campos para verificar a possibilidade de duplicação de uma issue.

Algumas características relevantes do Scarab são:

- Possibilita a criação de vários projetos;
- Facilidade de manutenção e instalação;
- Possibilita exportar e importar o projeto para outras ferramentas de gerenciamento de defeitos a partir de uma interface XML.

Trac

É uma aplicação com estrutura baseada em wiki que possui integração nativa com o SVN (Subversion), oferecendo aos usuários

uma excelente interface para o repositório. Além disso, possui capacidade de integração com outros sistemas de controle de versão.

O Trac possibilita a rastreabilidade das alterações realizadas no software através dos links criados por meio da formatação wiki. Ademais, é uma ferramenta desenvolvida e mantida pela Edgewall Software e escrita na linguagem Python.

Como curiosidade, até meados de 2005 era disponibilizada nos termos da GNU General Public License, e a partir da versão 0.9 passou a ser lançada sob a licença BSD modificada.

Algumas características relevantes do Trac são:

- Sistema de permissões de acesso simplificado;
- Facilidade de customização;
- Suporta diferentes bancos de dados.

JIRA

É uma ferramenta comercial desenvolvida pela empresa Atlassian. Robusta, voltada para a gestão de projetos, mudanças e tarefas, bem como para o acompanhamento de defeitos, destaca-se entre outras ferramentas por sua vocação especial para a cobertura de todas as etapas do ciclo de vida do desenvolvimento e manutenção de sistemas.

O Jira, geralmente utilizado para gerir processos ágeis, é uma ferramenta proprietária que possui disponibilidade de suporte técnico em planos anuais e requer um investimento variável. Por exemplo: para 50 licenças, o custo é de aproximadamente US\$2.400; acima de 2.000 licenças, o custo gira em torno de US\$12.000.

Algumas características relevantes do Jira são:

- Ferramenta web extremamente customizável;
- Possibilita a criação de workflows;
- Permite a integração com outras ferramentas.



FogBugz

Ferramenta comercial, desenvolvida pela Fog Creek Software, voltada para o acompanhamento de projetos ao longo do ciclo de desenvolvimento do software. É capaz de realizar o gerenciamento de bugs, mudanças, planejamento, projetos, colaboração e controle de tempo.

Um diferencial da ferramenta é que ela disponibiliza uma wiki própria, permitindo a criação de documentos e especificações técnicas e funcionais.

Algumas características relevantes do FogBugz são:

- Possibilita registrar bugs através do navegador ou via e-mail, usando um recurso chamado *screenshot* FogBugz;
- Possui um poderoso sistema de busca, que permite pesquisar tanto o conteúdo dos bugs registrados quanto o conteúdo da wiki da ferramenta;
- Funciona em smartphones e tablets com diferentes versões do iOS, Android e BlackBerry.

Vantagens

Como destacado, a gestão de mudanças tem grande relevância por lidar com as alterações levantadas durante todo o ciclo de desenvolvimento do software. A partir da sua utilização, é possível visualizar e acompanhar toda a evolução do projeto, propiciando também que o processo de teste se mantenha atualizado e bem direcionado.

Além destas, outras vantagens podem ser listadas, tais como:

- Planejamento dos testes direcionado para as mudanças do software;
- Agilidade na realização dos testes;
- Maior eficiência e consistência no Processo de Teste;
- Controle sobre o escopo do projeto;
- Manutenção do histórico de atualizações da aplicação;
- Redução dos custos e riscos de mudanças não testadas;
- Maior controle sobre o resultado final do projeto;
- Economia de tempo em função dos testes voltados para as mudanças;
- Maior possibilidade de execução do Teste de Recessão;
- Melhor acompanhamento das alterações e das implementações desenvolvidas;
- Mais qualidade, uma vez que cada mudança, antes de ser realizada, tem seu impacto avaliado;
- Permite minimizar os problemas decorrentes ao processo de desenvolvimento, através de um controle sistemático sobre as modificações;
- Possibilita que as mudanças sejam planejadas e ocorram sempre que possível, evitando falhas inerentes ao processo;
- Aumento no sucesso da implantação das mudanças no software;
- Interrupção do software minimizada;
- Maior rapidez na identificação e correção dos problemas detectados.

Visando atender à demanda contínua por qualidade, as alterações são inevitáveis e, por isso, os softwares estão sujeitos a



DEVMEDIA

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior
portal para
desenvolvedores
da América
Latina!



20
mil
posts

430
mil
cadastrados

10
milhões de
page-views
por mês

constantes mudanças. Deste modo, tanto durante o processo de desenvolvimento, quanto durante a manutenção de um sistema, há sempre a necessidade de mudar, melhorar.

Diante disso, incorporar a Gestão de Mudanças ao ciclo de desenvolvimento do software pode ser um grande aliado para a garantia da qualidade, tornando viável a mensuração das alterações do projeto e, consequentemente, dos seus testes.

Gerir as mudanças pode agregar muitos benefícios ao projeto, desde melhorar a eficiência dos requisitos, do código e consequentemente dos testes, até a redução do risco de uma alteração não ter um correto planejamento dos testes.

Enfim, a Gestão de Mudanças pode possibilitar que o Processo de Teste forneça maior confiança ao software, ao fazer com que os testes se voltem para os pontos certos, os pontos relacionados às alterações efetuadas no projeto, além de priorizar a melhor utilização dos recursos técnicos e humanos disponíveis, otimizando todo o processo e garantindo, inclusive, um tempo hábil para a realização dos testes de regressão.

Links:

MOLINARI, Leonardo, 2007, "Gerência de Configuração - Técnicas e Práticas no Desenvolvimento do Software". Florianópolis: Visual Books.

PRESSMAN, Roger S. 2010, "Software Engineering: A Practitioner's Approach", 7 ed., McGraw Hill.

Gerência de Configuração de Software – Funções.

<http://www.din.uem.br/~pg45189/ensino/gcs/gcs-aula03-funcoes.pdf>

Gerenciamento e Controle de Mudanças.

<http://www.engenhariadesoftware.net.br/artigos/artigo-21-gerenciamento-e-controle-de-mudancas>

Gestão de Mudanças.

<http://pt.slideshare.net/Siqueira/gesto-de-mudanas>

Mudança de Software.

http://www.noginfo.com.br/arquivos/CC_ESOF_II_09.pdf

O que é gerência de configuração?

http://www.pronus.eng.br/artigos_tutoriais/gerencia_configuracao/gerencia_configuracao.php?pagNum=3

Testing the Changes.

<http://www.bcs.org/content/ConWebDoc/11735>

What is configuration management in software testing?

<http://istqbexamcertification.com/what-is-configuration-management-in-software-testing>

Autor



Renata Eliza

renataeliza@gmail.com - @RenataEliza

Atua na área de Teste de Software há mais de dez anos.

Tecnóloga em Processamento de Dados, MBA em Teste de Software e ISTQB Certified Tester Foundation Level. Mantém o blog www.asespecialistas.blog.com.



Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



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



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

