Edição 143 :: R$ 14,90

DEVMEDIA

DESTAQUE:

Conciliando agilidade e arquitetura de software
Aprenda a lidar com a arquitetura em ambientes ágeis

Relatórios com Hibernate e JasperReports
Aprimorando o design da aplicação com os padrões MVC e DAO

Programando com JSF e MongoDB
Construa sistemas web utilizando uma solução NoSQL

Introdução ao Spring Batch
Soluções elegantes e robustas para processamento em lotes

JAVA EE + ANGULARJS

Uma nova combinação para aplicações web

Usabilidade em sistemas corporativos
Saiba o que fazer para garantir uma boa usabilidade

Desvendando o processo de Teste
Conheça um guia básico e saiba por onde começar

ISSN 1676836-1
9 771676 836002 00143

MVP

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

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!

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

Edição 143 • 2015 • ISSN 1676-8361

EXPEDIENTE

Editor
Eduardo Spínola (*eduspinola@gmail.com*)

Consultor Técnico Diogo Souza (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

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 a 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

06 – Como conciliar agilidade e arquitetura de software
[*Gabriel Novais Amorim*]

Artigo no estilo Solução Completa

12 – Desenvolva aplicações web com JSF, MongoDB e JasperReports
[*Luis Gustavo Souza*]

Conteúdo sobre Novidades, Artigo no estilo Solução Completa

24 – Desenvolva aplicações com AngularJS e Java EE
[*Lucas de Oliveira Pires e Carlos Eduardo de Carvalho Dantas*]

Artigo no estilo Curso

34 – Introdução ao Spring Batch – Parte 1
[*Pedro E. Cunha Brigatto*]

Artigo no estilo Curso

46 – Relatórios avançados com Hibernate, JasperReports e PrimeFaces – Parte 2
[*Marcos Vinicios Turisco Dória*]

Conteúdo sobre Boas Práticas

55 – Usabilidade em sistemas corporativos
[*Adriano Schmidt*]

Conteúdo sobre Eng. de Software

63 – Desvendando o processo de Teste de Software
[*Haroldo Pereira Nascimento e Francisco Carlos de Matos Jr.*]

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 Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA ALGUNS DOS CURSOS

- Curso de Multicamadas com Delphi e DataSnap
- Delphi para Iniciantes
- Criando componente Boleto em Delphi
- Loja Virtual em Delphi Prism

Para mais informações :

www.devmedia.com.br/cursos/delphi

(21) 3382-5038

http://www.devmedia.com.br

Como conciliar agilidade e arquitetura de software

Aprenda a lidar efetivamente com arquitetura de software em ambientes ágeis

Fique por dentro

Os princípios por trás das práticas ágeis ainda não foram totalmente compreendidos por boa parte dos envolvidos no desenvolvimento de software e, ainda hoje, é bastante comum que práticas ágeis sejam adotadas sem critérios, criando um processo de software alternativo e ineficaz. Além disso, há muita coisa mal compreendida e mitos se espalhando como se fossem verdades, especialmente em relação à arquitetura de software e sua relação com a agilidade.

Muitas ideias difundidas em equipes ágeis é a de que arquitetura de software é burocracia e agilidade não fornece espaço para documentação. Em projetos ágeis, realmente a documentação é reduzida, mas em nenhum momento a arquitetura é deixada de lado; muito pelo contrário. Em um projeto ágil, a arquitetura de software é primordial. No entanto, atualmente tem sido difundida a ideia de que arquitetura de software não tem espaço em métodos ágeis. Com base nisso, este artigo mostrará como conciliar ambas, levando em consideração tanto o propósito arquitetural promovido pela disciplina de arquitetura de software quanto as preocupações em termos de processos de software associadas aos métodos ágeis.

A adoção de práticas ágeis no desenvolvimento de software, iniciada após o manifesto ágil, publicado em 2001, aumentou o número de "projetos ágeis" rapidamente, fazendo com que os pensamentos presentes nesse documento atingissem boa parte das organizações que produzem software. Hoje, em grande parte das empresas, há inúmeros projetos autoproclamados ágeis. É como se qualquer projeto com pouca (leia-se nenhuma) documentação e algumas práticas relacionadas fosse elevado ao patamar oficial de "projeto ágil" e, assim, acabasse de forma rápida, boa e barata.

Entretanto, aqueles que têm algum tempo de experiência sabem como terminam esses projetos: geralmente da pior maneira possível. O principal problema nesses casos de fracasso é a falta de conhecimento sobre como os princípios do manifesto ágil moldam as práticas mais comuns do desenvolvimento ágil e isso traz a ilusão de que a simples adoção ou remoção de determinadas práticas elevam um projeto ao status de ágil. Nesse contexto, uma das principais práticas elegíveis para remoção, por aqueles que não entendem a essência da agilidade, é diminuir ou, em casos mais extremos, acabar com a documentação. Ainda hoje, um dos maiores mal-entendidos sobre agilidade está relacionado a isso, pois boa parte das pessoas associam documentação a burocracia e a falta dela a agilidade.

Porém, é importante entender que a documentação, em especial o documento de arquitetura de software, é um artefato de extrema importância ao desenvolvimento de software por vários motivos; dentre eles, a definição formal da arquitetura do sistema e a possibilidade de fornecer um entendimento base comum a todos os membros da equipe. Basicamente, o documento de arquitetura de software descreve a arquitetura do sistema como um todo, referindo-se a restrições, organização, padrões e responsabilidades de módulos e componentes.

Arquitetura de software e documentação são assuntos que andam ou deveriam andar relacionados, pois boa parte do sucesso de um sistema é devido a sua arquitetura. Atributos como coesão, acoplamento e modularidade, por exemplo, são tratados no âmbito arquitetural e essas decisões arquiteturais deveriam ser

documentadas para propagar o conhecimento entre os envolvidos e interessados no sistema. Sendo assim, um projeto considerado ágil, para alcançar a alta qualidade técnica, não pode abrir mão da documentação.

Embora seja comum o pensamento de pouca ou nenhuma documentação em projetos ágeis, esse tipo de pensamento é um equívoco e para contornar essa situação, os próximos tópicos mostrarão como conciliar arquitetura de software e agilidade, de forma a manter o projeto bem documentado sem abrir mão da agilidade.

Agilidade em desenvolvimento de software

Agilidade em desenvolvimento de software representa um conjunto de ideias e práticas que surgiram há muito tempo por influência de alguns membros relevantes da comunidade de desenvolvimento. Essas ideias começaram a evoluir a partir do manifesto ágil, uma série de princípios lançados no ano de 2001 com o objetivo de simplificar e mudar a forma como os projetos de software eram encarados.

Os principais pontos levantados pelo manifesto ágil foram:

- **Indivíduos e interação entre eles** mais que processos e ferramentas;
- **Software em funcionamento** mais que documentação abrangente;
- **Colaboração com o cliente** mais que negociação de contratos;
- **Responder a mudanças** mais que seguir um plano.

Dentro desses pontos, há uma série de princípios que fornecem diretrizes que guiam os métodos ágeis como um todo (veja a seção **Links**). Entretanto, o próprio manifesto concorda que não há uma verdade absoluta nesses princípios, pois diz que "estamos descobrindo maneiras melhores de desenvolver software fazendo-o nós mesmos e ajudando outros a fazê-lo", ou seja, boas práticas para o desenvolvimento de software podem surgir a qualquer momento e, do mesmo modo, podem ser substituídas ou complementadas por outras ainda melhores. Neste contexto, os princípios ágeis basicamente fornecem uma série de valores nos quais práticas eficazes de desenvolvimento de software são baseadas.

As considerações levantadas pelo manifesto ágil descrevem o que é mais valoroso no desenvolvimento de software em relação a outras ideias também consideradas valorosas (geralmente) em processos formais, como em "software em funcionamento mais que documentação abrangente". É muito importante destacar que isso não significa que uma documentação abrangente não tenha valor. Apenas que software em funcionamento tem mais valor do que uma documentação abrangente, o que faz todo sentido. Afinal, de que adianta ter uma documentação abrangente e um software que não funciona? Entretanto, muita gente interpreta essas premissas do manifesto ágil de forma equivocada, pensando que deve-se ter software funcional e nenhuma documentação.

Com o passar do tempo, essas ideias de agilidade pautadas no manifesto ágil tomaram conta do cenário de desenvolvimento de software corporativo. As promessas de rapidez, qualidade e redução de custos tornaram os métodos ágeis bastante atraentes pelos olhos de gerentes e diretores. Entretanto, desde de que as ideias do manifesto ágil ganharam notoriedade e passaram a fazer parte do dia a dia dos desenvolvedores, seus princípios foram, na maioria das vezes, interpretados de forma errônea. Assim, não é raro encontrar "ambientes ágeis" em que as crenças de que documentação não é um artefato necessário são levadas a sério.

Pensar que a adoção de métodos ágeis implica na eliminação da documentação é um grande equívoco. O processo ágil, que tem como base os princípios do manifesto ágil, é o resultado do confronto entre ambientes sem processos e ambientes com processos formais. O objetivo é encontrar um meio-termo que estabeleça um processo suficiente para o desenvolvimento do software. Adotar um método ágil não significa abandonar processos e muito menos a documentação, mas infelizmente é comum a adoção de algumas práticas ágeis em detrimento de outras, tornando o processo ágil em questão quase um "não processo".

Mas é verdade que métodos ágeis demandam menos documentação. Como os métodos ágeis são orientados a pessoas e não a processos, tem-se um ambiente de maior colaboração (teoricamente deveria ser assim), sendo possível, dessa forma, a eliminação de muitos documentos cujo propósito sejam apenas para comunicação interna.

Sendo assim, métodos ágeis não eliminam toda a documentação e o principal documento em um projeto de software, o documento de arquitetura de software (DAS), definitivamente não deve ser removido dos artefatos do projeto. Nesse ponto, torna-se necessário entender como conciliar a arquitetura de software com a agilidade.

Para muitos desavisados esses dois temas são opostos, quando na verdade arquitetura deveria andar de mãos dadas com agilidade; do contrário, um ambiente ágil sem controle arquitetural torna-se caótico e em ambientes caóticos a produtividade tem níveis muito baixos, fazendo com que a adoção de um método ágil perca o sentido. Diante disso, os próximos tópicos irão se concentrar em como a arquitetura de software e a agilidade podem complementar-se mutuamente, a fim de encontrar o meio termo ideal em um ambiente com processo pouco formal, mas devidamente organizado.

Métodos preditivos e métodos adaptativos

"Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adequam a mudanças, para que o cliente possa tirar vantagens competitivas.". Esse é um dos princípios do manifesto ágil; preceito este que abre espaço para uma nova consideração: a diferença entre métodos ágeis (métodos adaptativos) e métodos formais (métodos preditivos).

Os métodos preditivos, como o nome sugere, são métodos que predizem com antecedência o que deverá acontecer. No cenário de um processo formal, os requisitos são levantados, documentados, analisados, uma solução técnica é projetada sobre os requisitos e, por fim, o software é desenvolvido com base no projeto. O exemplo clássico é o famigerado processo em cascata (*waterfall*).

No entanto, todas essas fases estão presentes tanto nos processos formais (preditivos) quanto nos processos ágeis (adaptativos). O que torna um método adaptativo é o conceito de iteração em ciclos menores por todas as fases do processo de desenvolvimento.

Em vez de passar por um longo processo contemplando as fases de um processo de desenvolvimento de software, métodos ágeis passam por todas as fases repetidas vezes, em ciclos menores. Isso torna os métodos ágeis mais adaptativos que preditivos. Cabe aqui ilustrar esse conceito, comparando o processo formal com o processo ágil, conforme mostra a **Figura 1**. É interessante observar como os métodos ágeis englobam o processo formal e o transformam em um processo adaptativo por meio das iterações.

De forma geral, a previsibilidade em projetos de desenvolvimento de software não é possível, pois os requisitos são imprevisíveis, estão sempre mudando e variam conforme o ambiente no qual o software em questão está inserido. Regras governamentais, concorrência, alterações no padrão de consumo dos clientes, são todos fatores que influenciam os requisitos. Entretanto, apesar desse cenário e do fato do software ser imprevisível, isso não significa que não se possa controlar o seu desenvolvimento.

Para controlar o desenvolvimento de software em um ambiente ágil e adaptativo é preciso saber o que está pronto, qual a situação atual, o que se tem definido e o que precisa mudar. Nesse caso, a melhor maneira para se obter esse status é por meio de um mecanismo que diga a situação atual em curtos intervalos de tempo: o desenvolvimento iterativo.

Desenvolver iterativamente significa produzir versões funcionais do sistema final a cada iteração. Essas versões intermediárias devem possuir um subconjunto dos recursos definidos até o momento para o sistema em questão. Além disso, essas versões devem seguir o mesmo padrão de qualidade aplicados à versão final. Esse método é bastante eficaz, pois não há melhor forma para testar a realidade de um projeto do que a imposição de entrega de software funcional. Neste modelo, cada release de software funcional é um marco e responde às questões que precisam ser respondidas para se controlar o desenvolvimento do software em um ambiente adaptativo e ágil.

A base dos métodos ágeis é o desenvolvimento iterativo. Dessa forma, eles se tornam adaptáveis a mudanças no projeto. Entretanto, adaptabilidade sem organização não é ágil, é um processo *ad hoc*. Embora muitos projetos sejam classificados por seus membros como ágeis e adaptativos, eles não são de fato, pois pecam na interpretação dos princípios ágeis, especialmente no quesito documentação, onde virou quase unanimidade a ideia errada de que agilidade e documentação não se misturam. Diante disso, os próximos tópicos abordarão a conciliação de agilidade com arquitetura de software, trazendo à tona as principais ideias sugeridas pelos pensamentos ágeis.

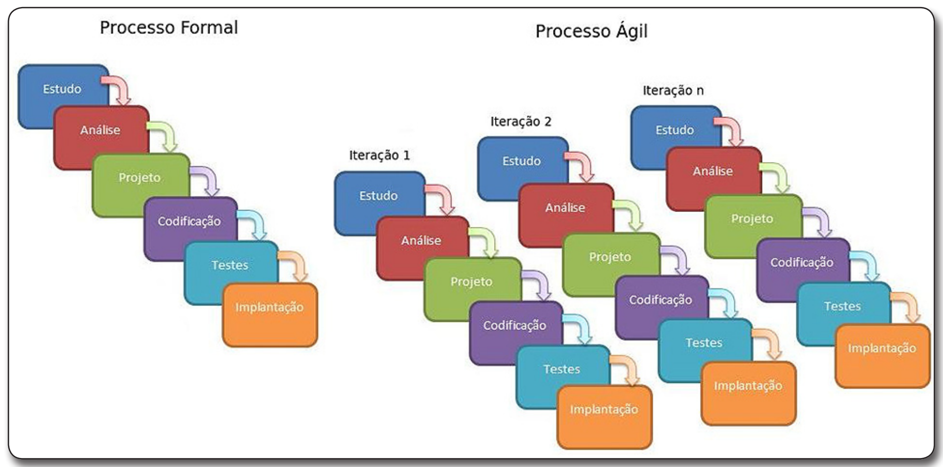


Figura 1. Processos ágeis englobam processos formais de forma iterativa

Conciliação de agilidade e arquitetura de software

A conciliação da arquitetura de software com a agilidade tem se tornado um tema recorrente em debates da comunidade de desenvolvimento e, até o presente momento, não há respostas conclusivas sobre como deve ser a relação entre esses dois tópicos. Uma das ideias mais difundidas é a de que a arquitetura de software não deve ser ignorada. Tanto é que uma das principais diretrizes do RUP, um processo de software precursor dos métodos iterativos, é definir a arquitetura o mais cedo possível.

Nota

Grande parte da literatura sobre métodos ágeis também comenta sobre a importância da definição da arquitetura no início do projeto de uma maneira bottom-up, além de enfatizar a responsabilidade do arquiteto como sendo a de cuidar dos pontos de maior risco e incertezas.

Talvez um dos pontos que mais causem confusão em relação à adoção de métodos ágeis e a aproximação com a arquitetura de software seja o mal entendimento do que os princípios ágeis querem dizer e qual o papel de cada membro em um projeto ágil de software. A fim de contribuir com este entendimento, os próximos tópicos irão expor as ideias por trás dos princípios ágeis, elucidar sobre os papéis dos membros em projetos ágeis e buscar a aproximação desses pontos com a arquitetura de software.

Mitos sobre arquitetura vs. agilidade

Equipes ágeis tendem a criar e manter pouca documentação em comparação às equipes com processos mais tradicionais. Este é o principal ponto de conflito entre arquitetura de software e agilidade e também o que causa o maior número de equívocos. Na comunidade de desenvolvimento existem alguns mitos quando se fala em adotar métodos ágeis e manter a arquitetura de software. Essas ideias deixam claro quais são os principais

pontos de divergência entre arquitetura e agilidade, elencadas na **Tabela 1**.

O entendimento da **Tabela 1** é o ponto de partida para a conciliação entre arquitetura de software e agilidade. As ideias apresentadas deixam claro que os pontos de divergência são falácias, que caem por terra ao se analisar e entender realmente a natureza da arquitetura de software ou do processo de software (no caso, um processo ágil). A maioria das pessoas possui um conhecimento muito superficial sobre boa parte dos assuntos que acredita dominar, e não é diferente com agilidade e arquitetura de software, tendo em vista os mitos mais comuns que foram apresentados anteriormente. As ideias enraizadas atualmente são de que métodos ágeis dispensam documentação e de que a arquitetura de software é burocrática e exige muita documentação, quando na verdade nenhuma destas ideias está correta.

Quando se entende realmente a natureza da arquitetura de software e das metodologias ágeis, essas falácias deixam de fazer sentido e a conciliação entre ambas passa a ser não apenas plausível, como necessária. Sendo assim, os próximos tópicos têm por objetivo apresentar a conciliação da arquitetura com a agilidade corroborando a realidade exposta na **Tabela 1**.

O papel do arquiteto em um projeto ágil

Para aproximar a arquitetura de software dos métodos ágeis, nada mais justo do que começar pelo arquiteto de software e suas responsabilidades em um projeto ágil.

Andrew Johnson, da *AgileArchitect.org*, escreveu que "em um projeto ágil, o arquiteto tem como principal responsabilidade levar em conta a mudança e complexidade, enquanto os desenvolvedores focam na próxima entrega" (veja na seção **Links** *The Role of Agile Architect*). Além disso, Mike Cohn, em *Succeeding with Agile*, comenta sobre "non-coding architect" (arquitetos de software que não codificam, apenas criam diagramas) e enfatiza que o arquiteto ideal é aquele que contribui com o time. Essa contribuição a que Cohn se refere é a codificação, ou seja, o arquiteto em um projeto ágil deve colocar as mãos na massa. Sendo assim, de acordo com o exposto anteriormente, chega-se à conclusão de que o arquiteto, em um projeto ágil, tem o papel de codificar os elementos mais complexos do software, levando-se em conta as mudanças.

É importante frisar a consideração feita anteriormente sobre as mudanças que o software possa ter. Note como a definição de Johnson, sobre o papel do arquiteto, coloca como principal responsabilidade deste a consideração a mudanças que possam ocorrer no software. Isso acontece devido ao fato de métodos ágeis serem adaptativos, ou seja, orientados pelas mudanças. São as mudanças que guiam o que deverá ser feito para as próximas releases. Nesse caso, o arquiteto de software, em um ambiente ágil, deve estar inserido na arquitetura do software conhecendo a fundo o código e ele mesmo participando da codificação. Essa prática torna as decisões e mudanças arquiteturais mais rápidas e condiz com o fato dos métodos ágeis aceitarem mudanças a qualquer momento, independente dessas mudanças impactarem ou não na arquitetura.

Um dos princípios do manifesto ágil prega exatamente o seguinte: "Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adequam a mudanças, para que o cliente possa tirar vantagens competitivas.". Esse princípio corrobora a importância das atribuições propostas para arquitetos de software em métodos ágeis comentadas anteriormente, tal como a consideração a mudanças como uma das principais responsabilidades.

Após entender o papel do arquiteto de software em um projeto ágil, o próximo tópico abordará práticas ágeis para se definir a arquitetura do projeto, como lidar com um dos artefatos arquiteturais de maior importância, o documento de arquitetura de software e como conciliar a criação e manutenção desse documento com métodos ágeis.

Definindo a arquitetura de software de modo incremental

A definição da arquitetura em um ambiente ágil pode traçar os rumos que o projeto irá tomar. A arquitetura sinaliza se o projeto será um caso de sucesso ou se irá definhar até o completo fracasso.

Mito	Realidade
Arquitetura de software produz "muito papel".	O processo de software adotado determina quais documentos são realmente necessários. Comunica-se somente o estritamente necessário.
Arquitetura de software implica em criar todos os modelos arquiteturais no início do projeto.	A arquitetura deve respeitar a natureza do método. Em projetos ágeis, a arquitetura do software deve ser evolutiva.
Requisitos arquiteturais não podem mudar a partir de um certo momento.	Métodos ágeis aceitam mudanças a qualquer momento, tendo impacto ou não sobre a arquitetura. O cliente deve sempre estar ciente das consequências de uma mudança de requisito (arquitetural ou não).
Softwares desenvolvidos com métodos ágeis não têm arquitetura.	Todo software tem uma arquitetura, independente se alguém a projetou intencionalmente ou não.
"Arquiteto de software é somente um novo e pomposo título que programadores pedem para ter em seus cartões.". Projetos ágeis não precisam do arquiteto.	Vários métodos ágeis prescindem de papéis. Mesmo que ninguém na equipe tenha o papel ou cargo de arquiteto de software, convém planejar a arquitetura.
Toda a arquitetura deve ser modelada no início do projeto.	Novamente: o arquiteto deve respeitar a natureza do projeto. Se o método prescreve "prove com código sempre que possível", é interessante codificar um pouco sem antes modelar completamente a solução.

Tabela 1. Mitos sobre arquitetura de software e sua união com agilidade

Todo software possui uma arquitetura, que pode ser criada de modo intencional, quando é feito um projeto arquitetural com o auxílio de diagramas, ou de modo acidental, quando o desenvolvimento parte direto para a codificação, sem a preocupação explícita com tais conceitos. A importância da arquitetura de software e a forma como surge um projeto arquitetural, seja ele intencional ou não, são comentadas com mais detalhes no artigo "Conformação Arquitetural: sincronizando o código e a arquitetura do projeto" (veja a seção **Links**), mas de uma maneira geral, a importância da arquitetura de software diz respeito às bases sobre as quais o projeto será construído, a fim de organizar de forma coesa os módulos e componentes de um sistema, com o objetivo de facilitar a implementação e manutenção do mesmo.

Em qualquer ambiente é desejável se ter um projeto de software fácil de implementar e manter, mas em ambientes ágeis essa necessidade é maior devido à natureza do processo, que é voltada a mudanças. No caso de métodos ágeis, a arquitetura deve ser, mais do que nunca, projetada considerando o contexto de negócio no qual o software está inserido e possíveis mudanças que possam ocorrer. Sendo assim, é necessário adotar os princípios e práticas ágeis no momento de modelar, desenvolver e evoluir uma arquitetura.

Uma prática difundida para o projeto arquitetural ágil é a *Agile Model Driven Development* (AMDD), que como o nome sugere, é a versão ágil do *Model Driven Development* (MDD). O MDD é uma prática de desenvolvimento de software onde o projeto é elaborado de forma extensiva por meio de modelos antes de se partir para o código fonte (veja o **BOX 1**). A diferença entre MDD e AMDD é a quantidade de modelos criados antes da codificação. No AMDD, onde os princípios ágeis são aplicados, em vez da criação de modelos representando todo o sistema, são criados modelos apenas para satisfazer os próximos esforços de desenvolvimento. Dessa forma, tem-se uma arquitetura definida de modo incremental.

A **Figura 2** mostra o ciclo de vida do AMDD, que, basicamente, ilustra a modelagem arquitetural durante o ciclo de vida do projeto. A ideia é desenvolver a arquitetura de forma incremental, seguindo os moldes dos processos ágeis. No diagrama apresentado, cada uma das caixas representa uma atividade de desenvolvimento.

Na abordagem AMDD, a primeira iteração (*Iteration 0: Envisioning*) tem por objetivo iniciar a previsão da arquitetura e inclui duas subatividades: previsão dos requisitos iniciais e previsão da arquitetura inicial. O objetivo nesta iteração é identificar o escopo de alto nível para a arquitetura de acordo com os requisitos iniciais, incluindo uma visão arquitetural para esse determinado momento contendo um modelo de alto nível dos requisitos e um modelo de alto nível arquitetural, sem se preocupar com especificações detalhadas. A previsão arquitetural, dependendo do processo adotado, pode ser a fase de concepção (RUP) ou ocorrer antes do primeiro sprint (Scrum). O importante é que a previsão arquitetural seja feita antes das iterações de desenvolvimento. O diagrama também indica a medida de tempo comumente utilizada para as subatividades. No caso da previsão dos requisitos iniciais e previsão da arquitetura inicial, a medida de tempo é em termos de dias.

As demais atividades podem ocorrer durante qualquer iteração de desenvolvimento (*Iteration 1..n: Development*) e têm a medida de tempo em termos de horas ou minutos. Isso faz com que a arquitetura seja evolutiva e possa ser replanejada durante as demais iterações de desenvolvimento. A ideia é se trabalhar de uma "maneira JIT" (*Just in Time*), fazendo somente o necessário naquele momento para o projeto continuar progredindo, lembrando que sempre é possível voltar atrás para mudar. O AMDD traz para o projeto arquitetural um processo em moldes ágeis para se definir e implementar a arquitetura. Da mesma forma, o documento de arquitetura de software pode ser criado e mantido seguindo as diretrizes desta abordagem.

BOX 1. Model Driven Development (MDD)

Model Driven Development (MDD) é uma prática que consiste na criação de modelos abstratos para representar o software antes de programar qualquer linha de código. De acordo com as ideias dessa abordagem, o modelo deve ter o mesmo nível de detalhes do código fonte. Por exemplo: as classes no diagrama devem contemplar todos os detalhes da classe codificada como, por exemplo, seus atributos. No MDD, essa representação deve ser bidirecional, ou seja, o modelo deve corresponder fielmente ao código e vice-versa, possibilitando assim a engenharia reversa entre modelo e código com o uso de ferramentas CASE.

O MDD é uma prática genérica e não especifica como os modelos devem ser implementados. Aproveitando esta lacuna, o Object Management Group (OMG) criou o OMG Model Driven Architecture (MDA), um padrão de modelagem arquitetural seguindo as práticas MDD e definindo quais artefatos e ferramentas utilizar, de forma a ser independente de tecnologia de implementação e voltada a padrões de representação como a UML. Veja a seção **Links** para saber mais sobre MDA.

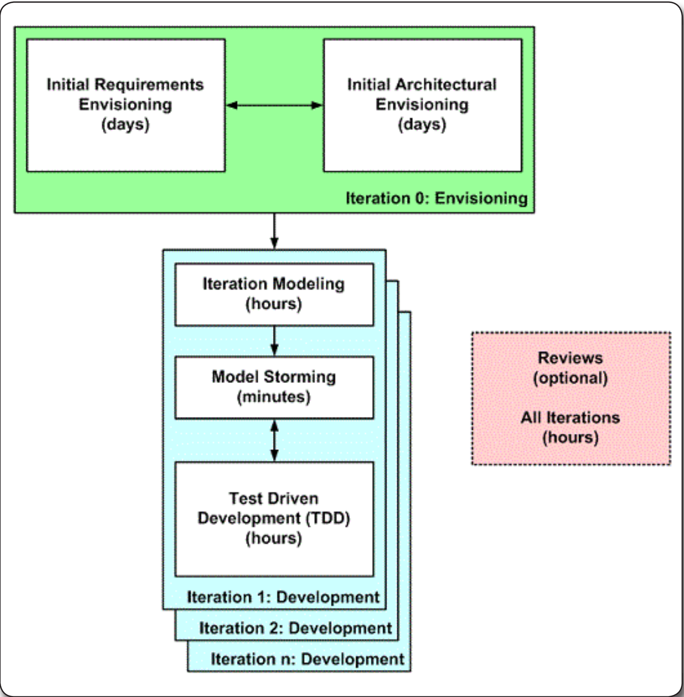


Figura 2. Ciclo de vida do AMDD

Enfim, a conciliação

Um dos pensamentos do manifesto ágil é o de que código funcionando é mais importante do que documentação, mas isto não significa que a documentação deve ser eliminada ou ignorada, como foi exposto nos tópicos anteriores. O ideal é responder a seguinte pergunta antes de sair criando artefatos ou documentos: "Quem vai usar isto? É útil?".

Em relação à ideia de que arquitetura de software é burocrática, é preciso deixar claro que arquitetura de software não é um processo de software. Afinal, é o processo de software adotado no projeto que é responsável por fornecer o caminho dos artefatos que devem ou não ser criados e como utilizá-los.

Independentemente do processo utilizado no projeto de software, a arquitetura estará presente, seja de modo acidental (quando existe simplesmente pelas decisões de implementação) ou intencional (quando há preocupação com sua elaboração e manutenção).

Sintetizando a conciliação de arquitetura de software e agilidade, chegamos à conclusão de que a arquitetura não pode ser encarada como um processo de software a ponto de ter artefatos e documentos criados sem utilidade: ela deve respeitar o processo. Sendo assim, em métodos ágeis, a arquitetura deve ser evolutiva, como manda o processo. A arquitetura deve refletir o projeto atual e acolher as mudanças, pois métodos ágeis aceitam mudanças a qualquer momento. E o mais importante é entender que todo software possui uma arquitetura, seja ela intencional ou não, sendo o ideal que ela seja intencional.

A conciliação da arquitetura de software com a agilidade é muito importante para o sucesso do projeto e para que o produto desenvolvido seja de qualidade. Sabendo disso, a principal forma para conseguir essa aproximação é conhecendo o propósito de cada uma e tratá-las como realmente o são, isto é, arquitetura como disciplina para organizar software e agilidade para tratar do processo, isto é, como disciplinas complementares.

Além dos conceitos analisados neste artigo, há muito a se explorar sobre o tema, especialmente sobre AMDD e como transformá-lo em AMDA para a criação de arquiteturas de forma incremental em um processo ágil por meio de procedimentos e ferramentas padronizados.

Autor

Gabriel Novais Amorim
novais.amorim@gmail.com – blog.gabrielamorim.com
Tecnólogo em Análise e Desenvolvimento de Sistemas (Unifieo) e especialista (MBA) em Engenharia de Software (FIAP). Trabalha com desenvolvimento de software há seis anos e atualmente tem trabalhado no desenvolvimento de soluções SOA na plataforma Java. Possui as certificações OCJP, OCEJWCD, OCEEJBD, IBM-OOAD, IBM-RUP, CompTIA Cloud Essentials e SOACP. Seus interesses incluem arquitetura de software e metodologias ágeis.

Links:

Manifesto para o desenvolvimento ágil de software.
http://www.manifestoagil.com.br/

Os princípios do manifesto ágil.
http://www.manifestoagil.com.br/principios.html

The Role of the Agile Architect.
http://www.agilearchitect.org/agile/role.htm

OMG Model Driven Architecture.
http://www.omg.org/mda/

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!

Desenvolva aplicações web com JSF, MongoDB e JasperReports

Veja neste artigo como construir sistemas JSF utilizando o MongoDB como ferramenta para o armazenamento de dados

A quantidade de informações a armazenar cresce de forma exponencial a cada dia. Neste cenário, para que seja possível viabilizar a persistência de tanto conteúdo, é necessário que existam soluções capazes de lidar com esses grandes volumes de dados, atualmente gerados por milhões de usuários.

Para tanto, devido às grandes limitações encontradas em bancos de dados relacionais para lidar com quantidades massivas de dados, corporações como o Google, Amazon, Facebook, entre outras, lideram pesquisas para o desenvolvimento de soluções não relacionais, mais popularmente conhecidas como NoSQL, com o intuito de construir opções que não possuam tais restrições.

A estrutura flexível proporcionada pelo modelo não relacional possibilita aos administradores de bancos de dados maior praticidade com a escalabilidade, o que, dentre outros fatores, tem sido um dos que mais atraem as empresas a aderirem a este modelo.

Como um exemplo, a Globo.com, responsável pelo fantasy game CartolaFC, que já possui mais de dois milhões de usuários, optou pela adoção do banco de dados MongoDB com o intuito de obter alta performance e disponibilidade.

Dentre algumas das vantagens de se utilizar o NoSQL, podemos destacar:

- A capacidade de armazenar e processar grandes quantidades de dados;
- Maior praticidade com a escalabilidade;
- Possui diversos bancos de dados gratuitos, sendo os mais populares o MongoDB, HBase e Apache Cassandra.

Fique por dentro

Este artigo, em forma de tutorial, apresenta em um exemplo prático como desenvolver aplicações JSF utilizando o banco de dados MongoDB para o armazenamento de informações. Deste modo, ele é útil para desenvolvedores que desejam conhecer uma opção NoSQL e adotá-la em soluções web. Além disso, abordaremos como integrar o MongoDB ao iReport para a construção e apresentação de relatórios nestas soluções.

Como desvantagem, vale ressaltar a enorme dificuldade em encontrar profissionais que estejam aptos a trabalhar com NoSQL. Por ser uma tecnologia nova, grande parte dos especialistas com experiência em bancos relacionais possui conhecimento limitado a respeito das soluções NoSQL.

Dito isso, aprenderemos neste artigo que não há nenhuma grande complicação ao utilizar um banco não relacional. Para tanto, vamos implementar uma aplicação JSF que possui um formulário de cadastro e utilizar o MongoDB como ferramenta para viabilizar a persistência dos dados. Em seguida, pensando em enriquecer ainda mais o exemplo, para exibir os dados armazenados vamos construir um relatório com a ferramenta iReport.

Ressaltamos que a intenção do artigo não é comparar o modelo não relacional com o modelo relacional, que por mais de quarenta anos tem sido a principal opção para o armazenamento de dados, mas sim demonstrar algumas características do banco de dados não relacional MongoDB e seu uso em uma aplicação JSF com relatórios JasperReports.

Sobre o MongoDB

O MongoDB é um dos vários bancos de dados criados nos últimos anos sob o conceito de bancos orientados a documentos. Este conceito tem como principal característica utilizar uma estrutura baseada em coleções e documentos para armazenar os dados. Assim, ao contrário dos bancos de dados relacionais, onde trabalhamos com tabelas, neste modelo lidamos com coleções. A **Tabela 1** traz uma comparação dos termos utilizados no modelo relacional e no MongoDB.

Como um dos grandes diferenciais, destaca-se que no modelo relacional é sugerido que a redundância de dados seja evitada, pois os relacionamentos entre entidades são uteis justamente para evitar a redundância de informações. Já no MongoDB ocorre exatamente o contrário: não existem relacionamentos e com isso a redundância de informações é incentivada. Deste modo, em um documento devem estar todas as informações do elemento persistido, mesmo que elas sejam redundantes. Isto evita a necessidade de realizar joins para obter as informações em uma coleção, o que resulta em ganho de performance.

SQL - Termos e Conceitos	MongoDB - Termos e Conceitos
DATABASE	DATABASE
TABLE	COLLECTION
ROW	DOCUMENT
COLUMN	FIELD
INDEX	INDEX

Tabela 1. Comparação dos conceitos SQL e MongoDB

Sobre o JasperReports e iReport

O JasperReports é uma biblioteca open source utilizada para a geração de relatórios na plataforma Java. Como grande vantagem, fornece o recurso de exportar os relatórios em diversos tipos de arquivos, como PDF, XML e XLS. Além disso, é possível agrupar outros relatórios em um único com a opção de criação de sub-relatórios para melhor organizar as informações. Outra característica interessante é a possibilidade de trabalhar com diversos bancos de dados que permitem efetuar conexões via JDBC, incluindo bancos não relacionais, como MongoDB e HBase.

Com a utilização do JasperReports, toda a estrutura de um relatório é definida em um arquivo XML, geralmente com a extensão .jrxml. Desta forma, permite ao desenvolvedor realizar modificações diretamente no código fonte do relatório, caso seja de sua preferência.

Já o iReport é uma ferramenta gráfica utilizada para a elaboração de relatórios que utilizam a biblioteca JasperReports. Esta ferramenta fornece diversos componentes que possibilitam a redução do tempo para desenhar os relatórios como, por exemplo, subreports, charts, barcodes, etc. Ademais, o iReport também disponibiliza diversos templates, que podem ajudar e poupar o tempo do desenvolvedor.

A aplicação exemplo

No decorrer do artigo, como já mencionado, apresentaremos o desenvolvimento de uma aplicação MVC com JavaServer Faces e MongoDB. Para isso, construiremos um formulário simples com

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!**

o intuito de cadastrar informações de venda de produtos. Pensando na exibição de tais informações, será demonstrado como recuperar dados do MongoDB e criar um relatório com iReport, quando abordaremos também o uso de alguns dos seus vários componentes, como o subreport.

Criando a aplicação

Para iniciar a implementação do nosso exemplo é necessário que você tenha o ambiente de desenvolvimento configurado. Neste caso, vamos adotar o Eclipse Indigo como IDE e o iReport para o desenvolvimento do relatório. Portanto, baixe e instale estas ferramentas, cujos endereços para download podem ser encontrados na seção **Links**. O MongoDB também pode ser baixado do site oficial do projeto. Lá você notará que existem versões compatíveis para Windows, Linux, Mac e Solaris. Aqui, iremos baixar a versão compatível para o Windows. Para instalá-lo basta seguir as etapas sugeridas durante a execução do wizard.

Encerrada essa etapa podemos, enfim, criar o projeto. Portanto, abra o Eclipse, acesse o menu *File* > *New* e, em seguida, escolha a opção *Dynamic Web Project*. Feito isso, acesse a pasta *bin* do MongoDB (em nosso exemplo, o diretório a ser acessado pelo terminal de comando é *C:\mongodb\bin*), digite "mongo" e pressione *Enter* para abrir o terminal onde poderemos gerenciar o banco de dados exemplo.

Feito isso, digite *use test* no terminal para informar que iremos utilizar a base nomeada como *test*. Esta base é criada durante o processo de instalação do MongoDB. Logo após, digite o código exibido na **Listagem 1** para criar as collections de produtos, clientes e vendas, que serão adotadas em nosso exemplo. Assim, teremos o banco de dados pronto.

Neste exemplo, iremos inserir produtos, clientes e registros de venda através da aplicação pela interface web, mas caso seja do seu interesse, você também pode inserir as informações diretamente no terminal, via linha de comando. O código de exemplo para isso pode ser encontrado a partir da linha 6.

Por fim, antes de iniciar o desenvolvimento, é preciso que você realize o download de mais algumas bibliotecas, a saber: *mongo-java-driver.jar*, para possibilitar a conexão com o banco de dados e a biblioteca *primefaces.jar*, para utilizarmos alguns componentes visuais do PrimeFaces (veja os endereços indicados na seção **Links**). Estes arquivos devem ser adicionados na pasta *lib* do projeto.

Conexão com o banco de dados

Com o projeto e o banco de dados criados, vamos configurar a conexão entre nossa aplicação e o MongoDB. Para isso, crie a classe **ConexaoMongo** com o código apresentado na **Listagem 2**.

Nesta classe implementamos o método **getConnection()** e nele solicitamos por parâmetro o nome da collection em que serão realizadas as operações de cadastro ou consulta.

Após isso, é criada a conexão na linha 14, onde informamos o endereço do banco de dados e a porta de acesso, que por padrão é a 27017. Por fim, informamos ao método **getCollection()** – na linha 15 – o nome da collection a ser utilizada.

Listagem 1. Código para criação das collections no MongoDB.

```
01. use db test;
02. db.createCollection("clientes");
03. db.createCollection("produtos");
04. db.createCollection("vendas");
05.
06. // Script para inserir os produtos
07.
08. db.produtos.insert({codigo: '1', descricao:'Chocolate', preco:'10.00', ativo:'S'})
09. db.produtos.insert({codigo: '2', descricao:'Morango', preco:'20.00', ativo:'S'})
10. db.produtos.insert({codigo: '3', descricao:'Leite', preco:'30.00', ativo:'S'})
11. db.produtos.insert({codigo: '4', descricao:'Café', preco:'40.00', ativo:'N'})
12.
13. // Script para inserir os clientes
14.
15. db.clientes.insert({código: '1', nome: 'Pedro', rg:'44.444.444-4', cpf:'444.444.444-
16. 44', ativo:'S'})
17. db.clientes.insert({código: '2', nome: 'Paulo', rg:'55.555.555-5', cpf:'555.555.555-
18. 55', ativo:'S'})
19.
20. // Script para inserir as vendas
21.
22. db.vendas.insert({codigo:'1', codigocliente:'1', nomecliente:'Pedro',
23. codigoproduto:'1', descricaoproduto:'Chocolate', quantidade:'10'});
24. db.vendas.insert({codigo:'2', codigocliente:'2', nomecliente:'Paulo',
25. codigoproduto:'4', descricaoproduto:'Café', quantidade:'20'});
26. db.vendas.insert({codigo:'3', codigocliente:'2', nomecliente:'Paulo',
27. codigoproduto:'3', descricaoproduto:'Leite', quantidade:'5'});
```

Listagem 2. Código da classe ConexaoMongo.

```
01. package br.com.bean;
02.
03. import java.net.UnknownHostException;
04. import javax.faces.bean.ManagedBean;
05. import com.mongodb.DB;
06. import com.mongodb.DBCollection;
07. import com.mongodb.Mongo;
08.
09. public class ConexaoMongo {
10.
11.    public static DBCollection getConnection(String nomeCollection) throws
12.    UnknownHostException {
13.         Mongo mongodb = new Mongo("localhost", 27017);
14.         DB database = mongodb.getDB("test");
15.         DBCollection collection = database.getCollection(nomeCollection);
16.         return collection;
17.     }
18.
19. }
```

Criando o managed bean

Após criar a classe de conexão com o banco de dados, vamos implementar os métodos responsáveis por invocar os métodos de inserção e consulta de informações no banco de dados. Para isso, crie a classe **VendasBean**, cujo código é apresentado na **Listagem 3**.

Neste código temos como primeiro método o **init()**, e dentro dele invocamos os métodos **limparTela()**, **listarClientes()** e **listarProdutos()**. Antes disso, no entanto, é verificado na linha 12 se a página não é um postback, ou seja, se a página não é fruto de um envio de formulário, por exemplo. Caso não seja, os métodos citados anteriormente são executados, pois isto só pode acontecer quando for o primeiro carregamento da página feito pelo usuário.

Listagem 3. Código da classe VendasBean.

```
01. public class VendasBean {
02.
03.   private Produto produto = new Produto();
04.   private Cliente cliente = new Cliente();
05.   private Venda venda = new Venda();
06.   public List<Venda> list = new ArrayList<Venda>();
07.   public List<Produto> listProdutos = new ArrayList<Produto>();
08.   public List<Cliente> listClientes = new ArrayList<Cliente>();
09.   public String codigoProduto, codigoCliente;
10.
11.   public void init() throws UnknownHostException {
12.     if (!FacesContext.getCurrentInstance().isPostback()) {
13.       limparTela();
14.       listarClientes();
15.       listarProdutos();
16.     }
17.   }
18.
19.   public void limparTela() {
20.     this.codigoCliente = null;
21.     this.codigoProduto = null;
22.     this.list = null;
23.   }
24.
25.   public void novoProduto() throws UnknownHostException {
26.     RequestContext context = RequestContext.getCurrentInstance();
27.     ProdutoService produtoService = new ProdutoService();
28.     if(produtoService.insertProduto(produto)) {
29.       FacesContext.getCurrentInstance().addMessage("messages",new
30.         FacesMessage(FacesMessage.SEVERITY_INFO,
31.
32.   "Produto cadastrado com sucesso!",""));
33.     } else {
34.       FacesContext.getCurrentInstance().addMessage("messages",new
35.         FacesMessage(FacesMessage.SEVERITY_ERROR,
36.
37.    "Erro ao cadastrar produto!", ""));
38.       context.addCallbackParam("success", false);
39.     }
40.   }
41.
42.   public void novoCliente() throws UnknownHostException {
43.     RequestContext context = RequestContext.getCurrentInstance();
44.     ClienteService clienteService = new ClienteService();
45.
46.     if(clienteService.insertCliente(cliente)) {
47.       FacesContext.getCurrentInstance().addMessage("messages",new
48.         FacesMessage(FacesMessage.SEVERITY_INFO,
49.
50.   "Cliente cadastrado com sucesso!",""));
51.     } else {
52.       FacesContext.getCurrentInstance().addMessage("messages",new
53.         FacesMessage(FacesMessage.SEVERITY_ERROR,
54.
55.   "Erro ao cadastrar cliente!", ""));
56.       context.addCallbackParam("success", false);
57.     }
58.   }
59.
60.   public void novaVenda() throws UnknownHostException {
61.     RequestContext context = RequestContext.getCurrentInstance();
62.     VendaService vendaService = new VendaService();
63.
64.     if(vendaService.insertVenda(venda)) {
65.       FacesContext.getCurrentInstance().addMessage("messages",new
66.         FacesMessage(FacesMessage.SEVERITY_INFO,
67.   "Venda cadastrada com sucesso!",""));
68.     } else {
69.       FacesContext.getCurrentInstance().addMessage("messages",new
70.       FacesMessage(FacesMessage.SEVERITY_ERROR,
71.
72.   "Erro ao cadastrar venda!", ""));
73.       context.addCallbackParam("success", false);
74.     }
75.     listarVendas();
76.   }
77.
78.   public String listarVendas() throws UnknownHostException {
79.     VendaService vendaService = new VendaService();
80.     list = vendaService.buscarVendas(venda);
81.
82.     return "success";
83.   }
84.
85.   public String listarProdutos() throws UnknownHostException {
86.     ProdutoService produtoService = new ProdutoService();
87.     listProdutos = produtoService.buscarProdutos(produto);
88.     return "success";
89.     }
90.
91.   public String listarClientes() throws UnknownHostException {
92.     ClienteService clienteService = new ClienteService();
93.     listClientes = clienteService.buscarClientes(cliente);
94.     return "success";
95.   }
96. }
```

Estes métodos, como indicado pelo nome, serão responsáveis por limpar os campos da tela no primeiro acesso e buscar clientes e produtos no banco de dados. O conteúdo retornado será listado de forma separada no componente **<p:selectOneMenu>** na camada view. Após isso, implementamos os métodos **novoProduto()**, **novoCliente()** e **novaVenda()**. Estes possuem a mesma estrutura interna e executarão os respectivos métodos – das classes **ProdutoService**, **ClienteService** e **VendaService** da camada de serviço – para inserir informações no banco de dados.

Por fim, criamos os métodos **listarVendas()**, **listarProdutos()** e **listarClientes()**, que irão buscar os valores inseridos nas respectivas collections.

Camada de serviço

No passo anterior definimos os métodos que serão invocados pela camada view e serão responsáveis por realizar a chamada dos métodos da camada de serviço. Para isso, criamos as classes **ProdutoService**, **ClienteService** e **VendaService**. Nelas estarão contidos os métodos que efetivamente irão realizar a busca e inserção de dados no banco. A **Listagem 4** exibe o código da classe **ProdutoService**.

Os procedimentos para efetuar a consulta e o cadastro de produtos, clientes e vendas são idênticos. Portanto, explicaremos apenas o código de dois deles: **inserirProduto()** e **buscarProdutos()**. No método **inserirProduto()**, de forma objetiva, adicionamos um registro de produto na devida collection.

Listagem 4. Código da classe ProdutoService.

```
01. public class ProdutoService {
02.
03.  public boolean inserirProduto(Produto produto) throws UnknownHostException {
04.
05.    DBCollection collection = ConexaoMongo.getConnection("produtos");
06.
07.    BasicDBObject document = new BasicDBObject();
08.    document.put("descricao", produto.getDescricao());
09.    document.put("preco", produto.getPreco());
10.    document.put("ativo", produto.getAtivo());
11.
12.    if (collection.insert(document) != null) {
13.      return true;
14.    }
15.
16.    return false;
17.
18.  }
19.
20.  public List<Produto> buscarProdutos(Produto prod)
21.  throws UnknownHostException {
22.
23.    DBCollection table = ConexaoMongo.getConnection("produtos");
24.    BasicDBObject searchQuery = new BasicDBObject();
25.    searchQuery.put("ativo", prod.getAtivo());
26.
27.    DBCursor cursor = collection.find(searchQuery);
28.    List<Produto> list = new ArrayList<Produto>();
29.
30.    while (cursor.hasNext()) {
31.
32.      DBObject obj = cursor.next();
33.      Map map = ProdutoService.converteDocumentoParaMap(obj.toString());
34.      Set<Entry<String, String>> set = map.entrySet();
35.      Iterator<Entry<String, String>> itr = set.iterator();
36.      Produto produto = new Produto();
37.
38.      while (itr.hasNext()) {
39.
40.        Entry<String, String> entry = itr.next();
41.        String key = entry.getKey();
42.        key = key.substring(2,key.length()-2);
43.        String value = entry.getValue();
44.
45.        if (key.equalsIgnoreCase("codigo")) {
46.          produto.setCodigo(value.substring(2, value.length() - 2));
47.        } else if (key.equalsIgnoreCase("descricao")) {
48.         produto.setDescricao(value.substring(2, value.length() - 2))
49.        } else if (key.equalsIgnoreCase("preco")) {
50.         produto.setPreco(value.substring(2, value.length() - 2));
51.        } else if (key.equalsIgnoreCase("ativo")) {
52.         produto.setAtivo(value.substring(2, value.length() - 2));
53.        }
54.
55.      }
56.
57.      list.add(produto);
58.    }
59.    return list;
60.  }
61.
62.  public static Map<String, String> converteDocumentoParaMap(String s) {
63.    s = s.substring(1, s.length() - 1);
64.    String sArr[] = s.split(",");
65.    Map<String, String> map = new LinkedHashMap<String, String>();
66.    for (int i = 1; i < sArr.length; i++) {
67.      if (!sArr[i].contains("$date")) {
68.      String keyValue[] = sArr[i].split(":");
69.        map.put(keyValue[0], keyValue[1]);
70.        System.out.println(keyValue[0] + "," + keyValue[1]);
71.      } else {
72.        String keyValue[] = sArr[i].split(":");
73.        map.put(keyValue[0], keyValue[2]);
74.      }
75.    }
76.    return map;
77.  }
78. }
```

Para isso, ele recebe como parâmetro o objeto **Produto** com as informações que serão gravadas. Na primeira instrução, obtemos a conexão com nossa base de dados. Em seguida, instanciamos um objeto da classe **BasicDBObject**, que nos permite inserir um novo documento na base de dados. Deste modo, após obter a instância, informamos através do método **put()** os campos que serão inseridos e seus devidos valores – neste caso, os valores carregados no objeto **Produto**. Por último, efetuamos o cadastro do novo registro ao executar do método **insert()**.

O próximo método a ser analisado é **buscarProdutos()**. Em seu código, estabelecemos a conexão com o banco ao chamar **getConnection()** e instanciamos o objeto **BasicDBObject**. Em seguida, através do método **put()** informamos ao objeto instanciado o atributo que é recuperado ao chamar **prod.getAtivo()**, indicando que devem ser buscados todos os produtos que estiverem com o status ativo (vide linha 25). Logo após, na linha 27, é efetuada a consulta através do método **find()**. Após isso, os valores retornados são armazenados no objeto **cursor**. Para que estes dados do documento possam ser exibidos em forma de lista, criamos o método **converteDocumentoParaMap()**, que será invocado a cada iteração do bloco **while** na linha 30. Este procedimento é necessário porque não há como exibir diretamente o conteúdo do objeto **cursor** em uma lista. Na linha 36 criamos um novo objeto e o nomeamos como **produto**, o qual será preenchido com os dados convertidos pelo método **converteDocumentoParaMap()**. Entre as linhas 38 e 55 é realizada uma verificação para que os valores das colunas da collection sejam atribuídos ao seu devido atributo do objeto **produto**. Por fim, adicionamos este objeto a uma lista na linha 57.

Criando um relatório com iReport

Antes de iniciarmos a criação do relatório é necessário estabelecer a conexão entre o banco de dados e o iReport, para possibilitar a busca de informações que serão visualizadas durante o desenvolvimento na IDE. Para isso, após iniciar o iReport, clique no botão que simboliza uma tomada na tela inicial e depois em *New*, para criar uma nova conexão. Em seguida, será apresentada uma lista com os tipos de conexão que o iReport aceita; escolha a opção *MongoDB Connection*. Para finalizar, preencha os campos

Mongo URI com a URL *mongodb://127.0.0.1:27017/test* e deixe vazio os campos *User Name* e *Password*, pois estamos utilizando o banco de dados *test*, que por padrão não necessita de informações de login. Caso todos os passos tenham sido realizados corretamente, após clicar no botão *Test* deverá aparecer a mensagem *Connection Test Successful*.

Com a conexão estabelecida, podemos iniciar a criação de um relatório que irá se conectar ao MongoDB para buscar as informações a serem apresentadas. Portanto, clique em *New* no menu principal do iReport, escolha o modelo de relatório a ser utilizado (em nosso caso selecionamos o modelo em branco) e clique em *Open This Template*. Com isso, o relatório modelo já estará criado e, a partir de então, conseguiremos dar início ao desenvolvimento do mesmo.

Após criar o relatório veremos uma tela semelhante à da **Figura 1**. Neste momento, clique no ícone ao lado direito da palavra *Preview* e inclua a query **{'collectionName' : 'clientes', 'findQuery' : {'codigo' : {'$eq' : $P{pCodigoCliente}}}}**. Esta query trará todos os clientes cadastrados em nosso banco de dados mediante o código informado no parâmetro **pCodigoCliente**.

Conforme explicado anteriormente, quando criamos um relatório no iReport, um arquivo XML com o código do relatório é criado. A partir de então, todo procedimento realizado visualmente, como a adição de uma caixa de texto, é mapeado para adição/alteração/remoção do código relacionado neste arquivo. Apesar desta facilidade, caso prefira, o desenvolvedor também pode trabalhar diretamente no XML. O código fonte do relatório pode ser acessado através da opção *XML* (vide **Figura 2**).

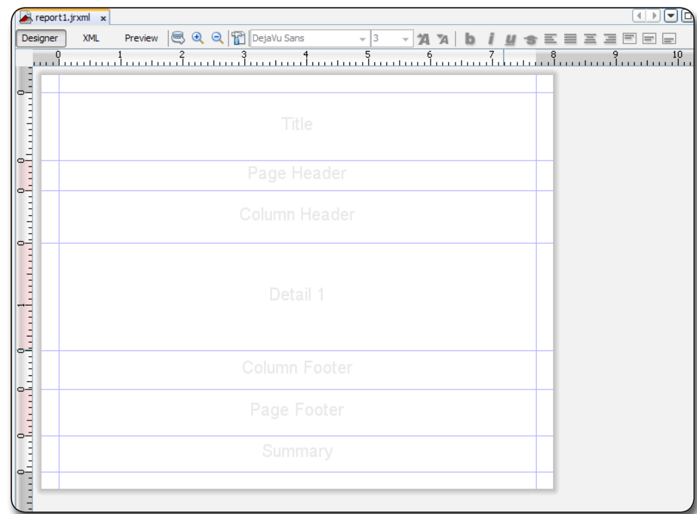
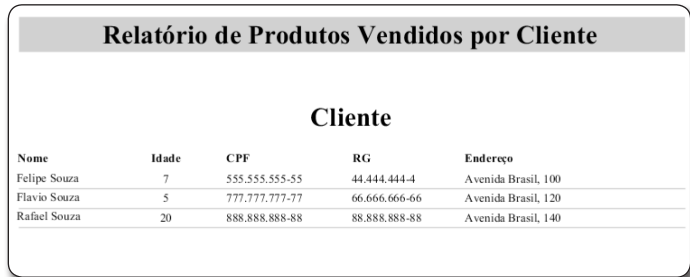


Figura 1. Novo relatório do iReport



Relatório de Produtos Vendidos por Cliente

Cliente

Nome	Idade	CPF	RG	Endereço
Felipe Souza	7	555.555.555-55	44.444.444-4	Avenida Brasil, 100
Flavio Souza	5	777.777.777-77	66.666.666-66	Avenida Brasil, 120
Rafael Souza	20	888.888.888-88	88.888.888-88	Avenida Brasil, 140

Figura 2. Relatório gerado

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

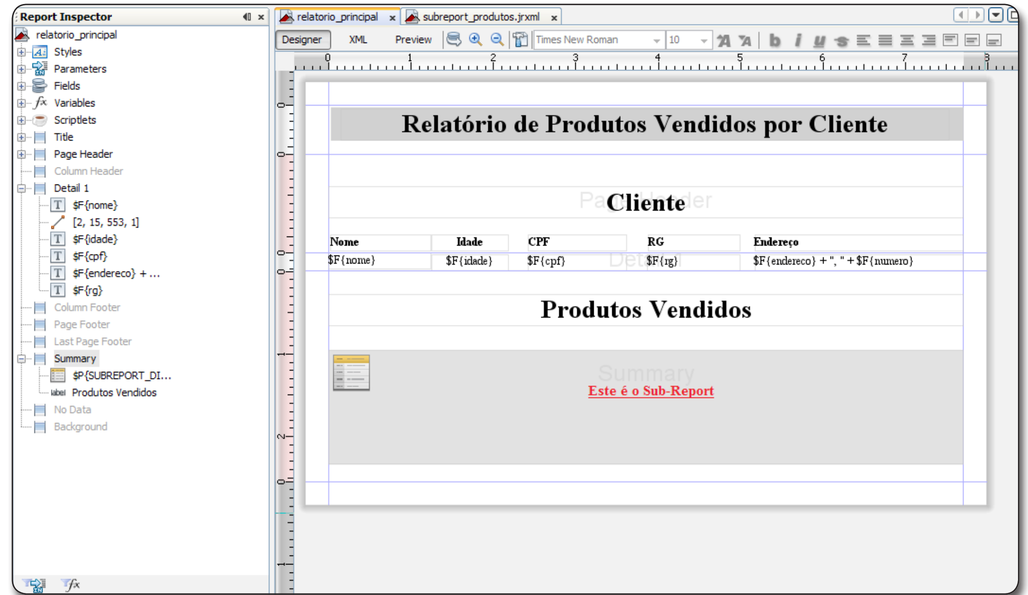


Figura 3. Relatório principal com o sub-relatório

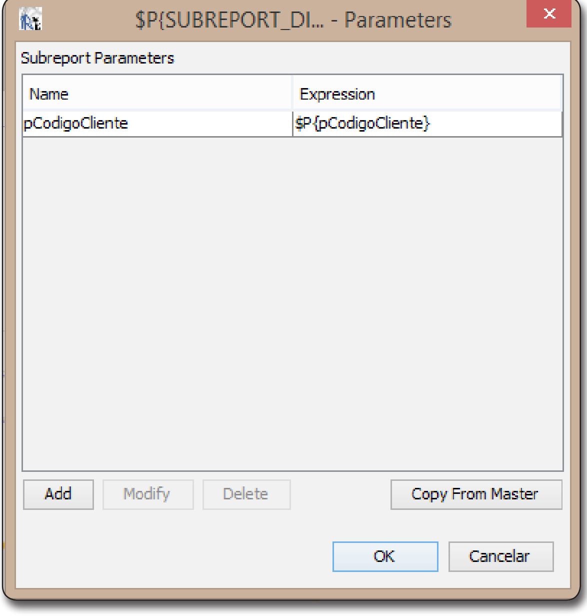


Figura 4. Configuração do parâmetro do sub-relatório

Para apresentar os dados retornados pela query, utilizamos o componente **TextField**. Nele, definimos no parâmetro *Text Field Expression* de qual coluna da collection desejamos que as informações sejam exibidas. Após montar o relatório, clique em *Preview* para compilá-lo e ver como ele ficou. A **Figura 2** exibe o resultado.

Criando sub-relatórios

Uma das melhores funcionalidades fornecidas pelo iReport é a possibilidade de incluir sub-relatórios em um relatório principal. Este recurso é bastante útil caso o desenvolvedor queira organizar melhor a apresentação dos dados. Além disso, muitas vezes também precisamos exibir valores provenientes de queries que não possuem relações (joins) com a query do relatório principal. Nestes casos, o uso de sub-relatórios também se torna um grande diferencial.

Para incluir um sub-relatório ao relatório principal, acesse a paleta de ferramentas do JasperReports, selecione o componente *SubReport* e o arraste até a aba *Summary*. O resultado será semelhante ao exibido na **Figura 3**. Em seguida, será exibida a tela onde você deverá configurar o sub-relatório, informando o nome e modelo de sub-relatório que deseja utilizar, a query que irá recuperar os dados a serem exibidos e se você deseja utilizar a mesma conexão que o relatório principal adota para se conectar ao banco. Neste exemplo faremos uso da mesma conexão. A query a ser incluída no sub-relatório é apresentada a seguir:

```
{'collectionName': 'vendas', 'findQuery': {'codigocliente': {'$eq': $P{pCodigoCliente}}}}
```

Esta query, que irá retornar os produtos vendidos, recebe por parâmetro o mesmo valor informado no parâmetro **pCodigoCliente** da query responsável por retornar o cliente, pois desta forma retornará apenas os produtos vendidos para este mesmo cliente. Para isso, acesse as propriedades do sub-relatório, procure pela propriedade *parameters* e clique duas vezes sobre ela. Logo após, clique no botão *add* e informe o nome do parâmetro como **pCodigoCliente**. No campo *Expression*, informe **$P{pCodigoCliente}**, conforme exibe a **Figura 4**.

Assim, caso você tenha inserido os registros manualmente, como mencionado no início do artigo, ao gerar o relatório devem ser exibidos o cliente – cujo o código é o informado na passagem de parâmetro – e os produtos vendidos para ele. Após montar o relatório, clique em *Preview* para compilá-lo e ver o resultado. A **Figura 5** exibe o relatório com o sub-relatório gerado.

Gerando o relatório pela aplicação JSF

Com a estrutura do projeto pronta, implementaremos agora os métodos necessários para a execução do relatório Jasper através da aplicação JSF. Para isso, criaremos um servlet a ser executado diretamente do HTML e uma classe para auxiliar o servlet a gerar o relatório. As **Listagens 5** e **6** exibem, respectivamente, o código do servlet, que chamamos de **RelatorioServlet**, e o código da classe auxiliar, **RelatorioUtil**.

No código do servlet (vide **Listagem 5**), definimos três variáveis do tipo **String**: duas para armazenar o caminho dos arquivos *relatorio.jasper* e *relatorio.jrxml*, que serão necessários para gerar o

Listagem 5. Código do servlet RelatorioServlet.

```
01. @WebServlet("/relatorioServlet")
02. public class RelatorioServlet extends HttpServlet {
03.
04.    private static final long serialVersionUID = 1L;
05.
06.    public RelatorioServlet() {
07.     super();
08.    }
09.
10.     protected void doGet(HttpServletRequest request,
11.        HttpServletResponse response) throws ServletException, IOException {
12.
13.         String caminhoRelatorioJasper = request.getSession().
14.              request.getRealPath("resources") + File.separator + "relatorio.jasper";
15.
16.       String caminhoRelatorioJrxml = request.getSession().
17.          request.getRealPath("resources") + File.separator + "relatorio. jrxml ";0
18.
19.       String nomeRelatorio = "relatorio.pdf";
20.
21.       // compila o relatório em tempo de execução
22.       try {
23.          JasperCompileManager.compileReportToFile(caminhoRelatorioJrxml,
24.               caminhoRelatorioJasper);
25.       } catch (JRException e2) {
26.          e2.printStackTrace();
27.       }
28.
29.       InputStream inputStream = new FileInputStream(caminhoRelatorioJasper);
30.
31.       Map<String, Object> parametros = new HashMap<String, Object>();
32.
33.       String mongoURI = "mongodb://127.0.0.1/test";
34.       MongoDbConnection connection = null;
35.
36.       try {
37.          connection = new MongoDbConnection(mongoURI, null, null);
38.
39.          final String codigo = request.getParameter("pCodigoCliente");
40.         parametros.put(MongoDbDataSource.CONNECTION, connection);
41.          parametros.put("pCodigoCliente", codigo);
42.
43.          RelatorioUtil.createPDFReport(inputStream, parametros, connection,
44.               response, nomeRelatorio);
45.
46.       } catch (Exception e1) {
47.         e1.printStackTrace();
48.       }
49.    }
50.
51. }
```

Relatório de Produtos Vendidos por Cliente

Nome	Idade	CPF	RG	Endereço
Paulo Santos	25	758.456.155-55	55.555.555-5	Rua Maria Freitas, 3211

Produtos Vendidos

Descrição Produto	Quantidade	Valor	Data Venda
4 - Café	20	40,00	25/03/2015
3 - Leite	5	30,00	12/03/2015

Figura 5. Exibição final do relatório com sub-relatório

relatório pela aplicação; e uma para armazenar o nome do arquivo a ser gerado para o usuário (neste caso, *relatorio.pdf*). Para obter o caminho destes arquivos, utilizamos o método **getRealPath()**. Este retornará o diretório e, então, concatenamos com o nome dos arquivos desejados (*relatorio.jasper* e *relatorio.jrxml*).

Logo após, instanciamos um objeto do tipo **FileInputStream** informando como parâmetro o caminho do arquivo *relatorio.jasper*, que contém os dados (bits) necessários para a classe **JasperPrint** preencher o relatório (veremos como isso funciona adiante). Estabelecemos, então, a conexão com o banco de dados para que a query que incluímos no relatório seja executada e os dados sejam apresentados no PDF. Para isso, definimos em uma variável do tipo **String** a URL do banco de dados e informamos esta variável como parâmetro para instanciar a classe **MongoDbConnection**, que além deste parâmetro recebe o nome de usuário e a senha para autenticação. Como neste exemplo não utilizaremos credenciais para autenticação, podemos informar os parâmetros de usuário e senha como **null**. Por fim, chamamos o método **createPDFReport()**, proveniente da classe **RelatorioUtil**. Esta, por sua vez, tem o objetivo de gerar o relatório e disponibilizá-lo para o usuário, seja em forma de download, onde o arquivo é baixado para o computador, seja abrindo-o diretamente no browser. Ambas as opções podem ser definidas no código desta classe.

No código da **Listagem 6**, temos apenas o método **createPDFReport()**, onde definimos algumas configurações para a geração do relatório. Neste método, definimos na linha 10 que o tipo de arquivo a ser gerado será um PDF e na linha 13 forçamos o download deste arquivo, ou seja, ele não será exibido no browser. Contudo, caso opte pela exibição no browser, comente essa linha de código. Em seguida, na linha 19, criamos o **JasperPrint**, que gera a versão do relatório preenchida. Esse objeto representa o relatório finalizado, e a partir dele podemos enviar para impressão diretamente ou exportar para outro formato, tal como o PDF. Em nosso código, exportamos o relatório através das linhas 23 a 31.

Criando a interface web

Com o banco de dados pronto e as classes necessárias para o funcionamento da aplicação implementadas, criaremos neste momento a interface web. Para isso, importe para o projeto as bibliotecas informadas no início do artigo e crie uma página XHTML.

Nesta página, adicionaremos alguns componentes do PrimeFaces para cadastrar e consultar as informações armazenadas no MongoDB (vide **Figura 6**). A **Listagem 7** exibe a primeira parte do código XHTML, local onde serão exibidos os filtros responsáveis pelas consultas e os botões que irão abrir os formulários de cadastro e a lista vendas.

No código exibido nessa listagem, definimos a estrutura da página principal onde iremos realizar a consulta e exibição das vendas cadastradas.

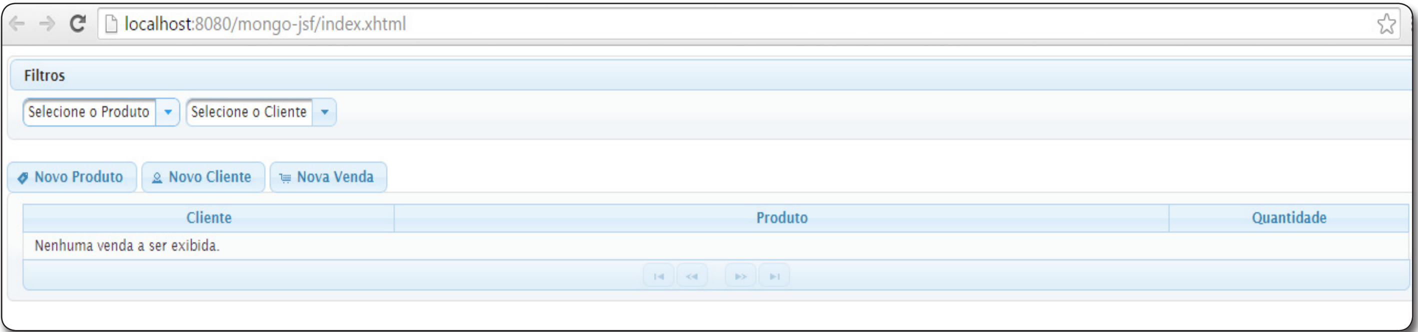


Figura 6. Tela principal da aplicação

Listagem 6. Código da classe RelatorioUtil.

```
01. public class RelatorioUtil {
02.
03.  public static OutputStream createPDFReport(InputStream inputStream,
04.    Map<String, Object> parametros,
05.     MongoDbConnection conexao,
06.     HttpServletResponse response
07.     String nomeRelatorio) throws JRException, IOException {
08.
09.   //Configura o content type do response
10.   response.setContentType( "application/pdf");
11.
12.   //Forçando o download
13.   response.setHeader("Content-Disposition", "attachment;
      filename=" + nomeRelatorio);
14.
15.   //Obtém o OutputStream para escrever o relatório
16.   OutputStream out = response.getOutputStream();
17.
18.   //Cria um JasperPrint, que é a versão preenchida do relatório usando
      //uma conexão.
19.   JasperPrint jasperPrint = JasperFillManager.fillReport(inputStream, parametros,
20.    conexao);
21.
22.   // Exporta em PDF, escrevendo os dados no outputstream do response.
23.   JRExporter exporter = new JRPdfExporter();
24.   exporter.setParameter( JRExporterParameter.JASPER_PRINT, jasperPrint );
25.   exporter.setParameter( JRExporterParameter.OUTPUT_STREAM, out );
26.
27.   //Gera o relatório
28.   exporter.exportReport();
29.
30.   //Tetorna o OutputStream
31.   return out;
32.
33.  }
34. }
```

Listagem 7. Código do layout da página principal contido no arquivo index.xhtml.

```
01. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
02.  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
03. <html xmlns="http://www.w3.org/1999/xhtml"
04.        xmlns:h="http://java.sun.com/jsf/html"
05.        xmlns:f="http://java.sun.com/jsf/core"
06.        xmlns:p="http://primefaces.org/ui"
07.        xmlns:ui="http://java.sun.com/jsf/facelets"
08.        xmlns:fn="http://java.sun.com/jsp/jstl/functions">
09.
10.  <h:head>
11.
12.  </h:head>
13.  <h:body>
14.   <f:metadata>
15.    <f:event listener="#{vendasBean.init()}"
16.      type="preRenderView"></f:event>
17.    </f:metadata>
18.
19.    <p:growl id="messages" />
20.
21.    <h:form id="formFiltros">
22.     <p:panel header="Filtros">
23.
24.     </p:panel>
25.
26.    </h:form>
27.
28.    <p:spacer height="3"></p:spacer>
29.
30.    <h:form id="formList">
31.
32.      <p:panel>
33.
34.       <p:dataTable
35.        value="#{vendasBean.list}"
36.        var="vendas" emptyMessage="Nenhuma venda a ser exibida."
37.        paginator="true" rows="10" paginatorPosition="bottom"
38.        scrollable="false" id="dataTable">
39.
40.        <p:column headerText="Cliente" style="width: 13px; text-align:
41.          center;">
42.         <h:outputText value="#{vendas.codigoCliente} –
43.           #{vendas.nomeCliente}" />
44.        </p:column>
45.
46.        <p:column headerText="Produto" style="width: 50px; text-align:
47.         center;">
48.         <h:outputText value="#{vendas.codigoProduto} –
49.           #{vendas.descricaoProduto}" />
50.        </p:column>
51.
52.        <p:column headerText="Quantidade" style="width: 1px; text-
53.          align: center;">
54.         <h:outputText value="#{vendas.quantidade}" />
55.        </p:column>
56.
57.       </p:dataTable>
58.      </p:panel>
59.     </h:form>
```

Neste bloco, inicialmente utilizamos o componente **<f:metadata>**, nas linhas 14 a 17, para executar ações (métodos) ao iniciar o carregamento da página; neste caso, ele irá executar o método **init()**, que é responsável por chamar a busca de produtos e clientes, informações que serão úteis para a consulta das vendas. Em seguida, na linha 19, declaramos o componente **<p:growl>** para exibir mensagens ao usuário após realizar o cadastro de produtos, clientes e ou vendas como, por exemplo: "Venda cadastrada com sucesso" e "Erro ao cadastrar venda". Depois disso, construímos duas estruturas de formulário: uma entre as linhas 21 e 26, que irá conter os componentes **<p:selectOneMenu>** para exibir produtos e clientes; e outra entre as linhas 30 e 59, que contém o componente **<p:dataTable>** para listar as vendas cadastradas.

Após criar o layout da página principal, implementaremos os filtros para realizar a consulta das vendas. Sendo assim, adicionaremos o componente **<p:selectOneMenu>** para exibir produtos e clientes que o usuário poderá selecionar para efetuar a consulta, como expõe a **Listagem 8**. Vale ressaltar que este código deve ser inserido entre as linhas 22 e 24 da listagem anterior.

Listagem 8. Código dos filtros utilizados para a consulta de vendas.

```
01.   <p:selectOneMenu id="produto-consulta"
        value="#{vendasBean.codigoProduto}">
02.
03.     <f:selectItem itemLabel="Selecione o Produto"
04.       itemValue="0"></f:selectItem>
05.     <f:selectItems value="#{vendasBean.listProdutos}" var="pro"
06.       itemLabel="#{pro.descricao}" itemValue="#{pro.codigo}" />
07.     <p:ajax event="change" listener="#{vendasBean.listarVendas()}"
08.       update=":formList" execute="@this" />
09.   </p:selectOneMenu>
10.
11.   <p:selectOneMenu id="cliente-consulta"
12.     value="#{vendasBean.codigoCliente}">
13.     <f:selectItem itemLabel="Selecione o Cliente"
14.       itemValue="0"></f:selectItem>
15.     <f:selectItems value="#{vendasBean.listClientes}" var="cli"
16.       itemLabel="#{cli.nome}" itemValue="#{cli.codigo}" />
17.     <p:ajax event="change" listener="#{vendasBean.listarVendas()}"
18.       update=":formList" execute="@this" />
19.   </p:selectOneMenu>
```

Demonstramos nesse bloco a utilização de dois componentes **<p:selectOneMenu>**: um para listar os clientes e outro para listar os produtos cadastrados no banco de dados. A partir disso, após o usuário selecionar o item em algum dos dois componentes, o método **listarVendas()**, especificado nas linhas 07 e 17, será acionado. Feito isso, a busca dos dados será realizada e o resultado atribuído à lista **listaVendas**. Para fazer com que o conteúdo dessa lista seja exibido no **<p:dataTable**>, informamos no atributo **value** deste componente a lista que possui os dados das vendas, conforme a linha 35 da **Listagem 7**.

Já na **Listagem 9** declaramos três componentes **<p:commandButton>**, que serão utilizados para abrir os formulários onde iremos cadastrar os produtos, os clientes e as vendas. Em cada um deles, definimos que ao completar a ação de chamada do botão (veja as linhas 2, 9 e 17), o formulário será exibido através do comando **PF().show()**. Além disso, declaramos também o botão que irá realizar a chamada do servlet responsável pela impressão do relatório. Este código deve ser inserido na linha 31 da **Listagem 7**.

Listagem 9. Código dos botões que irão abrir os formulários para cadastro e impressão do relatório.

```
01. <p:commandButton icon="ui-icon-tag" value="Novo Produto"
02.       styleClass="ui-priority-primary" oncomplete="PF('produtoDlg').show()"
03.         process="@this">
04. </p:commandButton>
05.
06. <p:spacer width="3"></p:spacer>
07.
08. <p:commandButton icon="ui-icon-person" value="Novo Cliente"
09.       styleClass="ui-priority-primary" oncomplete="PF('clienteDlg').show()"
10.         process="@this">
11. </p:commandButton>
12.
13. <p:spacer width="3"></p:spacer>
14.
15. <p:commandButton icon="ui-icon-cart" value="Nova Venda"
16.       styleClass="ui-priority-primary"
17.         oncomplete="PF('novaVendaDlg').show()" process="@this">
18. </p:commandButton>
19.
20.   <p:button href="/relatorioServlet?pCodigoCliente=#{vendasBean.codigoCliente}"
21.       icon="ui-icon-document" styleClass="ui-priority-document"
22.       value="Emitir Relatório"></p:button>
```

Por sua vez, a **Listagem 10** apresenta o código de um dos três formulários criados para cadastro; neste caso, o código do formulário para cadastro de produtos. Como todos realizam o mesmo procedimento, tendo praticamente os mesmos códigos, não exibiremos os outros dois.

Dentro do formulário, definimos na linha 2 o componente **<p:dialog>**, utilizado para exibir o formulário para o usuário em uma tela de diálogo, conforme exibe a **Figura 7**. Após isso, declaramos dois componentes **<p:inputText>**, onde serão informados pelo usuário a descrição e o preço do produto, e um componente **<p:selectOneMenu>**, onde serão exibidas as opções "Ativo" e "Inativo", para que o usuário defina se o produto está disponível ou não para ser vendido. Por fim, adicionamos dois botões, sendo o primeiro para cancelar a operação e o segundo para salvar o produto. Este código deve ser inserido após a linha 59 da **Listagem 7**.

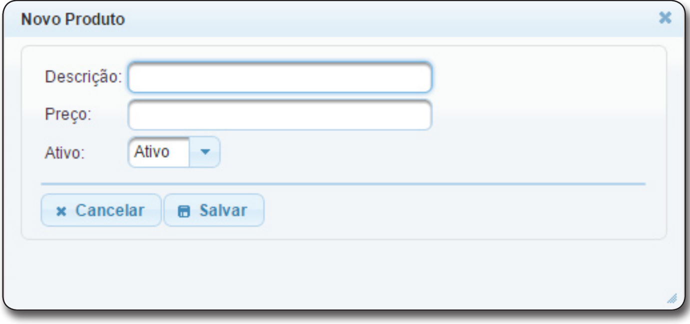


Figura 7. Formulário de cadastro

Listagem 10. Código do formulário de cadastro de produtos.

```
01. <h:form id="formCadastroProduto" onkeypress="return event.keyCode != 13">
02.   <p:dialog id="produtoDialog" header="Novo Produto"
03.     widgetVar="produtoDlg" modal="true" height="192" width="509">
04.
05.     <p:panel style="border: 1px solid #dddddd !important;"
06.       id="panelProduto">
07.
08.       <p:messages id="messages" />
09.
10.       <h:panelGrid columns="2">
11.
12.         <p:outputLabel value="Descrição: " />
13.
14.         <p:columnGroup>
15.           <p:inputText maxlength="60" size="40" id="descricao"
16.             value="#{vendasBean.produto.descricao}" />
17.         </p:columnGroup>
18.
19.         <p:outputLabel value="Preço: " />
20.
21.         <p:columnGroup>
22.         <p:inputText maxlength="60" size="40" id="preco"
23.           value="#{vendasBean.produto.preco}" />
24.       </p:columnGroup>
25.
26.       <p:outputLabel value="Ativo: " />
27.
28.     <p:columnGroup>
29.
30.         <p:selectOneMenu id="ativo" value="#{vendasBean.produto.ativo}">
31.             <f:selectItem itemLabel="Ativo" itemValue="S"></f:selectItem>
32.             <f:selectItem itemLabel="Inativo" itemValue="N"></f:selectItem>
33.             <p:ajax event="change" process="@this" />
34.           </p:selectOneMenu>
35.         </p:columnGroup>
36.
37.       </h:panelGrid>
38.
39.       <p:separator></p:separator>
40.
41.       <p:commandButton icon="ui-icon-close" value="Cancelar"
42.         oncomplete="PF('produtoDlg').hide()" />
43.
44.       <p:commandButton icon="ui-icon-disk" value="Salvar"
45.         id="btnSalvarProduto" process="@this, descricao, preco, ativo"
46.         styleClass="ui-priority-primary" update=":messages"
47.         actionListener="#{vendasBean.novoProduto}"
48.         oncomplete="PF('produtoDlg').hide()">
49.
50.       </p:commandButton>
51.
52.     </p:panel>
53.
54.   </p:dialog>
55.  </h:form>
56. </h:body>
57. </html>
```

A partir do conteúdo apresentado o leitor será capaz de desenvolver suas primeiras aplicações web utilizando como solução de persistência o banco de dados MongoDB. Como desafio, tente adicionar novos recursos à aplicação exemplo, como novos tipos de consulta, enriquecer o relatório, entre outras funcionalidades. Isso o estimulará a pesquisar por outros recursos do MongoDB e solidificar o assunto abordado. Lembre-se que a prática também é fundamental para que saibamos como explorar, da melhor maneira, os recursos no código.

Autor

Luis Gustavo Souza
os.luisgustavo@gmail.com
Cursa Sistemas de Informação pela Universidade de Franca. Desenvolvedor Java e PL\SQL também na Universidade de Franca, trabalha há cinco anos na área de informática. Gosta de estudar e atuar com tecnologias open source e atualmente tem como objetivo obter a certificação OCA Java.

Links:

Endereço para download do PrimeFaces.
http://www.primefaces.org/downloads

Endereço para download do MongoDB.
http://docs.mongodb.org/ecosystem/drivers/downloads/

Endereço para download do driver do MongoDB para Java.
http://mongodb.github.io/mongo-java-driver/

Endereço para download do iReport.
http://community.jaspersoft.com/project/ireport-designer/releases

Endereço para download do Eclipse.
https://eclipse.org/downloads/

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

Desenvolva aplicações com AngularJS e Java EE

Aprenda neste artigo como implementar aplicações MVC utilizando AngularJS no lado cliente e Java EE no lado servidor

Desde o surgimento da plataforma J2EE, em 1999, a cada nova versão desta especificação, que a partir de 2006 passou a ser chamada de Java EE e atualmente encontra-se na versão 7, novas APIs são adicionadas e outras já consolidadas são aperfeiçoadas, formando um conjunto poderoso de funcionalidades disponíveis aos desenvolvedores.

Estas funcionalidades ajudam a reduzir parte da complexidade da implementação de aplicações, especialmente quando envolvem requisitos não-funcionais relativamente complexos, como transações, web services, cache, balanceamento de carga, etc. Dentre estes requisitos, um dos mais esperados nos últimos anos é a separação física (tiers) e/ou lógica (layers) de camadas, utilizando para isso soluções arquiteturais como o MVC (Model View Control). Neste cenário, a especificação Java EE fornece diversas opções, a saber:

- **JPA (JSR 220):** viabiliza o mapeamento objeto-relacional, aumentando a produtividade na implementação de operações com bancos de dados;
- **CDI (JSR 299):** gerencia as dependências dos objetos, diminuindo o acoplamento entre classes;
- **JAX-RS (JSR 331):** define um padrão para criação de web services de acordo com o REST, permitindo que a camada Model responda a requisições de softwares escritos em diversas tecnologias, o que eleva a interoperabilidade.

Estes recursos geralmente são empregados na camada Model, pois o código-fonte que os utiliza controla dados e regras de negócio da aplicação. Já para as camadas View e Control, existem diversas opções a serem consideradas além das soluções disponibilizadas pela Java EE, especialmente para aplicações web. Para esta finalidade a plataforma enterprise do Java possui algumas especificações, sendo a mais utilizada o JSF, por sua fácil

Fique por dentro

Este artigo é útil por apresentar o desenvolvimento de uma aplicação com o padrão arquitetural MVC utilizando tecnologias conhecidas, como o framework AngularJS e a especificação Java EE. Neste cenário, a camada Model será implementada com CDI, JAX-RS e JPA, da especificação Java EE, a camada Control será escrita utilizando o AngularJS e a camada View será projetada com a linguagem de marcação HTML, as diretivas do AngularJS e os componentes responsivos do Bootstrap.

Será mostrado neste artigo que, com o uso destas tecnologias, os lados cliente e servidor estarão desacoplados um do outro, limitando-se a trocarem informações através do protocolo HTTP e utilizando estruturas de dados em JSON, promovendo maior reaproveitamento de código.

integração com outros recursos do Java, além do vasto conjunto de componentes que são fornecidos por suas implementações, o que possibilita a construção de interfaces ricas. Contudo, este framework possui algumas limitações, como:

- Persiste todo o estado da interface gráfica na memória do servidor, diminuindo o throughput da aplicação;
- Dependência de desenvolvedores Java para construir a camada View;
- Alta curva de aprendizado, pois seu ciclo de vida é relativamente complexo de entender.

Diante disso, diversos frameworks JavaScript têm sido apontados como opções razoavelmente boas para implementar as camadas View e Control. Dentre estes, um dos mais conhecidos atualmente é o AngularJS. Este framework se adapta e estende o HTML tradicional, adicionando conteúdo dinâmico. Além disso, é capaz de abstrair o acoplamento entre o lado cliente e o lado servidor da aplicação, viabilizando que o desenvolvimento desta evolua em ambos os lados de forma paralela, pois a comunicação entre estes lados ocorrerá por um protocolo conhecido (HTTP),

utilizando estruturas de dados que possuem formato também conhecido (JSON).

Além do AngularJS, se faz necessário o uso de um framework que forneça um layout responsivo para as páginas HTML. Layouts responsivos são páginas HTML que automaticamente se adaptam ao dispositivo do usuário, como PC, celular ou tablet, mudando a aparência e disposição com base no tamanho da tela em que elas são exibidas. Uma boa opção para isso é o framework Bootstrap.

Diante do que foi exposto, o objetivo deste artigo é apresentar o desenvolvimento de uma aplicação MVC, onde a camada Model utilizará as especificações Java EE já mencionadas, a camada Control será escrita utilizando o framework AngularJS e a camada View terá como base para sua construção a linguagem de marcação HTML, as diretivas AngularJS e os estilos do Bootstrap. Com isso, será possível aproveitar o que cada tecnologia possui de melhor, maximizando a qualidade e produtividade na implementação de cada camada.

Servidores de aplicação Java EE

Um servidor de aplicação Java EE é um middleware que disponibiliza para suas aplicações todos os recursos definidos em uma versão específica da especificação Java EE. Desta forma, não é necessário efetuar o download de nenhuma API para construir a camada Model da aplicação. É preciso apenas utilizar um servidor de aplicação certificado.

Existem diversos servidores disponíveis no mercado, como: GlassFish e WebLogic da Oracle, WildFly e JBoss AS da Red Hat, WebSphere da IBM, Geronimo da Apache, dentre outros. Para este projeto, será empregado o WildFly 8, por implementar a versão 7 da Java EE e também por este continuar o projeto do servidor de aplicações JBoss AS, que é bem popular no Brasil.

Preparando o ambiente de desenvolvimento

Antes de iniciar o desenvolvimento da aplicação, é necessário ter o JDK 8 instalado (veja o endereço na seção **Links**), pois é a versão recomendada pela Red Hat para ser utilizada juntamente com o WildFly 8.

Com relação à IDE, será adotado o Eclipse Luna, pois esta é a versão que tem o suporte oficial ao Java 8. Além disso, o Eclipse é uma boa escolha pelo seu rico editor de códigos e pela grande quantidade de plugins disponíveis.

Para criação das páginas HTML, será necessário efetuar o download do framework AngularJS. Os arquivos referentes a este framework que devem ser baixados são: *angular-resource.min.js*, *angular-resource.min.js.map*, *angular.min.js* e *angular.min.js.map*. Além do AngularJS, também é preciso efetuar o download do framework Bootstrap. Os endereços para download destes arquivos podem ser verificados na seção **Links**.

Além dos itens citados, é fundamental ter um banco de dados instalado. Neste artigo, será adotado o MySQL na versão 5. Na seção **Links** há um endereço que explica como instalar e configurar este banco de dados.

Por fim, deve-se efetuar o download do servidor de aplicação WildFly, versão 8.2.0. Feito isso, basta descompactar o arquivo em um diretório de sua preferência. Em seguida, deve-se configurar o WildFly para acessar o banco de dados MySQL e também para ser utilizado dentro do Eclipse. Os passos necessários para realizar tais configurações são mostrados a seguir:

1. Passo 1: Instale o plugin JBoss Tools no Eclipse. Para isso, acesse o menu *Help > Eclipse Marketplace*. No campo *Search*, digite o texto: "JBoss Tools". Na sequência, será mostrado o plugin *JBoss Tools (Luna) 4.2.2.Final*. Clique então no botão *Instalar*, conforme mostra a **Figura 1**;

2. Passo 2: Inclua o WildFly 8 entre os servidores mapeados pelo Eclipse. Para isso, acesse o menu *New > Other > Server*. Logo após, selecione os itens *Jboss Community > WildFly 8.x*, conforme a **Figura 2**;

3. Passo 3: Acesse o diretório onde o WildFly 8 foi descompactado e crie a seguinte estrutura de pastas: *wildfly-8.2.0.Final/modules/system/layers/base/com/mysql/main*. Em seguida, faça o download do driver JDBC para o MySQL (veja o endereço na seção **Links**) e copie para o diretório mencionado. Para finalizar, no mesmo diretório, crie um arquivo com o nome de *module.xml* e adicione neste o conteúdo mostrado na **Listagem 1**;

4. Passo 4: Ainda no WildFly 8, acesse o diretório *wildfly-8.2.0.Final/standalone/configuration* e abra o arquivo *standalone.xml* com o editor de sua preferência. Em seguida, procure pela *tag* **<datasources>** e adicione um novo datasource para o MySQL, como mostra a **Listagem 2**.

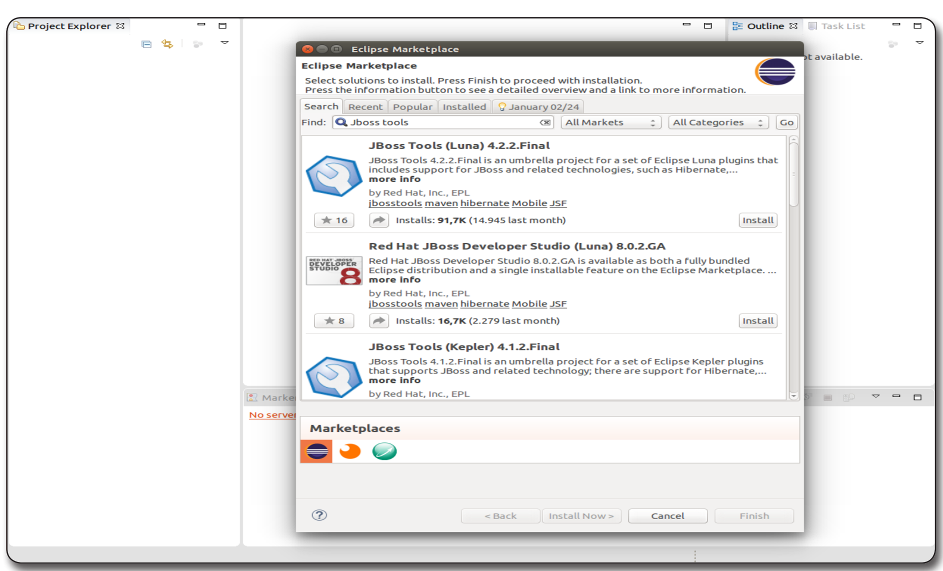


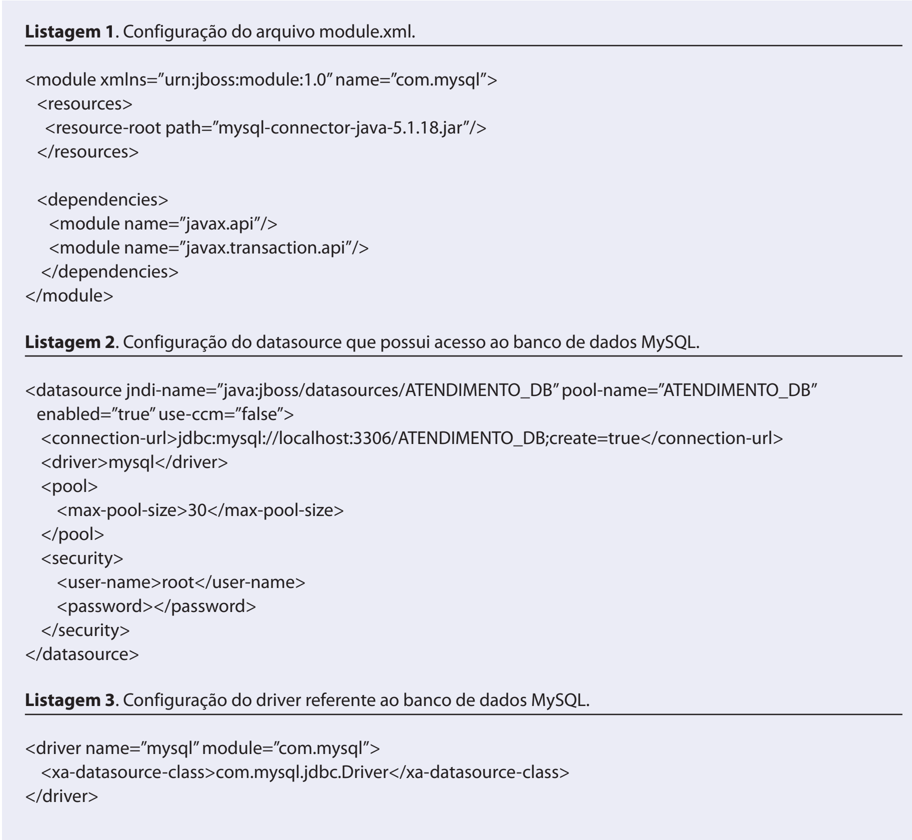
Figura 1. Adicionando o plugin Jboss Tools ao Eclipse

Além do datasource, também é necessário adicionar o driver do MySQL dentro da tag **<drivers>**, de acordo com a **Listagem 3**;

5. **Passo 5:** Por fim, para iniciar o WildFly 8 dentro do Eclipse, selecione o servidor adicionado no passo 2 e clique na opção *Iniciar*. Ao utilizar algum navegador web, será possível acessar a página inicial deste servidor com a URL *http://localhost:8080/*.

O sistema de atendimento

Para apresentar as tecnologias propostas ao longo do artigo, será desenvolvido um sistema que permite ao usuário informar sua opinião sobre um determinado atendimento, registrando sua avaliação e nota. Sendo assim, o sistema apresentará uma tela que permite cadastrar, editar e excluir registros de atendimentos. O código-fonte desta aplicação pode ser obtido na página desta edição da Java Magazine.



Listagem 1. Configuração do arquivo module.xml.

```
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.18.jar"/>
  </resources>

  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
  </dependencies>
</module>
```

Listagem 2. Configuração do datasource que possui acesso ao banco de dados MySQL.

```
<datasource jndi-name="java:jboss/datasources/ATENDIMENTO_DB" pool-name="ATENDIMENTO_DB"
  enabled="true" use-ccm="false">
  <connection-url>jdbc:mysql://localhost:3306/ATENDIMENTO_DB;create=true</connection-url>
  <driver>mysql</driver>
  <pool>
    <max-pool-size>30</max-pool-size>
  </pool>
  <security>
    <user-name>root</user-name>
    <password></password>
  </security>
</datasource>
```

Listagem 3. Configuração do driver referente ao banco de dados MySQL.

```
<driver name="mysql" module="com.mysql">
  <xa-datasource-class>com.mysql.jdbc.Driver</xa-datasource-class>
</driver>
```

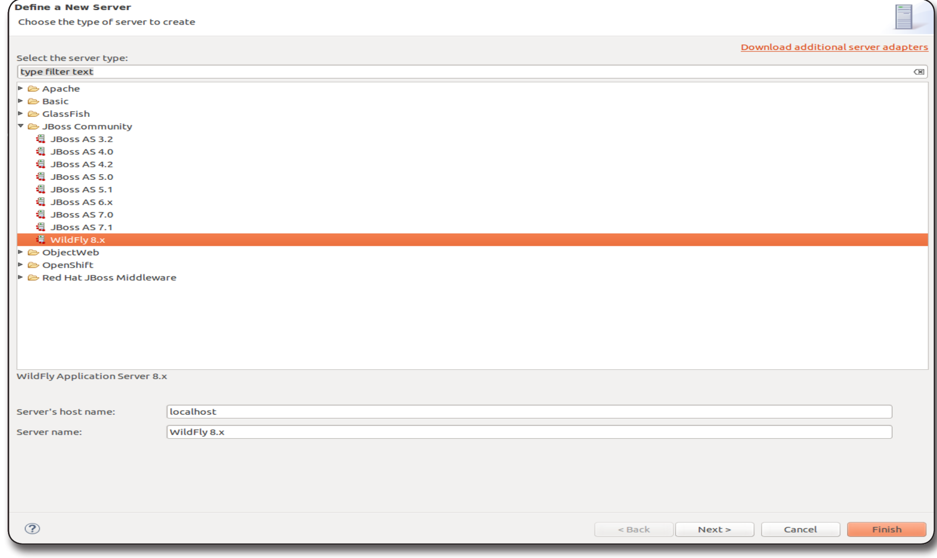
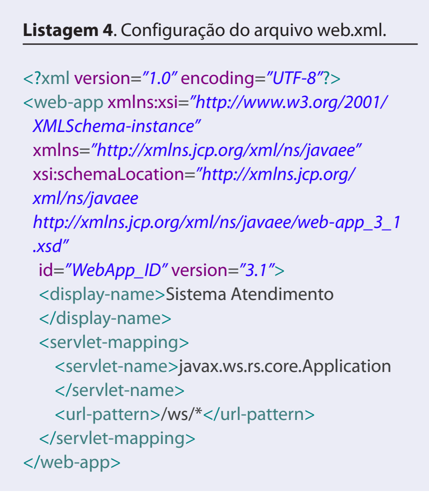


Figura 2. Adicionando o WildFly 8 ao Eclipse

Criando a aplicação no Eclipse

Para iniciar o desenvolvimento, crie um projeto no Eclipse clicando em *New > Dynamic Web Project* e especificando o nome do projeto como *SistemaAtendimento*. Na sequência, selecione as opções indicadas na **Figura 3** e clique em *Finish*.

Com o projeto criado, é necessário adicionar uma configuração no *web.xml* para que o JAX-RS efetue o tratamento das requisições HTTP que possuam URLs dentro de um prefixo específico. A configuração adicionada neste arquivo é mostrada na **Listagem 4**, arquivo este que se encontra no caminho *SistemaAtendimento/WebContent/WEB-INF/web.xml*. Tal configuração define que todas as URLs desta aplicação que possuírem o prefixo */ws/** serão tratadas pelo JAX-RS. Por exemplo, a URL *http://localhost:8080/SistemaAtendimento/ws/* possui o prefixo mencionado.



Listagem 4. Configuração do arquivo web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xsi:schemaLocation="http://xmlns.jcp.org/
  xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1
  .xsd"
   id="WebApp_ID" version="3.1">
   <display-name>Sistema Atendimento
   </display-name>
   <servlet-mapping>
      <servlet-name>javax.ws.rs.core.Application
      </servlet-name>
      <url-pattern>/ws/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Em seguida, deve-se habilitar o CDI no projeto. Para isso, basta adicionar o arquivo *beans.xml* dentro da pasta *SistemaAtendimento/WebContent/WEB-INF*. Este arquivo deve conter apenas a estrutura padrão mostrada na **Listagem 5**. Neste projeto, optou-se por definir as configurações do

CDI usando anotações em classes Java e por isso não será necessário efetuar novas edições neste arquivo.

Após o CDI estar habilitado, a API JPA deve ser configurada. Para tanto, será necessário incluir o arquivo *persistence.xml* na pasta *SistemaAtendimento/src/META-INF/persistence.xml*. A **Listagem 6** mostra o conteúdo deste arquivo. A tag **<persistence-unit>** define uma unidade de persistência para o banco de dados que irá fornecer sessões com o banco para a aplicação. Já a tag **<jta-data-source>** informa o nome do *datasource* configurado no WildFly. Este *datasource* foi definido na tag **<jndi-name>**, mostrada anteriormente na **Listagem 2**. Além disso, ao adicionar a propriedade **hibernate.hbm2ddl.auto** com o valor **create-drop**, define-se que a cada vez que o servidor WildFly for inicializado, as tabelas mapeadas neste sistema serão dropadas e criadas novamente.

Com relação aos arquivos referentes aos frameworks AngularJS e Bootstrap, estes devem ser colocados conforme a estrutura de diretórios mostrada na **Figura 4**.

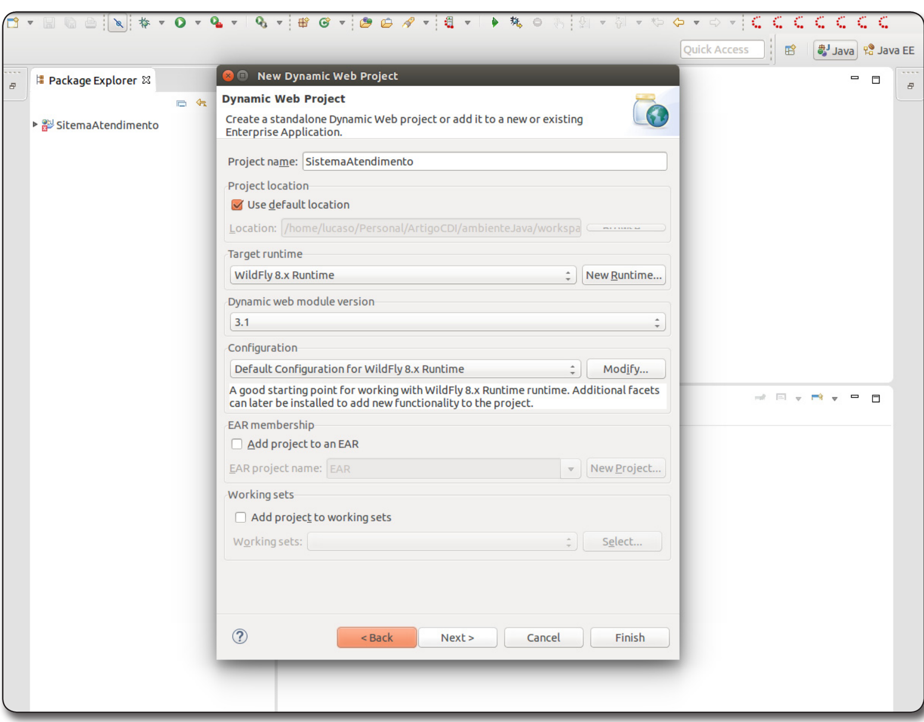


Figura 3. Criando o projeto no Eclipse

Listagem 5. Configuração do arquivo beans.xml.

```
<?xml version="1.0"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://jboss.org/schema/cdi/beans_1_0.xsd" />
```

Listagem 6. Configuração do arquivo persistence.xml.

```
01.  <?xml version="1.0" encoding="UTF-8"?>
02.  <persistence version="2.0"
03.    xmlns="http://java.sun.com/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04.    xsi:schemaLocation="
05.    http://java.sun.com/xml/ns/persistence
06.    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
07.
08.    <persistence-unit name="AtendimentoPU">
09.
10.      <provider>org.hibernate.ejb.HibernatePersistence</provider>
11.
12.      <jta-data-source>java:jboss/datasources/ATENDIMENTO_DB
         </jta-data-source>
13.
14.      <class>br.com.atendimento.model.domain.Atendimento</class>
15.
16.      <properties>
17.        <property name="hibernate.dialect"
             value="org.hibernate.dialect.MySQLDialect" />
18.        <property name="hibernate.hbm2ddl.auto" value="create-drop" />
19.        <property name="hibernate.show_sql" value="true" />
20.      </properties>
21.    </persistence-unit>
22.  </persistence>
```

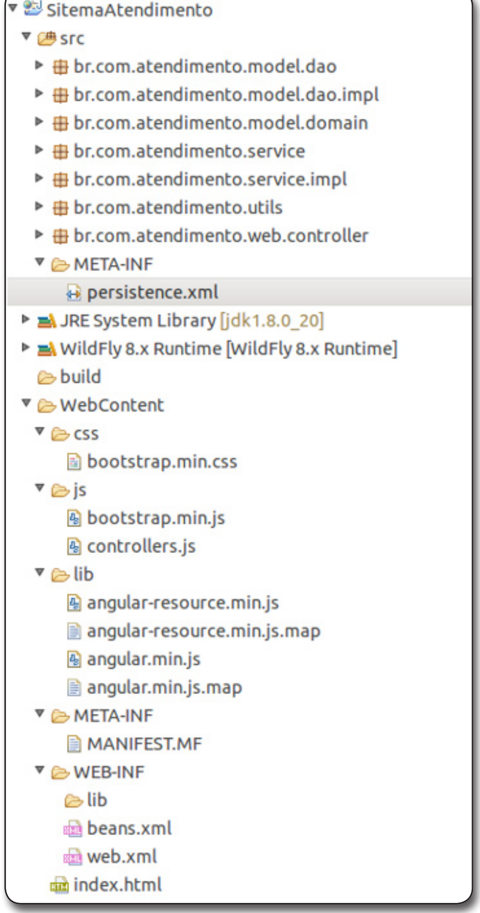


Figura 4. Estrutura de diretórios do projeto SistemaAtendimento

Implementando a camada Model

Com a estrutura do projeto pronta, serão criadas as classes da camada Model. Como já foi dito anteriormente, esta camada contém as classes Java que definem os dados e as regras de negócio da aplicação, utilizando as especificações da plataforma Java EE. Dentro do projeto, a camada está organizada em um pacote de nome **model**, que, por sua vez, possui quatro subpacotes, apresentados a seguir:

- **Domain** – contém as classes que representam os dados e seus comportamentos dentro do sistema;
- **Dao** – contém as classes que abstraem do restante do sistema as operações com o banco de dados, como persistência e recuperação de objetos;
- **Service** – contém as classes que centralizam os serviços envolvidos em uma requisição como, por exemplo, controle de transações, validações de objetos, dentre outros;
- **Facade** – contém as classes que encapsulam toda a complexidade (as regras de negócio) de cada funcionalidade do sistema.

No decorrer deste tópico, serão apresentadas as classes de cada um dos quatro pacotes mencionados. O pacote **Domain** contém a classe **Atendimento**, cujo código é mostrado na **Listagem 7**.

Listagem 7. Código da classe Atendimento.

```
01. @Entity
02. @Table(name = "ATENDIMENTO", schema="ATENDIMENTO_DB")
03. public class Atendimento implements Serializable {
04.
05.     private static final long serialVersionUID = 1L;
06.
07.     @Id
08.     @GeneratedValue
09.     @Column( name = "PROTOCOLO")
10.     private Integer protocolo;
11.
12.     @Column(name = "NOME_CLIENTE")
13.     private String nomeCliente;
14.
15.     @Column(name = "DETALHAMENTO")
16.     private String detalhamento;
17.
18.     @Column(name = "ATENDIMENTO_ENUM")
19.     private NotaAtendimentoEnum atendimentoEnum;
20.
21.     //construtor, getters e setters omitidos
22.      //implementação de equals() e hashCode() utilizando o campo protocolo
```

Nela, podemos verificar que foram utilizadas anotações JPA, mapeando os atributos de objetos do tipo **Atendimento** com colunas e linhas de uma tabela no banco de dados que armazena estes atendimentos. Cada anotação utilizada na classe **Atendimento** tem uma função, a saber:

- **@Entity**: Indica que a classe representa uma entidade no banco de dados;
- **@Table**: Utilizada para definir o nome da tabela e o *schema* do banco de dados;
- **@Id**: Define o campo que será a chave primária da tabela;
- **@GeneratedValue**: Informa que o valor da chave primária será um campo de auto incremento, ou seja, o número do protocolo de atendimento será gerado automaticamente.
- **@Column**: Utilizada para mapear um atributo da classe a uma coluna da tabela, parametrizando informações da coluna como nome, tamanho, se permite valores nulos, dentre outros.

Já o pacote **Dao** contém a classe **AtendimentoDaoImp**, cujo código é mostrado na **Listagem 8**. Como boa prática, os métodos públicos desta classe foram extraídos para uma interface chamada **AtendimentoDao**.

AtendimentoDaoImp, como podemos supor com base em seu nome, disponibiliza comportamentos responsáveis por realizar operações com o banco de dados. Para isso, o atributo **em**, do tipo **EntityManager**, receberá uma sessão com o banco que será injetada automaticamente pelo WildFly, de acordo com a anotação **@PersistenceContext**. Note que o parâmetro **unitName** define qual unidade de persistência (vide **Listagem 6**) será usada para criar as sessões.

A seguir explicamos cada um dos métodos desta classe:

- **salvarOuAtualizar():** este método é responsável por persistir um objeto do tipo **Atendimento** no banco de dados. Quando o atributo **protocolo** for igual a **null**, significa que o método **persist()** irá salvar um novo **Atendimento** no banco de dados. Caso contrário, este método irá atualizar um **Atendimento** já existente. Contudo, para que este objeto seja atualizado, é preciso executar o método **merge()**, para que o mesmo seja colocado em um estado gerenciado (*managed*) pelo JPA. Objetos neste estado são monitorados pelo JPA para verificar quais atributos sofreram alterações e, inclusive, evitar a execução de *updates* no banco caso nenhum atributo tenha sido alterado;
- **remover()**: este método exclui um objeto do tipo **Atendimento** do banco de dados utilizando o método **remove()** da JPA. Como para um objeto ser excluído ele precisa estar em um estado gerenciado, também se faz necessária a execução do método **merge()**. Para mais detalhes sobre o estado dos objetos na JPA, veja a seção **Links**;
- **listar()**: este método retorna todos os registros da tabela *ATENDIMENTO* utilizando o método **createQuery()** da classe **EntityManager**. Para isso, este método recebe uma **String** especificando o critério de consulta no formato JPQL (*Java Persistence Query Language*);
- **findByProtocolo()**: este método utiliza o método **find()** da classe **EntityManager** para retornar um único registro de **Atendimento**, caso exista o **protocolo** no banco de dados.

Os pacotes **Domain** e **Dao** definiram classes que representam as tabelas e realizam operações com o banco de dados. O pacote **Service** irá consumir estas classes, contudo colocando-as dentro de um contexto transacional com a anotação **@Transactional**, como mostra a classe **AtendimentoServiceImpl** na **Listagem 9**. Com esta anotação, cada vez que os métodos **salvar()** ou **remover()** forem executados, uma nova transação será criada e esta receberá um **commit()** caso o método termine de ser executado sem que

tenham sido lançadas exceções, ou um **rollback()** em caso contrário. Outro ponto a se destacar é que várias classes **Dao** podem ser executadas dentro de uma mesma transação, por isso que as transações precisam estar no pacote **Service**.

Listagem 8. Código da classe AtendimentoDaoImpl.

```
01.  public class AtendimentoDaoImpl implements AtendimentoDao {
02.
03.    @PersistenceContext(unitName = "AtendimentoPU")
04.    private EntityManager em;
05.
06.    @Override
07.    public Atendimento salvarOuAtualizar (Atendimento entity) {
08.
09.      if (entity.getProtocolo() != null) {
10.        entity = em.merge(entity);
11.      }
12.
13.      em.persist(entity);
14.
15.      return entity;
16.    }
17.
18.    @Override
19.    public void remover(Atendimento entity) {
20.
21.      entity = em.merge(entity);
22.
23.      em.remove(entity);
24.    }
25.
26.    @Override
27.    @SuppressWarnings("unchecked")
28.    public List<Atendimento> listar() {
29.
30.      Query query = em.createQuery("from Atendimento");
31.
32.      return query.getResultList();
33.    }
34.
35.    @Override
36.    public Atendimento findByProtocolo(Integer protocolo) {
37.
38.      return em.find(Atendimento.class, protocolo);
39.    }
```

Além das transações, também foi usada nesta classe a anotação **@Inject** do CDI. Este recurso deve ser usado nos pontos onde se faz necessária a injeção automática das dependências. Neste caso, a classe **AtendimentoServiceImpl** possui dependência com a interface **AtendimentoDao**.

Com esta anotação não será necessário instanciar a classe **AtendimentoDaoImpl** usando o operador **new**, pois o objeto será instanciado automaticamente pelo CDI. Um benefício do uso desta anotação é o desacoplamento, pois a classe **AtendimentoServiceImpl** não precisará conhecer as implementações da interface **AtendimentoDao**.

Como última observação, assim como no pacote **Dao**, os métodos públicos da classe **AtendimentoServiceImpl** também foram extraídos para uma interface chamada **AtendimentoService**.

Por fim, temos pacote **facade**, que contém uma única classe, chamada **AtendimentoFacade**, mostrada na **Listagem 10**.

Listagem 9. Código da classe AtendimentoServiceImpl.

```
01.  public class AtendimentoServiceImpl implements AtendimentoService {
02.
03.    @Inject
04.    private AtendimentoDao dao;
05.
06.    @Override
07.    @Transactional
08.    public void salvar(Atendimento atendimento) {
09.
10.      dao.salvarOuAtualizar(atendimento);
11.    }
12.
13.    @Override
14.    @Transactional
15.    public void remover(Integer protocolo) {
16.
17.      dao.remover(dao.findByProtocolo(protocolo));
18.    }
19.
20.    @Override
21.    public List<Atendimento> listarTodos() {
22.
23.      return dao.listar();
24.    }
25.
26.  }
```

Listagem 10. Código da classe AtendimentoFacade.

```
@Path("/atendimento")
@Produces({ "application/json" })
public class AtendimentoFacade {

  @Inject
  private AtendimentoService service;

  @GET
  @Path("/")
  public List<Atendimento> listarTodos() {
    return service.listarTodos();
  }

  @POST
  @Path("/")
  public void salvar(Atendimento atendimento) {
    service.salvar(atendimento);
  }

  @DELETE
  @Path("/{protocolo}")
  public void remover(@PathParam("protocolo") Integer protocolo) {
    service.remover(protocolo);
  }
}
```

Esta classe será responsável por toda a integração com as páginas HTML por meio de serviços REST, e também pelo acesso aos métodos da camada de serviço, fazendo o papel de uma interface entre as páginas HTML e a camada de negócio da aplicação.

O conteúdo da classe **AtendimentoFacade** será explicado a seguir. Nesta classe, as requisições HTTP que são recebidas pelos serviços REST serão mapeadas por URIs utilizando a anotação **@Path**. A anotação **@Produces({"application/json"})** indica que as respostas das requisições HTTP serão no formato JSON. Já a anotação **@Inject**, que já foi explicada anteriormente, será

responsável por injetar automaticamente a dependência desta classe, que é a interface **AtendimentoService**.

A execução dos métodos públicos desta classe depende de qual verbo foi solicitado via HTTP. Em nosso exemplo é possível receber requisições HTTP com os verbos GET, POST ou DELETE. Com isso, a cada nova requisição HTTP, o método que estiver com a anotação correspondente (**@GET**, **@POST** ou **@DELETE**) será executado. Por fim, temos a anotação **@PathParam**, que indica o parâmetro enviado nas requisições do tipo DELETE. Por exemplo, caso a URI *http://localhost:8080/SistemaAtendimento/ws/atendimento/7747* seja chamada com o verbo DELETE, será executado o método **remover()** com o parâmetro **protocolo** recebendo o número inteiro **7747**.

Implementando a camada Controller

Neste projeto, a camada Controller se resume a um arquivo JavaScript que irá efetuar requisições HTTP à camada Model, consumindo seus serviços mediante eventos recebidos da camada View. Além disso, esta camada também irá controlar a lista dos atendimentos que serão mostrados na camada View. Este arquivo pode ser verificado na **Listagem 11** e se chama *controllers.js*, situado em *SistemaAtendimento/WebContent/js*.

Na linha 1, é criado um módulo com o nome **crudAtendimento**. Módulos são uma forma de encapsular funcionalidades do AngularJS, para que estas sejam utilizadas em qualquer página HTML do sistema. Uma destas funcionalidades são os serviços do tipo **ngResource**, que fornecem suporte à interação com web services REST. Outra funcionalidade são os controllers, que possuem as informações necessárias à camada view, como dados a serem exibidos em tabelas HTML, ou funções JavaScript, que serão executadas quando um botão for clicado. A linha 3 mostra a criação de um controller chamado **AtendimentoController**, onde são utilizados dois parâmetros: **$resource**, que é um tipo de *data source* para serviços REST; e **$scope**, que é uma espécie de "ponteiro" para qualquer método ou objeto declarado dentro do Controller.

Nas linhas 5 e 6 são declaradas as variáveis que apontam para as URLs definidas na camada Model. Já na linha 8 é declarada a lista de **atendimentos**, que requisita todos os atendimentos ao serviço REST. Ao executar a função **query()**, o *data source* automaticamente faz uma requisição HTTP do tipo GET.

A função **salvar** é definida entre as linhas 10 e 16. Neste bloco, primeiramente é criada uma nova instância do *data source* passando como argumento os atributos do objeto **atendimento** que foram preenchidos na camada **view**. Ao executar a função **save()**, o *data source* automaticamente faz uma requisição HTTP com o verbo POST. Após a requisição ser concluída, o método **push()** será executado para incluir o novo atendimento na lista de atendimentos.

A função **editar**, definida entre as linhas 18 e 23, é praticamente idêntica à função **salvar**, com a única diferença que o atendimento editado não será adicionado na lista de atendimentos. Por sua vez, a função **excluir**, definida entre as linhas 25 e 31, utiliza a variável **angularResourceParam**, pois se faz necessário enviar o protocolo de atendimento como parâmetro. Ao executar a função **delete()**, o *data source* automaticamente faz uma requisição HTTP com o verbo DELETE. A função **splice** irá remover o atendimento excluído da lista de elementos. Além destas funções, também é utilizada a função **novo**, definida entre as linhas 33 e 35, que apenas limpa os campos de atendimento.

Listagem 11. Código-fonte do arquivo controllers.js.

```
01.  var crudAtendimento = angular.module('crudAtendimento', ["ngResource"]);
02.
03.  crudAtendimento.controller("AtendimentoController",
     ["$resource", "$scope", function($resource, $scope) {
04.
05.    var angularResource = $resource("/SistemaAtendimento/ws/atendimento");
06.    var angularResourceParam = $resource("/SistemaAtendimento/ws/
       atendimento/:protocolo",  {protocolo : "@protocolo"});
07.
08.    $scope.atendimentos = angularResource.query(function(){});
09.
10.    $scope.salvar = function() {
11.      var newResource = new angularResource($scope.atendimento);
12.      newResource.$save(function(){
13.        $scope.atendimentos.push(newResource);
14.         $scope.novo();
15.      });
16.    }
17.
18.    $scope.editar = function() {
19.      var newResource = new angularResource($scope.atendimento);
20.      newResource.$save(function(){
21.        $scope.novo();
22.      });
23.    }
24.
25.    $scope.excluir = function() {
26.      var newResource = new angularResourceParam($scope.atendimento);
27.      newResource.$delete(function() {
28.       $scope.atendimentos.splice($scope.atendimentos.indexOf
          ($scope.atendimento), 1);
29.           $scope.novo();
30.      });
31.    }
32.
33.    $scope.novo = function () {
34.      $scope.atendimento = "";
35.    };
36.
37.    $scope.seleciona = function (atendimento) {
38.      $scope.atendimento = atendimento;
39.    };
40.  }])
```

Por fim, temos a função **seleciona**, declarada entre as linhas 37 e 39. Com esta função, quando o usuário clicar em um atendimento mostrado no data grid da tela, automaticamente serão preenchidos todos os campos do formulário com os dados do atendimento selecionado. Logo, esta funcionalidade facilita o processo de edição e remoção de um atendimento.

Implementando a camada View

A implementação desta camada consiste basicamente em construir uma interface gráfica para que o usuário do sistema possa interagir com todas as funcionalidades definidas no módulo **crudAtendimento**, explicado anteriormente. Com isso, serão desenvolvidas páginas estáticas em HTML que estendem seu comportamento utilizando diretivas do framework AngularJS. A **Listagem 12** mostra o código da página *index.html*, que está situada em *SistemaAtendimento/WebContent/*. Esta página contém a tela de atendimento com operações como salvar, excluir e pesquisar. Por questão de espaço, apenas as diretivas do AngularJS serão detalhadas. Com isso, serão omitidas explicações sobre o funcionamento das tags básicas da HTML.

Na linha 1, a diretiva **ng-app** serve para ativar o módulo **crudAtendimento** em toda a página. Na linha 12, a diretiva **ng_controller** indica que todo o conteúdo dentro da tag **div** será gerenciado pelo controller **AtendimentoController**. Com isso, os métodos que foram definidos por este controller, na **Listagem 11**, podem ser chamados usando a diretiva **ng-click**, como mostram as linhas 19, 22, 25 e 72.

Além dos métodos, as linhas 38, 44, 49, 53 e 63 mostram atributos que são definidos dentro das diretivas **ng-model**. Por exemplo, a linha 38 apresenta um atributo chamado **atendimento.protocolo**, que consiste basicamente em um objeto de nome **atendimento** com um campo de nome **protocolo**. Quando algum valor for preenchido no campo de texto da linha 38, caso não exista, automaticamente será criado um objeto chamado **atendimento** dentro de **AtendimentoController**, e o campo **protocolo** estará preenchido com o que foi digitado no campo de texto. Deste modo, as informações preenchidas nos campos de texto estarão automaticamente disponíveis para os métodos da **Listagem 11**.

A linha 72 mostra a diretiva **ng-repeat**, que é parecida com o comando **for-each**. Esta diretiva irá iterar em uma lista de atendimentos filtrados pelo valor preenchido no campo de busca, que traz a diretiva **ng-model** com o nome **criterio**. Por fim, na linha 79 temos a diretiva **ng-show**, com uma condição booleana que define que a tag HTML que apresenta esta diretiva será mostrada somente se a condição for verdadeira.

Além das diretivas AngularJS, a **Listagem 12** também possui uma referência para o arquivo *controllers.js* e os estilos definidos

Listagem 12. Página index.html utilizando diretivas do AngularJS.

```
01. <html ng-app="crudAtendimento">
02. <head>
03. <meta http-equiv="Content-Type" content="text/html;charset=UTF-8">
04. <script src="lib/angular.min.js"></script>
05. <script src="lib/angular-resource.min.js"></script>
06. <script src="js/controllers.js"></script>
07. <script src="js/bootstrap.min.js"></script>
08. <link rel="stylesheet" href="css/bootstrap.min.css" />
09. </head>
10.
11. <body>
12.   <div ng-controller="AtendimentoController" class="well"
13.     style="width: 700px; margin-top: 60px; margin-left: auto; margin-right: auto;">
14.
15.
16.     <button class="btn btn-default" ng-click="novo()">Novo</button>
17.           </td>
18.           <td class="col-xs-3">
19.             <button class="btn btn-default" ng-click="salvar()">Salvar</button>
20.           </td>
21.           <td class="col-xs-2">
22.             <button class="btn btn-default" ng-click="editar()">Editar</button>
23.           </td>
24.           <td class="col-xs-4">
25.             <button class="btn btn-default" ng-click="excluir()">Excluir</button>
26.           </td>
27.         </tr>
28.     </table>
29.
30.
31.     <hr />
32.     <form name="atendimentoForm" class="form-inline">
33.       <table>
34.
35.         <tr class="row">
36.           <td class="col-xs-2">Protocolo:</td>
37.           <td class="col-xs-3"><input type="text" style="width: 100px;"
38.             ng-model="atendimento.protocolo" /></td>
39.         </tr>
40.
41.         <tr class="row">
42.           <td class="col-xs-2">Nome Cliente:</td>
43.           <td class="col-xs-3"><input type="text" style="width: 330px;"
44.             ng-model="atendimento.nomeCliente" /></td>
45.         </tr>
46.         <tr class="row">
47.           <td class="col-xs-2">Detalhamento:</td>
48.           <td class="col-xs-3"><input type="text" style="width: 330px;"
49.             ng-model="atendimento.detalhamento" /></td>
50.         </tr>
51.         <tr class="row">
52.           <td class="col-xs-2">Nota Atendimento:</td>
53.           <td class="col-xs-3"><select name="selectedFacilityId"
            ng-model="atendimento.atendimentoEnum" />
54.             <option value="">Nota Atendimento</option>
55.             <option value="EXCELENTE">Excelente</option>
56.             <option value="OTIMO">Ótimo</option>
57.             <option value="BOM">Bom</option>
58.             <option value="RUIM">Ruim</option> </select></td>
59.         </tr>
60.       </table>
61.     </form>
62.     <hr />
63.     <input type="text" class="form-control" ng-model="criterio"
64.       placeholder="O que você está procurando?" /><br />
65.     <table class="table table-striped">
66.       <tr>
67.         <th>Nome Cliente</th>
68.         <th>Protocolo</th>
69.         <th>Detalhamento</th>
70.         <th>Nota Atendimento</th>
71.       </tr>
72.       <tr ng-repeat="atendimento in atendimentos | filter:criterio"
        ng-click="seleciona(atendimento)">
73.         <td>{{atendimento.nomeCliente}}</td>
74.         <td>{{atendimento.protocolo}}</td>
75.         <td>{{atendimento.detalhamento}}</td>
76.         <td>{{atendimento.atendimentoEnum}}</td>
77.       </tr>
78.     </table>
79.     <span ng-show="(fornecedores | filter:criterio).length == 0">Infelizmente
80.       não temos o item que você está procurando!</span>
81.   </div>
82. </body>
83. </html>
```

pelo framework Bootstrap, que são referenciados em várias tags HTML por meio do atributo **class**. Além do layout responsivo, estes estilos proporcionam um bom visual para a nossa página, como mostra a **Figura 5**.

Novo | Salvar | Editar | Excluir

Protocolo:
Nome Cliente:
Detalhamento:
Nota Atendimento: Nota Atendimento ▾

O que você está procurando?

Nome Cliente	Protocolo	Detalhamento	Nota Atendimento
Renata Vieira	74647837	Telefone sempre ocupado	RUIM
Philippe Natal	92736383	Cliente ficou satisfeito	EXCELENTE
Altino Oliveira	1223253	Cliente comprou um mix de produtos	OTIMO

Figura 5. Tela de atendimento aos clientes

Aprendemos neste artigo como frameworks JavaScript podem ser produtivos para a criação das camadas View e Control em aplicações MVC. Já para a implementação da camada Model no lado servidor, a especificação Java EE segue como uma excelente opção, fornecendo APIs que maximizam a produtividade e a qualidade de código do desenvolvedor.

Com a abordagem mostrada no artigo o desenvolvimento do software poderia ser dividido entre no mínimo duas equipes, onde uma seria especialista no lado servidor e outra no lado cliente.

Desta forma são proporcionadas diversas vantagens, como paralelismo na execução dos projetos, especialização dos profissionais, reaproveitamento de código-fonte, dentre outros. A tendência é que as camadas sejam cada vez mais desacopladas umas das outras, e a arquitetura empregada neste artigo contribui para isto.

Autor

Lucas de Oliveira Pires
pires.info@gmail.com
É Formado em Sistemas de Informação pelo Centro Universitário do Triângulo (Unitri) e cursa especialização em Análise e Desenvolvimento de Sistemas Aplicados à Gestão Empresarial (IFTM). Trabalha na empresa Algar Telecom como Analista de Desenvolvimento.

Autor

Carlos Eduardo de Carvalho Dantas
carloseduardodantas@iftm.edu.br
É Formado em Engenharia de Computação, com Especialização em Desenvolvimento Java. Possui 10 anos de experiência no desenvolvimento de soluções corporativas em diversas plataformas e atualmente é Professor no IFTM Campus Uberlândia Centro e cursa mestrado em Engenharia de Software na UFU.

Links:

Endereço para download do Java 8.
http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Endereço para download do Eclipse Luna SR1a (4.4.1).
http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/lunasr1a

Endereço para download do WildFly (8.2.0).
http://wildfly.org/downloads/

Endereço para download do AngularJS (1.3.14).
https://code.angularjs.org/1.3.14/

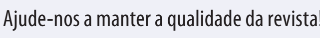
Endereço de instalação do MySQL 5.
http://dev.mysql.com/doc/refman/5.0/en/installing.html

Endereço para download do MySQL connector (5.1.18).
http://mvnrepository.com/artifact/mysql/mysql-connector-java/5.1.18

Ciclo de vida das entidades JPA.
http://www.arquiteturacomputacional.eti.br/2013/02/entenda-o-ciclo-de-vida-das-entidades.html

Você gostou deste artigo?

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



Ajude-nos a manter a qualidade da revista!

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

http://www.devmedia.com.br/

Introdução ao Spring Batch – Parte 1

Desenvolvendo soluções elegantes e robustas para processamento em lotes

ESTE ARTIGO FAZ PARTE DE UM CURSO

Fique por dentro

Ao longo deste artigo introduziremos os conceitos fundamentais de um framework muito popular empregado no desenvolvimento de aplicações de processamento em lote: o Spring Batch. Este tipo de solução é comum em ambientes nos quais são necessários a leitura, o processamento e a persistência de grandes volumes de dados.

O foco deste texto recai principalmente na apresentação dos elementos fundamentais que compõem o desenho do framework, usando como pano de fundo um projeto prático. Ao final do artigo, esperamos que o leitor tenha adquirido uma visão clara da arquitetura do Spring Batch, fundamental para que consiga explorar todos os seus recursos de maneira plena e consistente na construção de suas soluções.

Todo ambiente de negócio em que grandes volumes de dados são continuamente gerados e processados acabará se tornando um potencial cliente de sistemas de processamento em lotes. Imagine, por exemplo, grandes operadoras de telefonia que recebem, por dia, milhares de ligações de clientes para reclamar de um serviço, buscar informações gerais ou, ainda, cancelar ou atualizar seus planos. Todas as informações geradas a partir dessas experiências são muito valiosas e, quando combinadas com dados provenientes de outros sistemas e/ou serviços – como ERPs e CRMs – resultam em um material extremamente estratégico que pode ser usado – e frequentemente o é – para melhorar a experiência do público.

Neste caso que acabamos de citar, a leitura e a persistência desses dados são apenas os primeiros passos de uma série de outros que ocorrem no interior desses grandes sistemas. Todo registro de entrada, quando armazenado, representa um evento bastante pontual, uma experiência bastante particular. Quando combinados com toda uma base histórica previamente disponível, ganham uma importância muito mais evidente, pois somam-se a inúmeras ocorrências similares que, então, permitem a analistas traçar perfis de comportamento de seus clientes ao longo do tempo.

Além disso, nada impede que registros cheguem a uma plataforma por inúmeros pontos de entrada e, internamente, sejam combinados para representar uma informação de forma mais rica, mais completa. Voltando ao exemplo de atendimento ao cliente, é possível que os diversos componentes envolvidos – atendimento automático, transferência para e entre agentes, dentre outros – utilizem sistemas de informação/armazenamento separados e, para que a interação do cliente seja representada de forma plena, algumas transformações e combinações devam ser realizadas.

Outro cenário típico de processamento de dados em lotes é o de sistemas de fechamento contábil utilizado pelas empresas, na organização de sua saúde financeira. Periodicamente, a cada fim de mês, todos os registros de entrada e saída de capital são combinados e processados para que um balanço final possa ser obtido. Novamente, percebemos que há um volume importante de registros que deve ser lido e processado para que possa, enfim, ser empregado em análises e tomadas de decisão.

Torna-se simples constatar, pelos exemplos supracitados, o quão amplo é o leque de cenários em que este tipo de sistema pode ser aplicado. Logo mais, na parte prática do artigo, veremos em detalhes outro exemplo bastante corriqueiro.

Ao mesmo tempo que esta arquitetura de sistemas é comum e sua empregabilidade tende a ser alta, seus componentes essenciais variam muito pouco em comportamento. Em linhas gerais, os elementos centrais destinam-se a:

- Organizar o processamento em lotes de itens;
- Ler itens em alguma fonte de dados;
- Escrever itens em alguma fonte de dados;
- Registrar todas as operações realizadas ao longo da vida útil do sistema.

As fontes de dados mais comuns, por sua vez, são bancos de dados relacionais, arquivos de texto em formatos como XML ou CSV e filas e tópicos de sistemas de mensageria. Essas fontes servem tanto para consumo quanto para escrita de itens, mas a operação de escrita ou leitura sobre elas é algo bem conhecido da grande maioria dos desenvolvedores, e sua implementação tende a não variar em lógica, mas no conteúdo do que se lê ou escreve.

Por esta breve descrição de apenas parte do que sistemas dessa natureza fazem, já identificamos características e operações que tendem a se repetir em toda implementação. Sendo assim, escrevê-lo a cada projeto que iniciamos seria, sem dúvidas, um consumo desnecessário de tempo e recursos; traria, também, o risco que todo código-fonte novo tem, por mais bem escrito e testado que esteja: a possibilidade de falhar.

A solução natural para alcançar alta produtividade e qualidade no desenvolvimento de software, especialmente à luz do paradigma da orientação a objetos, todos nós sabemos: reuso. Quanto mais frameworks, bibliotecas e plataformas estiverem disponíveis e, principalmente, maduras, maiores serão as chances de combiná-las na construção de um produto ou serviço de alta qualidade, em um intervalo de tempo razoavelmente pequeno. Logo, devemos trabalhar constantemente nossa habilidade de não apenas identificar frameworks interessantes para os mais variados cenários de aplicação, mas também nos mantermos constantemente atentos ao código que produzimos, a fim de identificarmos qualquer possibilidade de reuso da lógica que desenvolvemos.

É neste contexto que o Spring vem, há anos, mostrando ser uma iniciativa muito bem-sucedida. Altamente modularizado, atende a praticamente todos os principais escopos técnicos de desenvolvimento de soluções de software. A Java Magazine tem sido, por anos, uma grande divulgadora deste framework, reservando em seu acervo excelentes artigos de profissionais renomados no mercado brasileiro, sobre várias das bibliotecas que compõem o que chamaremos aqui de 'Ecossistema Spring'. Da biblioteca básica de injeção de dependências e gerenciamento de contextos a outras mais complexas como Spring Data ou Spring Integration, fica sempre a impressão de um material muito bem produzido e fácil de utilizar.

Ao longo deste artigo, estudaremos as características fundamentais do Spring Batch. Esta é a oferta da Spring para a construção de sistemas de processamento em lotes. Marcado por sua estabilidade, é um framework muito bem escrito, fácil de utilizar e largamente empregado pela comunidade. Dito isso, iniciaremos com uma breve revisão sobre os principais pontos de sua arquitetura para, então, avaliar parte de seus recursos e de sua API no desenvolvimento de uma aplicação prática.

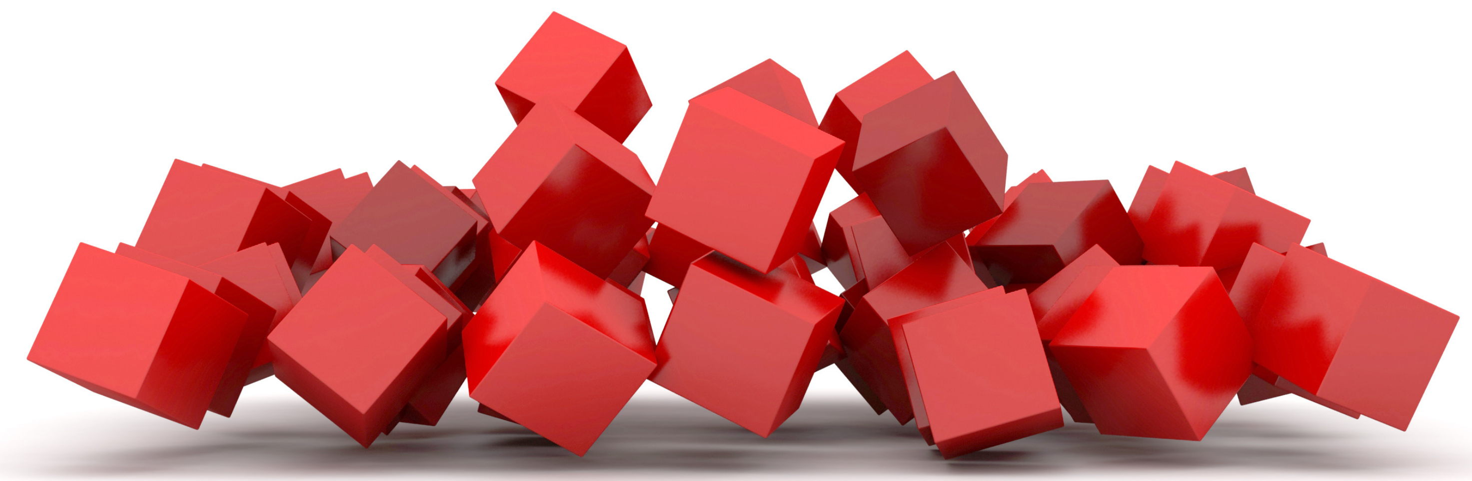
O que é o Spring Batch?

O Spring Batch oferece inúmeros recursos e funções essenciais não apenas para o processamento de grandes volumes de dados, mas, também, acompanhamento e registro de toda esta execução. A biblioteca prevê suporte para o gerenciamento de transações, análise estatística, reinicialização e cancelamento de tarefas, rastreamento e registro dos ciclos de execução de lotes, dentre outros recursos. O mais interessante em tudo isso é que grande parte do esforço que o desenvolvedor terá usando este framework estará relacionado à configuração dos componentes já existentes, acelerando significativamente e garantindo, ao mesmo tempo, alta qualidade no processo de criação de software.

É importante salientar que, embora sistemas de processamento em lote sejam executados periodicamente, esta não é uma funcionalidade oferecida nativamente pelo framework; por outro lado, é bastante simples integrá-lo com muitas das bibliotecas desenhadas para este fim. Historicamente, observa-se uma combinação muito bem-sucedida entre Spring Batch e *Quartz Scheduler*, e maiores informações sobre sua utilização podem ser encontradas adiante, na seção **Links**.

Principais componentes

Antes de começarmos a desenvolver nossa aplicação prática, é importante que nos acostumemos com o vocabulário utilizado pelo framework para representar os principais conceitos que formam o domínio do problema.



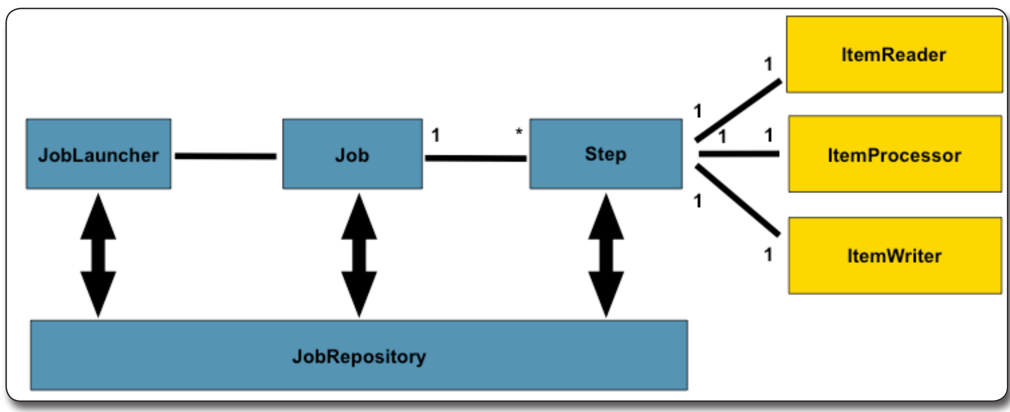


Figura 1. Composição do Spring Batch

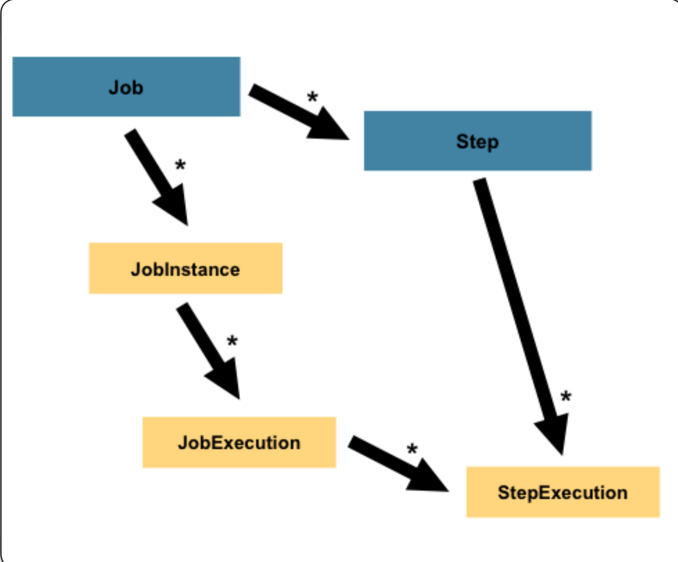


Figura 2. Análise mais granular do conceito de um job

Para facilitar este nosso estudo, observe os componentes ilustrados na **Figura 1**. É sobre esta estrutura fundamental que passaremos a conversar a partir de agora.

Job

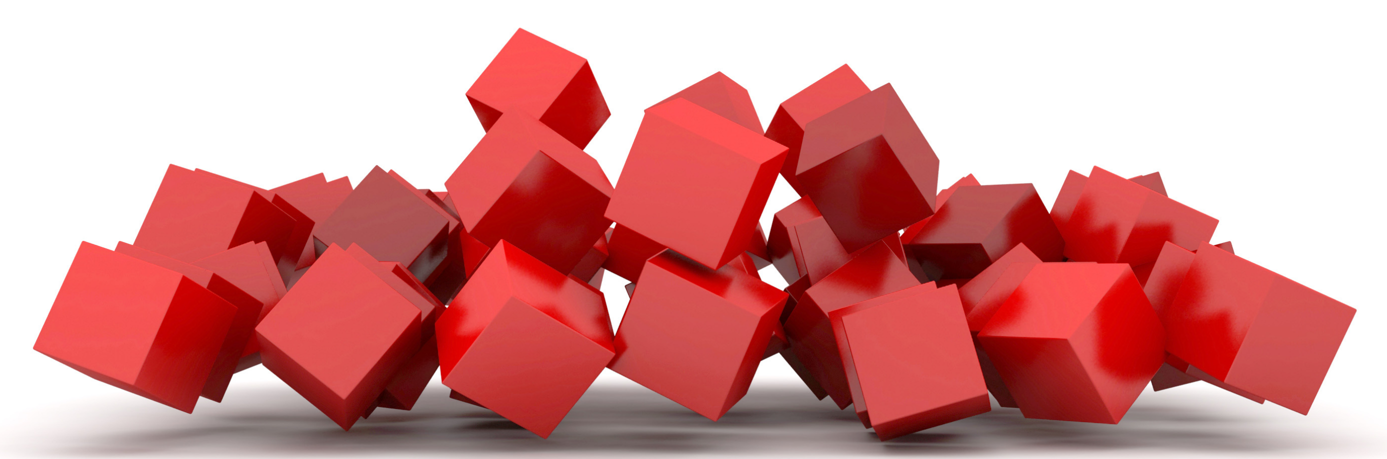
Este é o conceito mais genérico e mais importante dentro do Spring Batch. O próprio nome já sugere o seu significado dentro do framework: um trabalho! Em outras palavras, é o que será efetivamente executado pelo sistema.

Outro ponto que devemos esclarecer é que há uma separação muito bem desenhada entre o conceito de um trabalho e a sua execução. Quando configuramos um job, estamos declarando tudo aquilo que precisa ser feito quando este for colocado em execução. Esta, por sua vez, representa um evento específico em que tudo o que foi configurado e planejado é efetivamente realizado.

Esta separação de conceitos – e responsabilidades – está ilustrada na **Figura 2**. Nela, vemos que existem dois tipos de objetos que separam muito bem esses dois componentes: **Job** e **JobInstance**. Enquanto a tarefa em si é, digamos, ler um arquivo CSV, processá-lo, transformar os registros de acordo com algumas regras de negócio e, por fim, persistir toda esta informação em um banco de dados relacional, podemos querer que este mesmo procedimento seja realizado, digamos, mensalmente.

O **Job**, neste cenário, representa o escopo do trabalho a ser realizado. **JobInstance**, por sua vez, destina-se à representação de um evento particular em que o trabalho será executado. Uma combinação destes dois tipos de objetos seria, por exemplo:

- Um **Job** configurado para coletar todos os registros de movimentação financeira realizada por uma empresa ao longo de um mês, efetuando algumas agregações, cálculos de fluxo de caixa, dentre outras operações e, por fim, registrando tais resultados em alguma fonte de dados. Chamemos este **Job** de *calculo_contabil*;
- Como um ano é composto de 12 meses e o trabalho de fechamento contábil é executado mensalmente, teremos a configuração de cada um desses eventos a partir de um objeto



JobInstance diferente. Logo, para um exercício anual, teríamos 12 instâncias de **JobInstance** para o **Job** *calculo_contabil*. Todos esses objetos **JobInstance** representam o mesmo trabalho a ser feito, mas cada um mapeia as características deste trabalho para um mês do ano;
• A partir dessa combinação, o mesmo **Job** *calculo_contabil* teria, digamos, objetos **JobInstance** intitulados *calculo_contabil_**Jan***, *calculo_contabil_**Fev***, *calculo_contabil_**Abr***, *calculo_contabil_**Mai***, *calculo_contabil_**Jun***, *calculo_contabil_**Jul***, *calculo_contabil_**Ago***, *calculo_contabil_**Set***, *calculo_contabil_**Out***, *calculo_contabil_**Nov*** e *calculo_contabil_**Dez***.

JobInstance, no entanto, não representa o nível mais granular deste domínio. Imagine que, por qualquer motivo (uma queda de energia, ausência de rede, falta de memória), a execução do fechamento contábil do mês de janeiro (*calculo_contabil_**Jan***) resultasse em falha (ou fosse interrompida abruptamente). Seria necessário que, em algum momento, essa execução fosse reiniciada, o que significa que o **JobInstance** *calculo_contabil_**Jan*** teria que ser executado, no mínimo, mais uma vez.

Para representar as várias execuções de um mesmo **JobInstance**, o Spring Batch oferece um tipo de objeto chamado **JobExecution**. Assim, podemos mapear não apenas os eventos particulares de um Job a partir de objetos JobInstance, mas cada execução de cada evento, ao longo da vida útil de nossa aplicação. Voltando ao exemplo que iniciamos anteriormente, a combinação desses três tipos de objetos (Job, JobInstance e JobExecution) ficaria:
• O mesmo Job mencionado acima, *calculo_contabil*;
• As mesmas 12 instâncias de JobInstance, para o Job *calculo_contabil*, sendo cada instância dedicada à representação do trabalho de um mês do ano;
• Duas instâncias de JobExecution para o JobInstance *calculo_contabil_**Jan***:
  - *calculo_contabil_**Jan_01***, referente à primeira execução que falhou;
  - *calculo_contabil_**Jan_02***, referente à nova execução iniciada quando, por exemplo, a energia do prédio foi reestabelecida e o servidor voltou a operar.

Isto é tudo o que precisamos saber sobre Jobs, neste momento. Agora que já entendemos como um trabalho é representado dentro do contexto do Spring Batch, passaremos a avaliar a sua organização interna.

Steps

Todo trabalho é organizado em um conjunto de uma ou mais fases. É assim que o framework estrutura um Job: um conjunto de instâncias de um tipo de objeto denominado **Step**. Da mesma forma que um Job, Step é a representação conceitual de uma fase, não a execução propriamente dita.

Como vimos para o Job *calculo_contabil*, anteriormente, é possível que uma fase específica de sua execução falhe, por qualquer motivo que seja. Um objeto Step representa o que precisa ser realizado naquela fase específica, mas não é quem representará a sua execução. Para isso, o Spring Batch oferece um tipo de dados específico, denominado **StepExecution**. Voltando ao cenário que descrevemos quando conversamos sobre Jobs, podemos pensar da seguinte maneira:
• Há um **Job** que identificamos como *calculo_contabil*;
• Sua estrutura envolve apenas um **Step**, *processar_movimentacao*;
• Além disso, *calculo_contabil* contém 12 instâncias de **JobInstance**, cada uma dedicada à representação do trabalho de um mês do ano (*calculo_contabil_**Jan***, *calculo_contabil_**Fev***, *calculo_contabil_**Abr*** e assim por diante);
• O **JobIntance** de Janeiro (*calculo_contabil_**Jan***) teve de ser executado duas vezes pelo fato de ter falhado na primeira:
  - *calculo_contabil_**Jan_01***, referente à primeira execução que falhou;
  - *calculo_contabil_**Jan_02***, quando enfim foi executado com sucesso.
• O **JobExecution** *calculo_contabil_**Jan_01*** possui um **StepExecution** referente ao Step *processar_movimentacao*. Podemos representá-lo como:
  - *calculo_contabil_**Jan_01**_ processar_movimentacao_**01***;
• O **JobExecution** *calculo_contabil_**Jan_02*** possui um **StepExecution** referente ao Step *processar_movimentacao*. Podemos representá-lo como:
  - *calculo_contabil_**Jan_02**_ processar_movimentacao_**01***.

Perceba, por esta representação, que um mesmo *Step* pode ser executado mais de uma vez dentro de um *JobExecution*. Isso permitiria, por exemplo, termos uma situação como a mapeada a seguir:
• **JobExecution** *calculo_contabil_**Jan_01***:
  - *calculo_contabil_**Jan_01**_ processar_movimentacao_**01***;
  - *calculo_contabil_**Jan_01**_ processar_movimentacao_**02***;
  - *calculo_contabil_**Jan_01**_ processar_movimentacao_**03***.

A grande importância existente no mapeamento de jobs e steps, da forma como descrevemos, não está relacionada ao propósito do software em questão, mas na representação e no registro de seu comportamento ao longo do tempo. Para qualquer empresa, em qualquer ramo de negócio, é vital que haja meios de se investigar o histórico de execução de sistemas, para a rápida identificação e o tratamento de problemas e/ou gargalos, assim que surgirem.

ItemReader

Todo passo conterá, conforme veremos mais adiante, um componente responsável pela extração de dados. Sua função será identificar a origem dos dados a serem tratados, extraí-los e mapeá-los na forma de objetos que serão, posteriormente, processados pelos componentes seguintes desta cadeia.

Há, na API do Spring Batch, alguns componentes que já apresentam comportamento padrão para os tipos mais comuns de fontes de dados. Todos esses componentes derivam da interface

org.springframework.batch.item.ItemReader, e algumas das mais populares são:

- **org.springframework.batch.item.file.FlatFileItemReader:** usado para consumir registros gravados em arquivos de texto (o formato mais comum empregado na representação de dados em arquivos deste tipo é o CSV);
- **org.springframework.batch.item.jms.JmsItemReader:** utilizado para leitura de dados a partir de sistemas de mensageria (JMS);
- **org.springframework.batch.item.database.JdbcCursorItemReader:** utilizado para leitura de dados a partir de bancos de dados.

A interface padrão para a implementação de *item readers*, a já mencionada **org.springframework.batch.item.ItemReader**, declara apenas um método. Vamos, brevemente, analisar sua assinatura:

```
T read() throws Exception, UnexpectedInputException, ParseException, NonTransientResourceException
```

Por ela, vemos que há um método de nome **read()**, que deve ser implementado por todo item reader e que tem por função retornar um objeto de um tipo específico. Este tipo **T** é, portanto, um parâmetro que será vinculado a um tipo de objetos que definirmos na implementação ou na configuração do item reader que inserimos em nossa aplicação. Este objeto, quando retornado, cai novamente no fluxo do Job e, então, é encaminhado para o próximo componente na cadeia. Normalmente, este componente é um item processor, que veremos a seguir.

ItemProcessor

Item processors são componentes empregados por um Step em atividades como transformação, validações e verificações. Compõem o principal momento dentro do fluxo de um step, pois é aqui que todas as regras de negócio, validações, verificações e outros tratamentos devem ser realizados para que a informação coletada da fonte de entrada de dados esteja, enfim, absolutamente condizente com os objetos da aplicação. Logo, os *item processors* são os agentes que efetivamente transformarão a informação original e darão, a ela, uma roupagem mais completa, mais rica, adquirida a partir de todo um conjunto de regras e características ditadas pela aplicação.

Da mesma forma que *item readers*, todo *item processor* deriva de uma mesma interface que declara apenas um método. Vejamos a assinatura deste método:

```
O process (I var) throws Exception
```

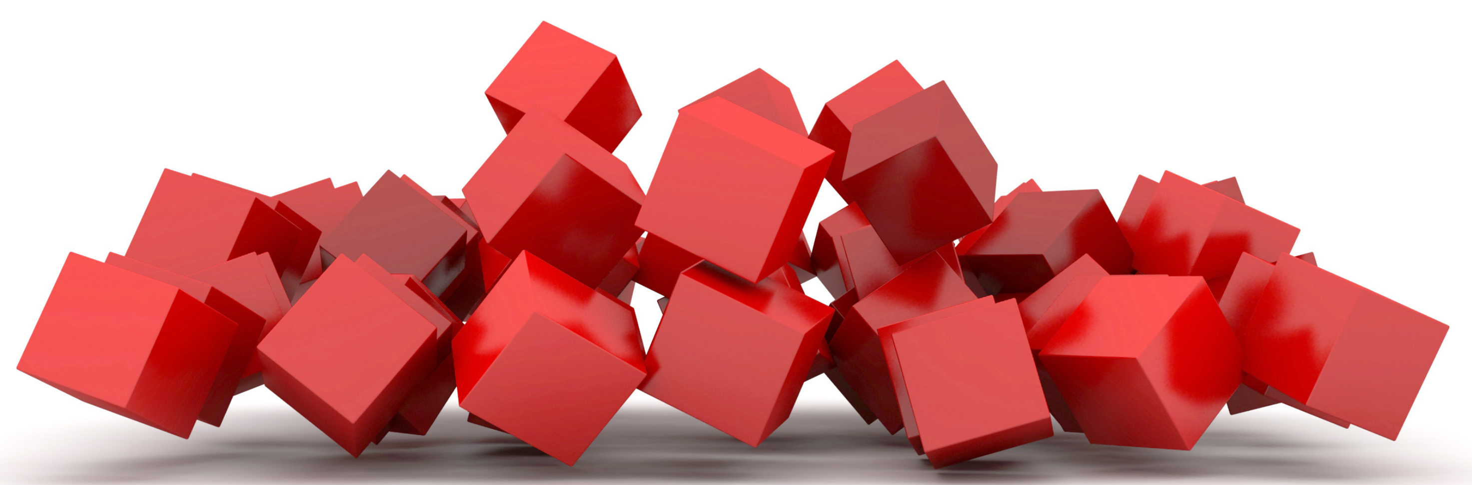
O parâmetro de entrada, representado pelo tipo **I**, corresponde ao objeto lido pelo **ItemReader**. Quando **ItemReader** extrai um registro da fonte de entrada de dados, mapeia todos os seus campos na forma de um objeto que, na sequência, é informado ao *Step*. Este, por sua vez, o envia para o processador, que trabalhará em cima da informação de modo a torná-la completamente aderente ao modelo de negócios aplicado ao sistema.

Quando o processamento é finalizado, o objeto resultante é informado ao step, que se encarrega de enviá-lo para o próximo – e último – componente do fluxo: o escritor de itens. O Spring Batch oferece, ainda, a possibilidade de se encadear múltiplos processadores de modo a tornar cada um dos processadores o mais especialista possível no tratamento do objeto de entrada. Embora não seja empregado no exemplo prático deste artigo, este recurso pode ser bem interessante para o leitor, e por isso disponibilizamos o link para este assunto, direto da documentação oficial do Spring Batch, na seção **Links**.

ItemWriter

Item writers são os últimos componentes no fluxo de execução de um step. É aqui que, finalmente, os objetos já transformados pelos *item processors* serão disponibilizados em algum canal de comunicação para consumo (que pode ser uma base de dados relacional, um sistema de mensageria, arquivos de texto ou outro).

Assim como para *item readers*, há alguns tipos de objetos com lógica pré-definida para a escrita de itens. Vejamos alguns exemplos:



- **org.springframework.batch.item.file.FlatFileItemWriter:** utilizado para a escrita de dados em arquivos comuns, de texto;
- **org.springframework.batch.item.database.JdbcBatchItemWriter:** utilizado para a escrita de registros em um banco de dados relacional;
- **org.springframework.batch.item.database.HibernateItemWriter:** utilizado para a persistência de objetos usando o Hibernate;
- **org.springframework.batch.item.jms.JmsItemWriter:** utilizado para a escrita de objetos em tópicos ou filas de sistemas de mensageria (RabbitMQ, ActiveMQ ou outro).

JobRepository

Voltemos um pouco à **Figura 1**. Nela, encontramos um componente chamado **JobRepository**, sobre o qual passaremos a conversar brevemente a partir de agora.

Este tipo de objeto está relacionado a um mecanismo do Spring Batch para realizar a persistência de todos os registros relacionados à configuração e execução de jobs. Como já vimos, esta possibilidade de identificação do comportamento de nossos sistemas de processamento em lote ao longo do tempo é vital, pois nos ajuda a melhorar tanto a infraestrutura na qual executam como, também, corrigir ou antecipar potenciais gargalos ou falhas.

É a partir de tabelas gerenciadas pelo **JobRepository**, por exemplo, que o framework identifica e guarda informações relacionadas a JobExecution, StepExecution, dentre outros. Veremos, na parte prática, como essas tabelas são criadas e, ao longo de algumas listagens de código, como esta informação é gravada e gerenciada.

Chegamos, enfim, ao ponto do artigo em que todos os conceitos fundamentais do framework já foram tratados. Com tudo o que vimos até aqui, já temos condição de começar a estudar um exemplo prático, entendendo como cada elemento se encaixa na configuração de um sistema real de processamento em lotes.

Exemplo prático

O exemplo que construiremos juntos, a partir deste tópico, é uma aplicação prática de um tipo de componente muito útil em ambientes corporativos, conhecido pelo acrônimo de ETL (do inglês ***E****xtract*-***T****ransform*-***L****oad*).

Seu emprego ocorre principalmente na integração de sistemas que não compartilham dos mesmos formatos, padrões, protocolos e até dimensões para a representação e comunicação de dados.

Nota

ETL é o acrônimo para Extract Transform Load. Este padrão de arquitetura é útil para o desenvolvimento de sistemas que precisam extrair dados de, ao menos, uma fonte de origem, processá-los de acordo com regras específicas de negócio e, enfim, disponibilizados em algum canal (como tópicos/filas MQ, bancos de dados relacionais, arquivos de texto, dentre outros).

Além disso, normalmente trabalham com volumes grandes e crescentes de dados.

ETLs são, também, frequentemente associados a frameworks de agendamento de tarefas, para que possam ter sua execução agendada e automaticamente iniciada de tempos em tempos, sem que seja necessária qualquer interação humana no processo.

Um exemplo típico de ETL, descrito no início do artigo, é o de sistemas de fechamento contábil. Outro ambiente no qual componentes ETL são muito importantes, é o de Contact Centers. Raramente homogêneos em sua composição, podem apresentar produtos e serviços (de hardware a software) de fabricantes diversos, que precisam interagir harmonicamente para que a solução como um todo opere de acordo com o desejado. Qualquer informação produzida em plataformas como as que rodam em contact centers são altamente relevantes para a avaliação da qualidade final do serviço, e o simples fato de que componentes podem não se comunicar bem entre si não pode, de forma alguma, implicar na perda de informação ao longo dos fluxos operacionais.

Neste tópico, desenvolveremos um ETL suficientemente simples para mantermos o foco no que realmente interessa: o estudo do Spring Batch. Antes, porém, pedimos ao leitor para fazer uma pausa para avaliar o conteúdo da **Tabela 1**, na qual o significado e a função de cada uma das letras que compõem o acrônimo ETL foram brevemente descritas.

A estrutura do projeto

O projeto prático deste artigo foi desenvolvido utilizando o Maven. Esta decisão foi tomada porque a adoção desta ferramenta não facilita apenas o gerenciamento de dependências, mas todo o ciclo de vida do projeto, desde a compilação até sua distribuição.

O código-fonte está organizado a partir de módulos, conforme ilustrado na **Figura 3**. Por ela, verificamos a seguinte composição:

- **springbatch-etl**: este é o projeto propriamente dito, mantendo todos os outros módulos listados a seguir como componentes (submódulos);

Camada	Descrição
Extract	Extratores/leitores de dados. Responsáveis por extrair dados de uma fonte de origem, para que sejam então processados e transformados de acordo com a necessidade
Transform	Transformadores/processadores de dados. Recebem os dados colhidos pelos extratores e, de acordo com as especificações definidas para o ETL, transformam estes dados de modo a atender os sistemas que deles farão uso.
Load	Carregadores/escritores de dados. Última camada a ser acionada no fluxo natural de um ETL, são responsáveis por persistir os dados já transformados ou, dependendo da natureza dos clientes desses dados, disponibilizá-los em data grids ou filas de sistemas de mensageria (como RabbitMQ ou similares).

Tabela 1. Descrição dos componentes de um ETL

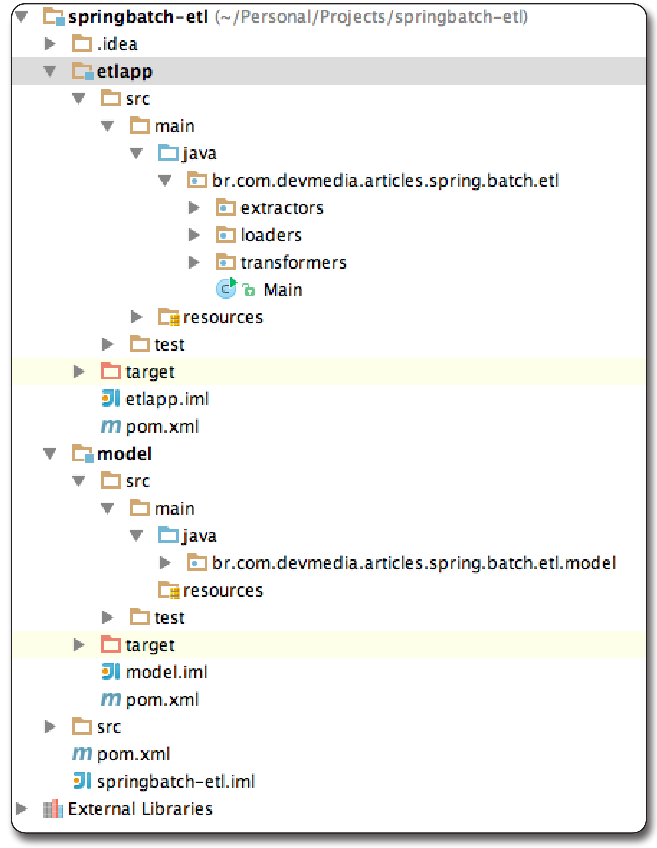


Figura 3. Estrutura do projeto

• **etlapp**: módulo no qual todo o comportamento de processamento em lote é encontrado;
• **model**: contém a representação do domínio do problema, mapeando os principais tipos de objetos relacionados ao negócio.

Utilizamos, ao longo do desenvolvimento, a IDE IntelliJ IDEA (veja a seção **Links** para mais informações). Entretanto, nada impede que o leitor empregue outras ferramentas mais populares, como Eclipse ou NetBeans. Todas essas ferramentas também oferecem, nativamente, excelente suporte para o desenvolvimento de projetos Maven. Portanto, o único ponto de atenção é que, independentemente da IDE que adote para estudar o código-fonte desta aplicação, certifique-se de importar o projeto como um Projeto Maven. A partir daí todos os ajustes e operações necessárias para abri-lo corretamente ficarão sob a responsabilidade da IDE.

Começaremos a análise do projeto por sua configuração. Cada módulo contém um arquivo no qual todas as suas dependências e características gerais são mapeadas. Este arquivo vem da estrutura do Maven e é conhecido como 'POM' (*Project Object Model*). A razão de termos um arquivo POM para cada módulo é, simplesmente, para hierarquizar a estrutura de dependências, propriedades, plug-ins e demais características, de modo que cada módulo declare apenas e tão somente o que é específico de sua composição.

A **Listagem 1** exibe o conteúdo do arquivo descritor do módulo *etlapp*, dedicado à configuração do módulo de ETL.

Tudo o que precisamos fazer para começar a usar o Spring Batch em um projeto é declarar sua dependência em relação à biblioteca *spring-batch-core*. Neste exemplo, utilizamos uma versão superior à ultima release oficial, para manter o artigo o mais atualizado possível em relação às distribuições do framework. Possivelmente, quando o leitor experimentar o código, esta versão atualmente disponível como SNAPSHOT já terá se tornado uma RELEASE oficial do framework.

Listagem 1. pom.xml – Arquivo de configuração do módulo ETL do projeto.

```
<?xml version="1.0" encoding="UTF-8"?>
<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">
  <parent>
    <artifactId>etl</artifactId>
    <groupId>br.com.devmedia.articles.spring.batch</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <properties>
    <spring.batch.version>3.1.0.BUILD-SNAPSHOT</spring.batch.version>
  </properties>

  <artifactId>etlapp</artifactId>

  <dependencies>
    <dependency>
      <groupId>br.com.devmedia.articles.spring.batch</groupId>
      <artifactId>model</artifactId>
      <version>${project.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework.batch</groupId>
      <artifactId>spring-batch-core</artifactId>
      <version>${spring.batch.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>4.1.4.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.34</version>
    </dependency>
  </dependencies>

  <repositories>
    <repository>
      <id>spring-snapshots</id>
      <name>Spring Snapshots</name>
      <url>http://repo.spring.io/snapshot</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </repository>
  </repositories>
</project>
```

Utilize a seção **Links**, ao final do artigo, para encontrar a referência para a página oficial do Spring Batch, onde são informadas as últimas versões lançadas, além de tutoriais rápidos de uso e outras informações gerais.

Outras duas dependências muito importantes são *spring-core* e *spring-context*. Entretanto, essas são dependências que não apenas o módulo *etlapp* terá, mas possivelmente todos os demais que possam vir a compor o projeto. Portanto, essas bibliotecas foram declaradas no descritor global do projeto (*springbatch-etl*), como podemos ver a partir da **Listagem 2**.

Este arquivo começa declarando todos os módulos que, juntos, compõem o projeto. Em seguida, podemos ver exatamente as dependências gerais, que apontam não apenas para as bibliotecas centrais do Spring (*spring-core* e *spring-context*), mas também para outras como o conector MySQL (usado para a persistência de registros pelo nosso *item writer*).

A estrutura do job

Passaremos, neste tópico, a estudar o único job que criamos em todo o projeto. Para compreender suas principais características, analisaremos o componente da **Listagem 3**. Este é o marcador, dentro do espaço de nomes do Spring Batch, utilizado para configurar um objeto Job. Neste caso, apenas a propriedade **restartable** foi explicitamente definida. Há, no entanto, inúmeras outras que também são configuráveis, e todas elas estão muito bem explicadas na documentação oficial do framework (vide seção **Links**).

Como já sabemos, todo Job pode ter uma ou mais fases. Além disso, já aprendemos que fases são representadas, no Spring Batch, por um componente chamado Step. Para declarar este tipo de objeto em arquivos XML como o exibido na **Listagem 3**, precisamos fazer uso de um marcador denominado **batch:step**.

Na fase que criamos, identificada como **singlestep**, declaramos um objeto do tipo **org.springframework.batch.core.step.Tasklet**. Este é um componente que, de acordo com a documentação do Spring Batch, representa 'uma estratégia de um job para processar uma tarefa', e pode ser composta – como fazemos em nosso exemplo – por um objeto do tipo **org.springframework.batch.core.step.item.Chunk**. Este, por sua vez, é responsável por agrupar e coordenar uma lista com os seguintes tipos de atividades:

- Leitura do item (*item reader*);
- Processador do item (*item processor*);
- Escrita do item (*item writer*).

Todos esses tipos de componentes já foram descritos anteriormente, e o código-fonte de cada um será apresentado em detalhes, logo mais.

Listagem 2. pom.xml – Configuração principal do projeto.

```
<?xml version="1.0" encoding="UTF-8"?>
<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>br.com.devmedia.articles.spring.batch</groupId>
  <artifactId>etl</artifactId>
  <version>1.0-SNAPSHOT</version>

  <modules>
    <module>model</module>
    <module>etlapp</module>
  </modules>

  <packaging>pom</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <spring.version>4.1.4.RELEASE</spring.version>
    <log4j.version>2.1</log4j.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>${spring.version}</version>
    </dependency>

    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.34</version>
    </dependency>

    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-dbcp2</artifactId>
      <version>2.0.1</version>
    </dependency>

    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.12</version>
    </dependency>

    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.0</version>
    </dependency>
  </dependencies>
</project>
```

Chunks também podem conter listeners atuando em cada uma das etapas de sua execução. Em nosso exemplo não incluímos nenhum, mas recomendamos ao leitor que experimente este recurso quando já estiver familiarizado com o código-fonte. Listeners podem ser inseridos na declaração de um chunk a partir do marcador **<batch:listeners>**. Dentro deste nó é que são definidos todos os listeners, o que é feito a partir do marcador **<batch:listener>**.

A última característica que precisamos entender muito bem sobre chunks, antes de darmos continuidade à análise do ETL, é o seu atributo **commit-interval**. Este intervalo é definido em número de itens/registros que são processados pelo job. O chunk usará este valor como parâmetro para agrupar itens antes de enviá-los para o próximo elemento. Somente quando o intervalo for atingido é que, enfim, todos eles serão despachados para o objeto responsável pela próxima atividade. Para entendermos este mecanismo, vamos observar novamente a configuração de nosso job na **Listagem 3**.

Nela, vemos que o atributo **commit-interval** está valorado em 1. Também vemos que, dentro da execução deste chunk, são realizadas as três atividades típicas de um Job, descritas no início do artigo: leitura, processamento e escrita de itens, representadas, respectivamente, pelos atributos **reader**, **processor** e **writer**. A primeira atividade realizada pelo chunk será a leitura de itens, cujos detalhes exploraremos logo mais.

O intervalo de commit em 1 significa que cada item lido pelo item reader será imediatamente despachado para a próxima atividade. Caso tivéssemos definido o valor 2 para o intervalo, por exemplo, este mesmo despacho seria feito a cada dois itens lidos.

O registro de execuções de um batch

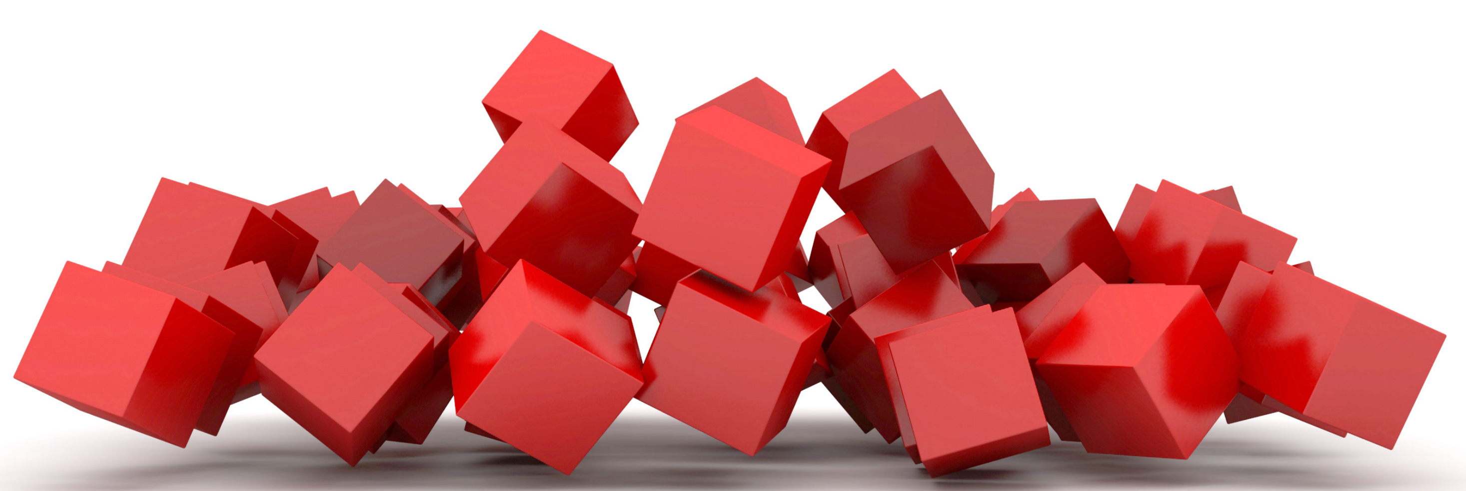
Quando descrevemos os conceitos principais que compõem o domínio do Spring Batch, dedicamos um tópico para falar sobre JobRepository, JobInstance, JobExecution, Step e StepExecution. Na ocasião, comentamos que elementos como JobInstance, JobExecution e StepExecution são metadados muito importantes de todo batch, pois ajudam a registrar todo o histórico de execução de um sistema de processamento em lotes. Através deles, podemos descrever com precisão quando e como um job foi invocado e, eventualmente, usar essas informações para reiniciar uma execução específica.

O que faremos, neste instante, é unir todo esse conhecimento que já adquirimos ao contexto da nossa aplicação. Para isso, observemos mais uma vez a **Listagem 3**. O foco, agora, estará no bean identificado como **jobRepository**.

O bean que usamos para obter objetos JobRepository em nosso projeto é do tipo **org.springframework.batch.core.repository.support.JobRepositoryFactoryBean**. Esta é, como o seu nome sugere, uma fábrica objetos, e tem por principal função abstrair o processo de criação e configuração dos objetos JobRepository. O tipo de objeto que ela constrói, por sua vez, usa um conjunto de objetos DAO internamente, que se comunicam com um banco de dados via JDBC. Este é o motivo pelo qual o leitor vê, em sua composição, objetos de data source, gerenciamento de transações e, por fim, o tipo de banco de dados. Este banco de dados com o qual esta fábrica de JobRepository é associada é utilizado para que se registre toda a execução do sistema.

O job repository criado por este bean identificado por **jobRepository** é, por sua vez, associado mais adiante a um job launcher (**SimpleJobLauncher**), que é quem efetivamente executa o job. Estas características de persistência são importantes porque definem em que tabelas serão registradas todas as execuções do job, dando-nos a condição de, futuramente, reiniciá-los e/ou avaliá-los.

Toda esta configuração, no entanto, baseia-se na premissa de que as tabelas nas quais toda a informação será gravada já estarão disponíveis para o Spring Batch. Caso não estejam, a execução da aplicação falhará. Neste ponto, temos duas possibilidades:



1. Automatizar o processo de criação das tabelas;
2. Criarmos essas tabelas do Spring Batch manualmente.

Ainda na **Listagem 3**, o leitor pode ver que optamos pela primeira opção. Assim, configuramos nossa aplicação para, automaticamente, criar todas as tabelas necessárias para a persistência de todas as informações referentes à execução do job. Fazemos isso a partir da configuração do nó **<jdbc:initialize-database>**. Nele, apontamos para os scripts fornecidos pelo próprio Spring Batch, responsáveis por remover todas as tabelas referentes a metadados de batch do banco de dados (caso tais tabelas já existam) e, na sequência, recriá-las.

Ainda sobre a inicialização do banco de dados, o leitor perceberá que o nó **jdbc:initialize-database** pede uma referência ao data source no qual deverá se conectar. No entanto, se procurarmos ao longo de toda a **Listagem 3**, não veremos nenhum objeto identificado como data source, que é o valor que associamos a este atributo. Isto acontece porque, logo no início do descritor do ETL, carregamos o conteúdo de outro descritor, especificamente destinado à configuração dos objetos de persistência (data source,

Listagem 3. 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,
              numberOfPages, 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>

  <!-- 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.JobRepository
     FactoryBean">
    <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>
```

DAO e serviços). Este arquivo é o *etl_persistence-context.xml*, exibido na **Listagem 4**.

Quando olhamos para o conteúdo deste descritor, veremos a configuração do data source a partir de um objeto do tipo **org.apache.commons.dbcp2.BasicDataSource**. Nele, adicionamos todas as propriedades de comunicação com o banco de dados, como a sua localização e as credenciais de acesso. Perceba que este mesmo objeto data source será usado não apenas pelo Spring Batch na inicialização das tabelas relacionadas aos metadados do batch, mas também pelos nossos objetos DAO para gravar os registros de livros, autores e editoras. Neste último caso, perceba que o acesso ao banco de dados será feito por um objeto do tipo **org.springframework.jdbc.core.JdbcTemplate**, também declarado no código da **Listagem 4**, e que acaba empregando o mesmo objeto data source em sua configuração.

Ao executarmos nossa aplicação com todas essas configurações já devidamente realizadas, a estrutura que obteremos no banco de dados será exatamente igual à ilustrada na **Figura 4**. Entretanto, é importante diferenciarmos, aqui, as tabelas que foram efetivamente criadas de forma automática daquelas que criamos manualmente. Os scripts SQL que mencionamos no parágrafo anterior cobrem apenas as tabelas relacionadas aos metadados do Spring Batch e, portanto, não têm relação alguma com as tabelas *author*, *book* e *pbhouse*. Caso o leitor deseje automatizar também a criação das tabelas referentes ao modelo de dados da aplicação, basta que gere o script correspondente e o inclua na configuração do nó **jdbc:initialize-database**. O código-fonte do projeto, que pode

localhost ▸ devmedia_etl

Structure | SQL | Search | Query | Export | Import | Operations | Privileges | Drop

Table	Action	Records	Type	Collation	Size	Overhead
author		0	InnoDB	utf8_bin	16.0 KiB	-
BATCH_JOB_EXECUTION		1	InnoDB	utf8_bin	32.0 KiB	-
BATCH_JOB_EXECUTION_CONTEXT		1	InnoDB	utf8_bin	16.0 KiB	-
BATCH_JOB_EXECUTION_PARAMS		0	InnoDB	utf8_bin	32.0 KiB	-
BATCH_JOB_EXECUTION_SEQ		1	InnoDB	utf8_bin	16.0 KiB	-
BATCH_JOB_INSTANCE		1	InnoDB	utf8_bin	32.0 KiB	-
BATCH_JOB_SEQ		1	InnoDB	utf8_bin	16.0 KiB	-
BATCH_STEP_EXECUTION		1	InnoDB	utf8_bin	32.0 KiB	-
BATCH_STEP_EXECUTION_CONTEXT		1	InnoDB	utf8_bin	16.0 KiB	-
BATCH_STEP_EXECUTION_SEQ		1	InnoDB	utf8_bin	16.0 KiB	-
book		0	InnoDB	utf8_bin	48.0 KiB	-
pbhouse		0	InnoDB	utf8_bin	16.0 KiB	-
12 table(s)	Sum	8	InnoDB	utf8_bin	288.0 KiB	0 B

Check All / Uncheck All With selected:

Figura 4. Banco de dados do ETL, com as tabelas criadas automaticamente pelo Spring Batch

Listagem 4. etl_persistence-context.xml – Configuração da persistência para o módulo de processamento em lotes.

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

    <context:property-placeholder location=”classpath:db.properties,
      classpath:csv.properties” />

    <!-- Configuração do acesso ao banco de dados a partir de data source -->
    <bean id=”datasource” class=”org.apache.commons.dbcp2.BasicDataSource”>
        <property name=”driverClassName” value=”com.mysql.jdbc.Driver”>
        </property>
        <property name=”url” value=”${db.url}”></property>
        <property name=”username” value=”${db.usr}”></property>
        <property name=”password” value=”${db.pwd}”></property>
    </bean>

    <bean id=”transactionManager”
        class=”org.springframework.batch.support.transaction.Resourceless
        TransactionManager” />

    <bean id=”jdbcTemplate” class=”org.springframework.jdbc.core.JdbcTemplate” >
        <property name=”dataSource” ref=”datasource” />
    </bean>

    <!-- Configuração de todos os DAOs usados no módulo do ETL -->
    <bean id=”bookDao” scope=”prototype”
      class=”br.com.devmedia.articles.spring.batch.etl.loaders.databases.BookDao” >
        <property name=”jdbcTemplate” ref=”jdbcTemplate” />
        <property name=”sqlBookSave” value=”${sql.book.save}” />
    </bean>

    <bean id=”publishingHouseDao” scope=”prototype”
        class=”br.com.devmedia.articles.spring.batch.etl.loaders.databases.
        PublishingHouseDao” >
        <property name=”sqlPubHouseListAll” value=”${sql.pbhouse.select.all}” />
        <property name=”sqlPubHouseSave” value=”${sql.pbhouse.save}” />
    </bean>
    <bean id=”authorDao” scope=”prototype”
          class=”br.com.devmedia.articles.spring.batch.etl.loaders.databases.AuthorDao” >
        <property name=”sqlAuthorListAll” value=”${sql.author.select.all}” />
        <property name=”sqlAuthorSave” value=”${sql.author.save}” />
    </bean>

    <!-- Configuração dos serviços de persistência usados no módulo do ETL -->
    <bean id=”bookService” scope=”prototype”
      class=”br.com.devmedia.articles.spring.batch.etl.loaders.services.BookService” >
      <property name=”bookDao” ref=”bookDao” />
      <property name=”authorDao” ref=”authorDao” />
      <property name=”publishingHouseDao” ref=”publishingHouseDao” />
    </bean>
</beans>
```

ser obtido a partir da seção **Links**, conterá este script para que este exercício possa, também, ser realizado.

Isto é tudo o que precisamos saber sobre o comportamento e a configuração de jobs para entender como o nosso ETL funcionará, na prática. Até aqui, entendemos como todo o trabalho está organizado e aprendemos, também, as características e os papéis básicos desempenhados por cada um dos elementos que o compõe.

Para prosseguir no estudo de nossa aplicação, dedicaremos algum tempo para entender o escopo do problema que estamos solucionando, o tipo de informação que estaremos manipulando e, também, todos os formatos em que esses dados podem ser encontrados ao longo de todo o fluxo operacional.

O Spring Batch é um poderoso framework que podemos empregar no desenvolvimento rápido e eficiente de soluções de processamento em lotes. Por meio de uma arquitetura bastante clara e organizada, somos facilmente guiados na identificação da composição de trabalhos e, também, nos meios que são disponibilizados para customizar o comportamento de cada uma das fases dos jobs que precisamos desenvolver.

A construção de sistemas desta natureza, sem a ajuda de um framework como este que exploramos ao longo do artigo, seria bastante custosa e sujeita a erros. Quando fazemos um uso de uma biblioteca já tão madura como o Spring Batch, garantimos que todo o domínio de dados e a infraestrutura lógica necessária são estáveis e robustas, permitindo que concentremos nossa atenção apenas na lógica de negócio que precisamos atender.

Como vimos no início do texto, processamento em lotes é algo bastante corriqueiro no mercado, principalmente em contextos nos quais grandes volumes de dados são constantemente produzidos e requerem, para serem úteis para usuários finais – ou mesmo outros sistemas dentro de uma infraestrutura, algum tratamento (como conversões de dados, agregações, dentre outros).

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, JasperReports e PrimeFaces – Parte 2

Saiba como melhorar o design da sua aplicação com os padrões MVC e DAO

ESTE ARTIGO FAZ PARTE DE UM CURSO

Fique por dentro

Apresentaremos neste artigo como construir um sistema com alta qualidade, produtividade e escalabilidade utilizando como ponto de referência o padrão arquitetural MVC. Além disso, buscaremos também uma melhor extensibilidade para o sistema, o que pode facilitar tarefas de manutenção e adição de novos recursos, o que será auxiliado pela adoção do DAO. Com estas boas práticas o leitor terá ciência de novas opções que podem ser adotadas em praticamente qualquer aplicação web.

Com a chegada da era da informação a velocidade com que o mercado muda o rumo dos negócios cresce rapidamente, e para atender essa demanda, o comportamento empresarial também teve que se adaptar. Diante disso, as empresas passaram a investir cada vez mais em tecnologias que potencializassem os seus negócios e os desenvolvedores, como consequência, começaram a adotar novos frameworks que possibilitassem a construção de sistemas web capazes de lidar com conteúdo mais dinâmicos, oferecer interfaces mais atraentes, maneiras mais amigáveis de navegar por suas páginas e que fossem capazes de se adaptar mais facilmente às mudanças.

Neste momento, os sistemas também passaram a lidar com quantidades cada vez maiores de informações, de forma que em situações extremas, tornou-se um importante requisito a gerência do uso da memória. Pensando nisso, apresentamos na primeira parte deste artigo as técnicas de virtualização e paginação para geração de relatórios em sistemas Java, o que foi feito com o uso do JasperReports e do Hibernate.

Neste artigo, evoluiremos a estrutura do simulador quiz já desenvolvida, de modo que ao término da implementação ela se torne uma arquitetura de alta escalabilidade. Para isso, utilizaremos o padrão arquitetural MVC, onde organizaremos os principais componentes da aplicação em camadas (Modelo Visão Controlador). Veremos também como implementar o padrão de projetos DAO, que isolará toda a camada de dados da aplicação, viabilizando assim uma melhora da extensibilidade do sistema. Isso significa que o software terá uma maior capacidade de incorporar mudanças sem comprometer drasticamente outros elementos da arquitetura. Finalmente, para que não precisemos administrar as transações em diversos pontos do sistema, centralizaremos o seu gerenciamento em um único lugar e deixaremos que o Tomcat se encarregue por este controle.

Criando o servlet de carga de dados

O leitor mais atento deve ter encontrado algumas ausências de imports em três listagens do artigo anterior. Vamos a elas:

• Na **Listagem 8**, atualize o **import java.util.List** para **java .util.***;
• Na **Listagem 12**, adicione **import java.util.Date**;
• Na **Listagem 17**, adicione **import br.com.javamagazine.simulador.domain.Pergunta**.

Dando continuidade ao desenvolvimento do simulador quiz, implementaremos agora o servlet responsável por efetuar a carga automática dos dados da aplicação, que é um procedimento independente e necessário para o correto funcionamento do sistema. Normalmente os dados seriam carregados dinamicamente via SGBD, mas para simplificar o exemplo, o faremos com recursos do Hibernate. Para isso, utilizaremos o **EntityManager**, que possibilita executar operações com o banco de dados da mesma forma que o faríamos diretamente através de scripts SQL.

Sendo assim, crie um pacote de nome **br.com.javamagazine.simulador.persistence** e depois um servlet chamado **InitServlet**. Feito isso, substitua o código gerado pelo Eclipse para ficar semelhante ao da **Listagem 1**.

Ao compilar esse código podemos verificar que temos um erro de compilação. No entanto, deixaremos assim até implementar a classe **HibernateUtil**, mais adiante.

Os principais trechos deste código são explicados a seguir:
• Linha 04: A anotação **@WebServlet** faz com que o container execute este servlet assim que o usuário acessar a página inicial da aplicação, mapeada no atributo **urlPatterns**;
• Linha 05: Estendemos a classe abstrata **HttpServlet**. Desta forma, devemos codificar pelo menos um de seus métodos de serviço, neste exemplo codificamos o **doGet**();
• Linhas 12 e 13: Inicializamos as propriedades **em** e **tx** através da classe utilitária **HibernateUtil**, que será implementada mais adiante;
• Linhas 14 e 15: Inicializamos as variáveis **qtdeCargaBaseDados** e **porcentualAcerto** com valores obtidos de parâmetros

Listagem 1. Código da classe InitServlet.

```
01  package br.com.javamagazine.simulador.persistence;
02  //imports omitidos...
03
04  @WebServlet(urlPatterns={"/pagina_inicial.xhtml"},loadOnStartup=1)
05  public class InitServlet extends HttpServlet {
06    private EntityManager em;
07    private EntityTransaction tx;
08    private Integer qtdeCargaBaseDados;
09    private Double porcentualAcerto;
10
11    public void init(ServletConfig config) throws ServletException {

12      this.em = HibernateUtil.getEntityManager();
13      this.tx = this.em.getTransaction();
14      this.qtdeCargaBaseDados = Integer.parseInt(config .getServletContext()
        .getInitParameter("qtde_dados_carga_base"));
15      this.porcentualAcerto = Integer.parseInt(config .getServletContext()
        .getInitParameter("porcentual_acerto_avaliacao"));
16    }
17
18    protected void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {
19        this.inicializarTemas();
20        this.inicializarPerguntas();
21        this.inicializarRespostas();
22        this.inicializarAvaliacoes();
23        RequestDispatcher rd = request.getRequestDispatcher
          ("/avaliacao/simulador.xhtml");
24        rd.forward(request, response);
25    }
26
27    private int sortear(int limite) {
28      return (int) ((Math.random() * limite - 1) + 1);
29    }
30
31    private void inicializarTemas() {
32      tx.begin();
33      em.persist(new Tema("Tema 1"));
34      em.persist(new Tema("Tema 2"));
35      tx.commit();
36    }
37
38    private void inicializarPerguntas() {
39      List<Tema> temas = (this.em.createNamedQuery(Tema.LISTAR_TEMAS,
        Tema.class).getResultList());
40      List<NivelEnum> niveis = Arrays.asList(NivelEnum.values());
41      List<TipoQuestaoEnum> tipos = Arrays.asList(TipoQuestaoEnum.values());
42      for (Tema tema : temas) {
43        for (int i=1; i<=this.qtdeCargaBaseDados; i++) {
44          NivelEnum nivel = niveis.get(this.sortear(niveis.size()));
45          TipoQuestaoEnum tipoQuestao = tipos.get(this.sortear(tipos.size()));
46          tx.begin();
47        em.persist(new Pergunta("Pergunta " + i, nivel, tipoQuestao, tema));
48          tx.commit();
49        }
50      }
51    }
52
53    private void inicializarRespostas() {
54      List<Tema> temas = (this.em.createNamedQuery(Tema.LISTAR_TEMAS,
      Tema.class).getResultList());
55      for (Tema tema : temas) {
56      List<Pergunta> perguntas = (this.em.createNamedQuery
      (Pergunta.LISTAR_TEMA_PERGUNTAS, Pergunta.class).setParameter
      ("tema", tema).getResultList());
57        for (int i=0; i<perguntas.size(); i++) {
58          for (int j=0; j<this.qtdeCargaBaseDados; j++) {
59            tx.begin();
60            em.persist(new Resposta("Reposta " + j, perguntas.get(j), this.isCorreta()));
61            tx.commit();
62          }
63        }
64      }
65    }
66
67    private void inicializarAvaliacoes() {
68      List<Tema> temas = (this.em.createNamedQuery(Tema.LISTAR_TEMAS,
      Tema.class).getResultList());
69      for (int i=0; i<temas.size(); i++) {
70        List<Pergunta> perguntas = (this.em.createNamedQuery
        (Pergunta.LISTAR_TEMA_PERGUNTAS, Pergunta.class).setParameter
        ("tema", temas.get(i))).getResultList();
71        tx.begin();
72        this.em.persist(new Avaliacao("Avaliacao " + i, perguntas, perguntas.size(),
         Cronometro.configurarTempoAvaliacao(0, 1, 30), temas.get(i)));
73        tx.commit();
74      }
75    }
76
77    public void destroy() {
78      HibernateUtil.closeEntityManagerFactory();
79    }
80  }
```

provenientes do arquivo *web.xml*. Fizemos desta forma para incentivar o uso desta prática, que torna o sistema mais flexível a mudanças. Assim podemos modificar o valor dos parâmetros sem ter a necessidade de alterar o código fonte da aplicação. O único procedimento necessário para que as alterações surtam efeito é a reinicialização do servidor;

- Linha 18: Chama o método de serviço **doGet()**, utilizado para realizar a carga automática dos dados na seguinte ordem de prioridade: temas, perguntas, respostas e avaliações. Ao final do procedimento de carga, fazemos, através do método **forward()**, com que a solicitação seja encaminhada para a página onde ocorrerá a avaliação;
- Linhas 27, 28 e 29: Como a carga de dados é feita automaticamente, optamos por selecionar as informações de forma aleatória para montar os objetos que serão cadastrados no banco de dados. Por exemplo: uma pergunta deverá ter um nível de dificuldade. Para selecionar qual nível essa pergunta terá, enviamos para o método **sortear()** a quantidade de níveis disponíveis e ele se encarregará de retornar um número aleatório entre zero e o valor recebido como parâmetro. O retorno deste método será utilizado como índice para selecionar um nível em uma lista. Também adotamos esta lógica para selecionar objetos que representam os tipos de questões (simples, múltipla escolha ou arrastar e soltar);
- Linha 31 a 36: Realizamos o cadastro dos temas;
- Linha 38 a 51: Realizamos o cadastro das perguntas;
- Linha 53 a 65: Realizamos o cadastro das respostas;
- Linha 67 a 75: Realizamos o cadastro das avaliações do simulador;
- Linha 78: De acordo com o ciclo de vida de um servlet, o último método a ser executado é o **destroy()**. Desta forma, liberamos aqui todos os recursos utilizados pelo Hibernate durante a carga automática dos dados.

Criando o controlador de transações e a fábrica de DAOs

Nosso próximo passo é criar as classes que darão suporte à aplicação nas operações com o banco de dados. Nestas classes implementaremos toda a lógica de acesso ao MySQL e então delegaremos o controle das transações para o container. Nossa intenção ao adotar um controlador de transações é centralizar todo esse controle em uma só classe. Para fazer com que o Tomcat seja o gerente destas transações, implementaremos nesta classe a interface **Filter**, como veremos mais adiante.

Para isso, crie um pacote, de nome **br.com.javamagazine.simulador.persistence**, e depois uma classe, chamada **HibernateUtil**. Da mesma forma que fizemos anteriormente, modifique o código gerado pelo Eclipse para que fique semelhante ao da **Listagem 2**.

Para ter acesso à unidade de persistência, declaramos na linha 8 uma variável estática do tipo **String** e a inicializamos com o mesmo valor da tag **persistence-unit**, definida no arquivo *persistence.xml*. Na linha 9 instanciamos a variável de classe **manager**, que será adicionada no escopo **ThreadLocal** dentro do método **getEntityManager()**. Desta forma, teremos uma única instância de **EntityManager** disponível para uso durante toda a execução de uma solicitação e será possível compartilhá-la com outros componentes da aplicação envolvidos no tratamento desta requisição. Por exemplo, ao selecionar uma avaliação na página *simulador.xhtml*, consultamos todas as suas perguntas e suas respectivas respostas utilizando apenas uma instância de **EntityManager**.

Listagem 2. Código da classe HibernateUtil.

```
01  package br.com.javamagazine.simulador.persistence;
02
03  import javax.persistence.*;
04  import org.hibernate.Session;
05
06  public class HibernateUtil {
07
08    private static final String PERSISTENCE_UNIT_NAME = "simuladorPU";
09    private static ThreadLocal<EntityManager>
      manager = new ThreadLocal<EntityManager>();
10    private static EntityManagerFactory factory;
11
12    public static boolean isEntityManagerOpen() {
13      return HibernateUtil.manager.get() != null && HibernateUtil.manager
        .get().isOpen();
14    }
15
16    public static void closeEntityManagerFactory() {
17      closeEntityManager();
18      HibernateUtil.factory.close();
19    }
20
21    public static void closeEntityManager() {
22      EntityManager em = HibernateUtil.manager.get();
23      if (em != null) {
24        EntityTransaction tx = em.getTransaction();
25        if (tx.isActive()) {
26          tx.commit();
27        }
28        em.close();
29        HibernateUtil.manager.set(null);
30        HibernateUtil.manager.remove();
31      }
32    }
33
34    public static EntityManager getEntityManager() {
35      if (HibernateUtil.factory == null) {
36        HibernateUtil.factory = Persistence.createEntityManagerFactory
          (PERSISTENCE_UNIT_NAME);
37      }
38      EntityManager em = HibernateUtil.manager.get();
39      if (em == null || !em.isOpen()) {
40        em = HibernateUtil.factory.createEntityManager();
41        HibernateUtil.manager.set(em);
42      }
43      return em;
44    }
45  }
```

Na linha 10 declaramos a variável **factory**, do tipo **EntityManagerFactory**, pois somente com ela conseguimos recuperar uma instância de **EntityManager**. Já na linha 13 codificamos o método **isEntityManagerOpen()**, onde checamos se a conexão com a base de dados está aberta. Entre as linhas 16 e 32, implementamos os métodos **closeEntityManagerFactory()** e **closeEntityManager()**, que se encarregarão de fechar a conexão com o banco de dados e liberar os recursos utilizados pelo Hibernate.

Já no método **getEntityManager()**, mais precisamente na linha 36, obtemos de fato o acesso à unidade de persistência da aplicação. Nas linhas 40 e 41 criamos uma instância de **EntityManager** e a adicionamos de fato no escopo **ThreadLocal** para que tenhamos apenas uma ocorrência da mesma por solicitação no sistema. E para finalizar, na linha 43 retornamos o **EntityManager** preparado para efetuar operações no MySQL.

Dando continuidade ao desenvolvimento do simulador, criaremos agora a classe **TransactionFilter**, que terá a função de delegar o controle das transações para o Tomcat.

Por ser um filtro, esta classe tem seu ciclo de vida monitorado pelo container. Desta forma, podemos codificar o gerenciamento das transações dentro de seus métodos de call-back (**init()**, **doFilter()** e **destroy()**), abrindo e fechando as transações por intermédio do container. Para isso, crie esta classe no mesmo pacote da classe **HibernateUtil** e substitua o código gerado para que fique igual ao da **Listagem 3**.

Listagem 3. Código da classe TransactionFilter.

```
01  package br.com.javamagazine.simulador.persistence;
02
03  import java.io.IOException;
04  import javax.persistence.*;
05  import javax.servlet.*;
06  import javax.servlet.annotation.WebFilter;
07
08  @WebFilter("/*")
09  public class TransactionFilter implements Filter {
10
11    @Override
12    public void init(FilterConfig filterConfig) throws ServletException {}
13
14    @Override
15    public void doFilter(ServletRequest request, ServletResponse response,
      FilterChain chain) throws IOException, ServletException {
16      EntityManager em = HibernateUtil.getEntityManager();
17      EntityTransaction tx = em.getTransaction();
18      try {
19        tx.begin();
20        chain.doFilter(request, response);
21        tx.commit();
22      } catch (Exception e) {
23        if (tx != null && tx.isActive()) {
24          tx.rollback();
25        }
26      } finally {
27        if (em.isOpen()) {
28          em.close();
29        }
30      }
31
32    @Override
33    public void destroy() {
34      HibernateUtil.closeEntityManagerFactory();
35    }
36  }
```

Para que o container seja capaz de executar a classe **TransactionFilter**, declaramos a anotação **@WebFilter**, conforme o código da linha 8. A partir disso, quando a aplicação receber uma requisição, o Tomcat se encarregará de processar o filtro de transações e executar os seguintes métodos: **init()**, **doFilter()** e **destroy()**. No método **doFilter()** obtemos uma instância de **EntityTransaction** para manipular as transações por intermédio do Hibernate, como pode ser visto na linha 17. Já na linha 19 inicializamos a transação para que quando for necessário realizar alguma operação com o banco de dados, outros componentes da aplicação possam utilizá-la. Outra vantagem em utilizar um filtro de transação é que centralizamos o seu gerenciamento em um só lugar. Desta forma não precisamos nos preocupar em manipular transações em outros locais.

Ao abrir uma transação a mantemos disponível até que o sistema finalize o processamento das operações com o SGBD, como consultar perguntas de uma determinada avaliação. Caso ocorra alguma exceção proveniente de erros do sistema ou do banco de dados, desfazemos todas as operações realizadas, conforme implementado na linha 24. Finalmente, na linha 34 liberamos todos os recursos utilizados pelo Hibernate.

Criando as interfaces DAO e suas implementações

A arquitetura do simulador é baseada em um importante conceito: a programação para interfaces, uma prática comum em linguagens orientadas a objetos que deixa a aplicação mais flexível a mudanças. Esta prática permite o uso do polimorfismo, que diminui o acoplamento entre os componentes da aplicação possibilitando, por exemplo, adicionar ou substituir elementos derivados de uma mesma estrutura (classe ou interface) que tenham operações com as mesmas assinaturas, mas que imprimam diferentes comportamentos no sistema.

Com base no que foi dito, criaremos agora a primeira interface da aplicação. Para simplificar nosso exemplo, definiremos apenas uma operação referente à entidade **Pergunta**. Portanto, crie o pacote **br.com.javamagazine.simulador.dao** e depois a interface **PerguntaDAO**. Feito isso, modifique o código gerado para que fique igual ao da **Listagem 4**.

Listagem 4. Código da interface PerguntaDAO.

```
01  package br.com.javamagazine.simulador.dao;
02
03  import br.com.javamagazine.simulador.domain.Pergunta;
04
05  public interface PerguntaDAO {
06
07    public Pergunta buscar(Long id);
08
09  }
```

Por contrato, a classe que implementar esta interface deverá codificar o método **buscar()**, declarado na linha 7. Feito isso, será possível consultar uma **Pergunta** no banco de dados a partir do *id*. O parâmetro recebido como argumento neste método representa uma propriedade da classe **Pergunta** e foi escolhido por ser o seu principal campo e representar a chave primária da tabela.

Dando continuidade ao processo de criação das interfaces da aplicação, criaremos a interface que disponibilizará as operações

referentes à entidade **Avaliacao**. Desta forma, seguindo os mesmos passos para criar **PerguntaDAO**, gere a interface **AvaliacaoDAO** e substitua seu código pelo da **Listagem 5**.

A diferença entre a listagem anterior e a atual está na linha 9, no método **listar()**. Ele será utilizado para listar as avaliações cadastradas pelo **InitServlet** em um combobox na página inicial do simulador, para que seja possível o usuário selecionar qual avaliação deseja realizar.

Para concluir esta etapa de construção das interfaces **DAO** da aplicação, criaremos a interface **ResultadoAvaliacaoDAO**, que se encarregará de definir a operação referente à classe **ResultadoAvaliacao**. O seu código é apresentado na **Listagem 6**.

Listagem 5. Código da interface AvaliacaoDAO.

```
01  package br.com.javamagazine.simulador.dao;
02
03  import java.util.List;
04
05  import br.com.javamagazine.simulador.domain.Avaliacao;
06
07  public interface AvaliacaoDAO {
08
09      public List<Avaliacao> listar();
10
11  }
```

Listagem 6. Código da interface ResultadoAvaliacaoDAO.

```
01  package br.com.javamagazine.simulador.dao;
02
03  import br.com.javamagazine.simulador.domain.ResultadoAvaliacao;
04
05  public interface ResultadoAvaliacaoDAO {
06
07      public void salvar(ResultadoAvaliacao resultadoAvaliacao);
08
09  }
```

O ponto mais importante dessa listagem pode ser verificado na linha 7, com a declaração do método **salvar()**. A partir disso, a classe que implementar esta interface deverá codificar este método para que seja possível persistir um objeto **ResultadoAvaliacao** no banco de dados.

Com as interfaces prontas, apresentaremos agora suas respectivas implementações. Ao finalizar esta etapa, teremos em nosso projeto o padrão DAO. Com ele contabilizamos uma nova camada no sistema (de acesso aos dados), que é independente das camadas de visão e de negócio. Ao criar esta abstração, diminuímos o acoplamento entre os componentes e tornamos o software mais extensível. Deste modo, podemos, até mesmo, substituir todo o mecanismo de persistência da aplicação sem causar grande impacto nas outras camadas. Ao adotar este padrão também melhoramos a organização do código, pois evitamos misturar artefatos de negócio com scripts SQL ou códigos do Hibernate.

Dito isso, no mesmo pacote utilizado pelas interfaces, crie a classe **PerguntaDAOImpl** e substitua o código gerado pelo da **Listagem 7**.

Listagem 7. Código da classe PerguntaDAOImpl.

```
01  package br.com.javamagazine.simulador.dao;
02
03  import java.io.Serializable;
04  import java.util.HashMap;
05  import java.util.Map;
06
07  import javax.persistence.TypedQuery;
08
09  import br.com.javamagazine.simulador.domain.Pergunta;
10  import br.com.javamagazine.simulador.persistence.HibernateUtil;
11
12  public class PerguntaDAOImpl implements Serializable, PerguntaDAO {
13
14    private static final long serialVersionUID = 910541959344994442L;
15
16    @Override
17    public Pergunta buscar(Long id) {
18      String jpql = "select p from Pergunta p where p.id = :id";
19      Map<String, Object> params = new HashMap<>();
20      params.put("id", id);
21      TypedQuery<Pergunta> query = HibernateUtil.getEntityManager().
        createQuery(jpql, Pergunta.class);
22      for (Map.Entry<String, Object> param : params.entrySet()) {
23        query.setParameter(param.getKey(), param.getValue());
24      }
25      return query.getSingleResult();
26    }
27
28  }
```

Como podemos verificar, esta classe implementa a interface **PerguntaDAO** e, portanto, o método **buscar()**, codificado entre as linhas 16 e 26. Nele montamos uma consulta JPQL passando o *id* da pergunta como parâmetro. E já que sabemos qual será o tipo de retorno da consulta, utilizamos a interface **TypedQuery** do **Hibernate** para indicar ao compilador que esperamos como retorno um objeto do tipo **Pergunta**. Assim evitamos a necessidade de realizar conversões desnecessárias.

Nosso próximo passo é criar a classe **AvaliacaoDAOImpl**. Para isso, repita o procedimento de criação da classe anterior e substitua o código gerado via Eclipse pelo da **Listagem 8**.

Esta classe implementa a interface **AvaliacaoDAO** e por isso, nas linhas 14 a 16 codificamos o método **listar()**. Com ele retornamos uma listagem de todas as avaliações cadastradas no sistema. A parte mais importante desta listagem pode ser verificada na linha 15, onde obtemos uma referência do **EntityManager** criada pelo container, por intermédio da classe **HibernateUtil**, para executar a listagem dos dados.

Finalmente, vamos criar a última implementação das interfaces do pacote DAO. Portanto, crie a classe **ResultadoAvaliacaoDAOImpl** e a codifique conforme a **Listagem 9**.

Nesta classe implementamos a interface **ResultadoAvaliacaoDAO** e consequentemente seu método **salvar()**, que recebe um objeto do tipo **ResultadoAvaliacao** e o persiste no banco de dados com o Hibernate.

Por fim, codificaremos a fábrica de DAOs, responsável por centralizar a criação das instâncias dos DAOs da aplicação. Deste modo, utilize o mesmo pacote da classe **InitServlet**, apresentada na **Listagem 1**, e gere a classe **DAOFactory** com o código exposto na **Listagem 10**.

Listagem 8. Código da classe AvaliacaoDAOImpl.

```
01  package br.com.javamagazine.simulador.dao;
02
03  import java.io.Serializable;
04  import java.util.List;
05
06  import br.com.javamagazine.simulador.domain.Avaliacao;
07  import br.com.javamagazine.simulador.persistence.HibernateUtil;
08
09  public class AvaliacaoDAOImpl implements Serializable, AvaliacaoDAO {
10
11    private static final long serialVersionUID = -4899619244944350786L;
12
13    @Override
14    public List<Avaliacao> listar() {
15      return HibernateUtil.getEntityManager().createNamedQuery
          (Avaliacao.LISTAR_TODOS, Avaliacao.class).getResultList();
16    }
17
18  }
```

Listagem 9. Código da classe ResultadoAvaliacaoDAOImpl.

```
01  package br.com.javamagazine.simulador.dao;
02
03  import java.io.Serializable;
04
05  import br.com.javamagazine.simulador.domain.ResultadoAvaliacao;
06  import br.com.javamagazine.simulador.persistence.HibernateUtil;
07
08  public class ResultadoAvaliacaoDAOImpl implements Serializable,
    ResultadoAvaliacaoDAO {
09
10    private static final long serialVersionUID = -4899619244944350786L;
11
12    @Override
13    public void salvar(ResultadoAvaliacao resultadoAvaliacao) {
14      HibernateUtil.getEntityManager().persist(resultadoAvaliacao);
15    }
16
17  }
```

Listagem 10. Código da classe DAOFactory.

```
01  package br.com.javamagazine.simulador.persistence;
02
03  import br.com.javamagazine.simulador.dao.*;
04
05  public class DAOFactory {
06
07    public static PerguntaDAO criarPerguntaDAO() {
08      return new PerguntaDAOImpl();
09    }
10
11    public static AvaliacaoDAO criarAvaliacaoDAO() {
12      return new AvaliacaoDAOImpl();
13    }
14
15    public static ResultadoAvaliacaoDAO criarResultadoAvaliacaoDAO() {
16      return new ResultadoAvaliacaoDAOImpl();
17    }
18
19  }
```

Podemos observar nesta listagem que para cada classe que implementa uma interface da camada **DAO**, temos um método equivalente declarado. Por exemplo, na linha 7 declaramos o método **criarPerguntaDAO()**, que retorna uma instância de **PerguntaDAOImpl**. Na linha 11, temos o método **criarAvaliacaoDAO()** que, por sua vez, retorna uma instância de **AvaliacaoDAOImpl**. Por fim, temos na linha 15 a declaração do método **criarResultadoAvaliacaoDAO()**, que retorna uma instância de **ResultadoAvaliacaoDAOImpl**.

Criando as interfaces de negócio e suas implementações

Seguindo a mesma linha de raciocínio de programar para interfaces, criaremos agora as interfaces de negócio da aplicação. Com isso, aumentaremos um pouco mais a complexidade da arquitetura, pois adicionaremos mais uma camada. Em contrapartida, deixaremos o código mais extensível e organizado. Nosso objetivo ao fazer isso é separar as regras de negócio em uma camada à parte e que seja independente das outras (modelo, persistência e visão).

Para simplificar o nosso exemplo, na maioria das vezes esta camada servirá apenas para conectar as camadas de acesso a dados aos controladores da aplicação – mais uma forma de diminuir o acoplamento entre os componentes.

Todas as interfaces de definição das regras de negócio da aplicação terão o sufixo BC (*Business Controller*) e serão semelhantes às interfaces da camada DAO, pois na maioria das vezes as utilizaremos apenas como intermediadoras entre os componentes de visão e de acesso a dados.

Dito isso, começaremos pela interface que determina a única operação relacionada à entidade **Pergunta**. Assim, crie um novo pacote, de nome **br.com.javamagazine.simulador.business**, e depois a classe **PerguntaBC**. Em seguida, substitua o código gerado pelo Eclipse para que fique igual ao da **Listagem 11**.

Listagem 11. Código da interface PerguntaBC.

```
01  package br.com.javamagazine.simulador.business;
02
03  import br.com.javamagazine.simulador.domain.Pergunta;
04
05  public interface PerguntaBC {
06
07    public Pergunta buscar(Long id);
08
09  }
```

Note que esta interface possui apenas o método **buscar()**. Como dissemos previamente, ela é semelhante à interface **PerguntaDAO**, que também possui um método com a mesma assinatura.

Nosso próximo passo é criar a interface **AvaliacaoBC**, que definirá as operações relacionadas à classe **Avaliacao**. Seu código é apresentado na **Listagem 12** e como verificado, possui apenas o método **listar()**.

Por fim, criaremos a última interface do pacote de negócios, **ResultadoAvaliacaoBC**, que define apenas uma operação

relacionada à entidade **ResultadoAvaliacao**. Seu código deve ficar semelhante ao da **Listagem 13**.

Listagem 12. Código da interface AvaliacaoBC.

```
01  package br.com.javamagazine.simulador.business;
02
03  import java.util.List;
04
05  import br.com.javamagazine.simulador.domain.Avaliacao;
06
07  public interface AvaliacaoBC {
08
09     public Long decrementarTempo(Long tempoAvaliacao);
10
11     public List<Avaliacao> listar();
12
13  }
```

Listagem 13. Código da interface ResultadoAvaliacaoBC.

```
01  package br.com.javamagazine.simulador.business;
02
03  import br.com.javamagazine.simulador.domain.ResultadoAvaliacao;
04
05  public interface ResultadoAvaliacaoBC {
06
07     public void salvar(ResultadoAvaliacao resultadoAvaliacao);
08
09  }
```

Concluída esta etapa, podemos passar para a codificação das classes que implementam as interfaces de negócio. Para seguir a mesma ordem de criação das interfaces, começaremos pela classe **PerguntaBCImpl**. Como seu próprio nome sugere, ela implementa o método da interface **PerguntaBC**. Seu código é apresentado na **Listagem 14**.

Com relação a essa listagem, vale observar as linhas 16 e 20. Na linha 16, obtemos uma instância de **PerguntaDAOImpl** através da fábrica de DAOs e repassamos sua referência para a propriedade **perguntaDAO**. Já na linha 20, em vez de implementarmos a busca neste método, chamamos o método implementado na classe **PerguntaDAOImpl**, uma vez que consultar dados é responsabilidade da camada **DAO**.

Agora, codificaremos a classe **AvaliacaoBCImpl**, responsável por implementar os métodos da interface **AvaliacaoBC**. Apresentamos seu código na **Listagem 15**.

Nesta listagem, inicializamos no construtor uma instância de **AvaliacaoDAOImpl**, que depois será utilizada para chamar o método **listar()**. Verificamos também que na linha 20 declaramos o método **decrementarTempo()**, o único que trabalha diretamente com regras de negócio nesta camada da aplicação. Ele poderia ter sido codificado em outro lugar, como em uma classe controladora, mas o implementamos aqui para ilustrar a forma como esta camada pode ser utilizada. Como o próprio nome do método sugere, sua função é retornar um tempo com seu valor decrementado. Para isso, fazemos uso da classe utilitária **Cronometro**. Finalmente, na linha 26 temos o método **listar()**, que retorna uma listagem das avaliações cadastradas no sistema por intermédio do método equivalente da camada **DAO**.

Listagem 14. Código da interface PerguntaBCImpl.

```
01  package br.com.javamagazine.simulador.business;
02
03  import java.io.Serializable;
04
05  import br.com.javamagazine.simulador.dao.PerguntaDAO;
06  import br.com.javamagazine.simulador.domain.Pergunta;
07  import br.com.javamagazine.simulador.persistence.DAOFactory;
08
09  public class PerguntaBCImpl implements Serializable, PerguntaBC {
10
11     private static final long serialVersionUID = -862450388803283219L;
12
13     private PerguntaDAO perguntaDAO;
14
15     public PerguntaBCImpl() {
16        this.perguntaDAO = DAOFactory.criarPerguntaDAO();
17     }
18
19     public Pergunta buscar(Long id) {
20        return this.perguntaDAO.buscar(id);
21     }
22
23  }
```

Listagem 15. Código da interface AvaliacaoBCImpl.

```
01  package br.com.javamagazine.simulador.business;
02
03  import java.io.Serializable;
04  import java.util.List;
05
06  import br.com.javamagazine.simulador.dao.AvaliacaoDAO;
07  import br.com.javamagazine.simulador.domain.Avaliacao;
08  import br.com.javamagazine.simulador.persistence.DAOFactory;
09
10  public class AvaliacaoBCImpl implements Serializable, AvaliacaoBC {
11
12     private static final long serialVersionUID = -3018194744635275029L;
13
14     private AvaliacaoDAO avaliacaoDAO;
15
16     public AvaliacaoBCImpl() {
17        this.avaliacaoDAO = DAOFactory.criarAvaliacaoDAO();
18     }
19
20     public Long decrementarTempo(Long tempoAvaliacao) {
21        Calendar cal = Cronometro.decrementarTempoAvaliacao(tempoAvaliacao);
22        return Cronometro.configurarTempoAvaliacao(cal.get(Calendar.HOUR),
           cal.get(Calendar.MINUTE), cal.get(Calendar.SECOND));
23     }
24
25     public List<Avaliacao> listar() {
26        return this.avaliacaoDAO.listar();
27     }
28
29  }
```

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

http://www.devmedia.com.br

Para encerrar esta etapa, criaremos a última classe do pacote de negócios, **ResultadoAvaliacaoBCImpl**. Ela define uma operação de persistência relacionada à entidade **ResultadoAvaliacao**. Seu código pode ser analisado na **Listagem 16**.

Listagem 16. Código da interface ResultadoAvaliacaoBCImpl.

```
01  package br.com.javamagazine.simulador.business;
02
03  import java.io.Serializable;
04
05  import br.com.javamagazine.simulador.dao.ResultadoAvaliacaoDAO;
06  import br.com.javamagazine.simulador.domain.ResultadoAvaliacao;
07  import br.com.javamagazine.simulador.persistence.DAOFactory;
08
09  public class ResultadoAvaliacaoBCImpl implements Serializable,
    ResultadoAvaliacaoBC {
10
11     private static final long serialVersionUID = -3018194744635275029L;
12
13     private ResultadoAvaliacaoDAO resultadoavaliacaoDAO;
14
15     public ResultadoAvaliacaoBCImpl() {
16        this.resultadoavaliacaoDAO = DAOFactory.criarResultadoAvaliacaoDAO();
17     }
18
19     @Override
20     public void salvar(ResultadoAvaliacao resultadoAvaliacao) {
21        this.resultadoavaliacaoDAO.salvar(resultadoAvaliacao);
22     }
23
24  }
```

Neste código, seguimos a mesma lógica adotada pelas classes que implementam as interfaces de negócio da aplicação. Na linha 16, dentro do construtor, repassamos uma instância de **ResultadoAvaliacaoDAO** para a propriedade **resultadoAvaliacaoDAO**, que será utilizada mais à frente na chamada ao método **salvar()**.

A diferença entre esta classe e as demais deste pacote é que aqui implementamos um método de persistência, como podemos observar na linha 20. Já nos demais métodos que acessam o banco de dados, implementamos apenas rotinas que consultam informações.

Finalmente, da mesma forma que fizemos anteriormente, vamos redirecionar as chamadas a este método (**salvar()**) ao seu equivalente da classe **ResultadoAvaliacaoDAOImpl**, pois foi onde implementamos de fato a persistência através do Hibernate.

Para desenvolver um software de qualidade, uma das nossas principais preocupações deve ser a sua arquitetura. Hoje sabemos que é de fundamental importância para o sistema projetar uma arquitetura bem estruturada, definida. Assim, temos mais chances de alcançar o resultado desejado e garantir uma boa manutenibilidade. Com uma boa arquitetura, também simplificamos a evolução das funcionalidades do sistema e a tornamos mais produtiva, evitando ou reduzindo possíveis impactos em outros componentes.

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.

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/

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!

Usabilidade em sistemas corporativos

Conheça um pouco de usabilidade e tenha um caminho para seguir nos projetos que você está trabalhando

Fique por dentro

Este artigo é útil para analistas, desenvolvedores ou outros profissionais envolvidos no desenvolvimento de software que têm o interesse em conhecer os conceitos de usabilidade e compreender como aplicá-los no dia a dia mesmo em pequenos projetos, com prazos curtos e orçamentos baixos. Para isso, serão apresentadas as características estudadas na usabilidade, técnicas e também exemplos reais para facilitar o entendimento. Assim, você irá adquirir conhecimentos que lhe permitirão discutir sobre o assunto e tomar decisões mais adequadas em seus projetos, bem como saberá como continuar os estudos caso queira se aprofundar no assunto.

É muito comum a preocupação com a usabilidade e experiência do usuário em sites, e-commerces, softwares vendidos como serviços (SaaS) ou outros sistemas com muitos usuários. Apesar disso, a busca por uma boa usabilidade pode ser almejada por analistas e desenvolvedores em qualquer projeto, até mesmo em pequenas funcionalidades ou alterações em um sistema já existente.

Grandes empresas costumam contratar profissionais especializados nesta área ou recorrem a consultorias externas para melhorar a usabilidade do seu produto, porém, esse conhecimento pode ser adquirido pelas empresas e utilizado no dia a dia, tornando desnecessária a contratação de serviços externos quando não se tem recursos para isso, por exemplo.

Ao desenvolver um sistema corporativo ou uma funcionalidade em um sistema já existente, muitas vezes pensa-se apenas no “o quê” e no “como” fazer. No entanto, mais importante do que isso é entender o “porquê”, pois ao compreender a real necessidade de um sistema/funcionalidade é possível criar algo que realmente atenda aos usuários e que faça com que eles queiram e gostem de utilizar.

Neste contexto, um dos princípios utilizados para a implementação de soluções no Google é: “Foque no usuário que o resto se ajeita” (veja a seção **Links**). Não fazer isso talvez seja um dos grandes erros cometidos por desenvolvedores de software.

A experiência do usuário e formas para compreender o que ele realmente precisa são assuntos muito interessantes, mas também muito amplos. Assim sendo, esse artigo focará nos elementos estudados na usabilidade explicados através de exemplos que você poderá aplicar em projetos que está trabalhando. Conhecendo o básico sobre usabilidade torna-se possível aprimorar alguns itens no seu projeto já nas etapas iniciais, como no brainstorm ou na análise de requisitos, e estas alterações poderão gerar vantagens significativas ao aumentar a satisfação do usuário.

É comum um usuário não ter uma experiência agradável ao tentar realizar tarefas simples, como fazer o check-in para seu voo, e não conseguir finalizar o que queria ou ter dificuldades para isso. Outro exemplo relevante, identificado a partir de alguns testes de usabilidade, sinaliza que um em cada três usuários que tentavam realizar alguma tarefa em e-commerces não conseguia completar seu objetivo com sucesso. Essa visão fez as empresas começarem a se preocupar com a usabilidade, pois elas perceberam que estavam perdendo dinheiro.

Por sua vez, em sistemas corporativos, onde esse prejuízo não é tão notável por ser difícil de ser mensurado, existem danos que também podem ser muito negativos para a empresa, como: diminuição da produtividade do usuário, estresse e insatisfação, falha na comunicação da equipe, entre outros.

Com base nisso, neste artigo serão apresentados os conceitos que compõem a base da usabilidade, exemplos e casos reais que poderão ser adotados como referência nos projetos em que você está trabalhando.

O que é Usabilidade?

O termo usabilidade é estudado por diversas áreas e se refere à forma de uso de algo. A usabilidade diz respeito à facilidade que o usuário tem de aprender a usar uma interface e atingir seu objetivo sem ter mais problemas do que ele está disposto a aceitar.

A norma ISO 9241 define usabilidade como uma "medida na qual um produto pode ser usado por usuários específicos para alcançar objetivos específicos com eficácia, eficiência e satisfação em um contexto específico de uso".

Ademais, diversos autores e normas apresentam atributos considerados essenciais para a usabilidade. Os mais comuns na literatura estão relacionados à condução dos usuários (forma de guiá-los durante o uso do sistema), controle e adaptação à tarefa (controle que o usuário possui e a capacidade do sistema de se adaptar conforme a necessidade), consistência (na exibição e nos resultados de ações), gestão de erros (evitar erros e o que fazer quando eles acontecem) e carga de trabalho (tudo aquilo que é apresentado em tela); conceitos estes que serão apresentados a seguir.

Condução

A condução está relacionada à forma de guiar os usuários durante o uso de um sistema, apresentando todas as informações, instruções e avisos necessários.

Ela pode ser dividida em:

• **Presteza:** Possibilitar que o usuário identifique onde está e fornecer ferramentas de ajuda para melhorar a navegação;

• **Agrupamento e distinção de itens:** Organização visual dos itens e suas relações. Ordenar, posicionar e distinguir o que é apresentado de forma clara;

• **Feedback imediato:** Sempre que possível, dar respostas às ações do usuário no momento em que elas ocorrerem;

• **Legibilidade:** Preocupação com as características das informações apresentadas que possam dificultar ou facilitar a leitura desta, como tamanho da fonte, contraste letra/fundo, espaçamentos, etc.

Controle e adaptação à tarefa

O usuário precisa ter controle de toda a sua interação com o sistema. Dessa forma, erros e ambiguidades são limitados. Além disso, o sistema deve processar somente as ações solicitadas pelo usuário e somente quando solicitado a fazê-lo.

O sistema será mais bem aceito pelos usuários se eles tiverem controle sobre o que acontece no sistema. O usuário precisa, por exemplo, conseguir cancelar qualquer ação que esteja realizando, podendo voltar facilmente ao estado anterior da aplicação. Por exemplo: ele não deveria assistir a um vídeo que ele não possa pausar ou voltar para o início.

Como uma única interface dificilmente atenderá a todos os seus usuários em potencial, pois cada usuário é único (visto que possui experiências e conhecimentos diferentes), as interfaces devem se adaptar aos diferentes tipos de usuários. Para isso, é necessário diferenciar usuários novatos de usuários com mais experiência, dando subsídios como tutoriais passo-a-passo para os novatos e fornecendo atalhos e funcionalidades mais avançadas para os experientes.

Consistência

A consistência é o princípio que garante que:

• Informações semelhantes sejam sempre apresentadas da mesma forma;

• Uma mesma ação, mesmo que em lugares diferentes, tenha o mesmo efeito.

A consistência facilita a aprendizagem e desenvolve a previsibilidade do sistema, visto que usuários, por meio do que já conhecem, podem aprender a usar outras partes do sistema sem grandes dificuldades.

Gestão de erros

A gestão de erros visa evitar ou reduzir o número de erros e o tratamento adequado para quando ocorrer algo inesperado. Nem todas as falhas podem ser previstas e evitadas, mas deve haver um esforço para isso e para deixar o sistema preparado para possibilitar ao usuário compreender o que aconteceu de errado e dar continuidade ao que estava fazendo.

A gestão de erros pode ser dividida em:

• **Proteção contra os erros:** Uso de todos os meios para detectar e prevenir erros no sistema. Um exemplo básico seria a utilização de máscaras para impedir a digitação incorreta de um e-mail, por exemplo, pelo usuário;

• **Correção de erros:** Deve ser indicado ao usuário onde aconteceu o erro e como corrigir. Por exemplo, deve ser indicado o campo que não foi preenchido corretamente e informar qual seria o formato adequado. Podem ser exibidos caminhos alternativos ou formas de entrar em contato com a empresa ou equipe de suporte para resolver o problema;

• **Qualidade das mensagens de erro:** As mensagens devem sempre ser claras ao usuário e com informações úteis, explicando o que aconteceu e o que pode ser feito em seguida.

Muitos autores tratam o conceito de gestão de erros focando apenas em entradas de dados ou comandos incorretos realizados pelos usuários, mas todas as técnicas – como "qualidade nas mensagens" – podem ser empregadas para qualquer tipo de erro que aconteça no sistema, até mesmo um problema de infraestrutura, como um banco de dados fora do ar.

Carga de trabalho

A carga de trabalho é o conjunto de todos os elementos apresentados em tela mais as ações realizadas pelo usuário que não sejam intuitivas. Assim, quanto mais botões, textos, imagens, entre outros elementos existirem na tela e tudo que faça o usuário ter dúvidas, que não seja claro no primeiro momento e o faça pensar em como realizar tal ação, aumentam a carga de trabalho.

Em termos de usabilidade, recomenda-se a diminuição da carga de trabalho a fim de reduzir a probabilidade de o usuário cometer erros. A tarefa cognitiva imposta ao usuário deve ser breve, concisa, com o mínimo possível de ações e com baixa densidade informacional. Desta forma o usuário poderá desempenhar suas tarefas de forma mais eficiente.

Experiência do Usuário (UX)

A experiência do usuário (ou *User eXperience*), por sua vez, não envolve apenas a facilidade do uso e conseguir atingir os objetivos como a usabilidade propõe. Esse conceito também está relacionado aos sentimentos e emoções do usuário antes, durante e depois de utilizar o sistema.

Muitos autores ilustram essa diferença com o exemplo de uma rodovia com alta usabilidade e outra com boa experiência do usuário. Uma rodovia com alta usabilidade é larga, tem poucas curvas, não tem subidas e descidas, não tem tráfego contrário, não tem buracos e é bem sinalizada.

Mas uma rodovia dessas, por mais que faça você chegar rapidamente ao seu destino, pode ser monótona, maçante e até estressante. Por sua vez, uma rodovia com boa experiência do usuário proporciona outros elementos, como ter como vista o mar e/ou montanhas, você sentir o cheiro da natureza, você passar ao lado de um rio, atravessar cidades com culturas diferentes, e encontrar com facilidade lojas com todo o tipo de comida, bebida e artesanato. Por mais que você não chegue tão rápido ao seu destino, sua experiência pode ser melhor nesta segunda rodovia, não acha?

Em software, a experiência do usuário começa desde o primeiro contato com o produto, como num anúncio, e vai além do uso, como em questões de engajamento com a marca, ou seja, também envolve questões como atendimento pelos canais de contato, entrega do produto, prazos, pós-venda, surpreender o cliente ao superar expectativas e o que mais você conseguir para proporcionar mais prazer ao uso do sistema.

A usabilidade pode ser essencial para o sucesso do seu sistema, mas pensar de forma mais ampla, em toda a experiência do usuário, pode ser um grande diferencial.

Arquitetura da Informação

O conceito de Arquitetura da Informação (AI) pode ser simplificado como a forma de organizar o conteúdo de tal modo que as pessoas consigam encontrar o que procuram. Alguns autores estruturam a arquitetura da informação em quatro sistemas:

• **Organização:** Criação de categorias e agrupamentos de todas as informações apresentadas para atender os objetivos do negócio e as necessidades dos usuários;

• **Navegação:** Todas as maneiras de navegar através do conteúdo, possibilitando ao usuário saber onde ele está e para onde pode ir;

• **Busca:** Possibilidade de pesquisas para encontrar algo dentro de todo o conteúdo disponível de forma eficiente;

• **Rotulação:** Preocupação com a descrição das categorias, opções, títulos, links, etc., de uma maneira consistente e em uma linguagem significativa e clara para o usuário.

Como importante complemento, ela tem como base três variáveis que devem ser levadas em consideração sempre que você for projetar algo, são elas:

• **Usuários:** Suas tarefas, necessidades, comportamentos, experiências, etc., vão auxiliar na definição da forma que a informação será organizada e disponibilizada;

• **Conteúdo:** Os tipos de informação apresentados (notícias, histórias, produtos, comentários), tipos de documentos (imagem, vídeo, PDF, texto), o volume dos dados (os vídeos são pequenos ou grandes? São quantos gigas de dados? Isso crescerá?), quem escreve ou cria (donos do sistema, usuários externos, qualquer um), etc., influenciarão na definição de como apresentar o conteúdo aos usuários;

• **Contexto:** Toda decisão também dependerá dos objetivos estratégicos e da proposta de valor do sistema, assim como da cultura e política da empresa, recursos, restrições tecnológicas, localização, etc. É necessário entender o negócio e os objetivos por trás do sistema para fazer um melhor projeto de arquitetura da informação.

Essas variáveis são únicas para cada sistema e o arquiteto da informação deve obter e utilizá-las para conhecer as necessidades específicas e o que pode ser feito em cada projeto para tomar suas decisões.

Existem muito termos utilizados quando se fala da preocupação com o usuário (veja o **BOX 1**), cada um com suas especificidades. Porém, mais importante do que o nome utilizado é a sua atitude e as técnicas que você pode utilizar para contribuir com seus usuários.

Nota

O livro "Information Architecture for the WWW" de Rosenfeld e Morville, publicado pela editora O'Reilly, se aprofunda no assunto "Arquitetura da Informação" explicando de um modo fácil de compreender, através de analogias e exemplos, todos os conceitos de AI e como aplicá-los.

BOX 1. Muitos nomes, mas uma só preocupação: o usuário!

No final dos anos 1990, os nomes mais usados para descrever a preocupação com os usuários eram Usabilidade e Design Centrado no Usuário (UCD). Os profissionais dessas áreas eram os usabilistas e os arquitetos da informação. No entanto, com o passar dos anos, o termo Experiência do Usuário (UX) ganhou notório espaço e se tornou um dos mais citados. Outros termos que também são referenciados, mas não com tanta frequência, são: Design Interativo, Design de Interface e Gestão de Conteúdo.

Como aplicar a usabilidade no dia a dia?

A partir de agora serão apresentados conceitos com exemplos reais da adoção de usabilidade. A partir disso, você poderá transportá-los para situações que já vivenciou e para o projeto que está trabalhando no momento.

O que é mais importante para o usuário?

Para descobrir o que é mais importante para o usuário é necessário buscar o essencial, e você pode fazer isso através de algo

simples. Lembre-se, a página mais acessada no mundo tem, basicamente, uma imagem, um campo de texto e um botão.

Uma técnica que vem sendo muito utilizada no desenvolvimento de software é o *Mobile First*, onde você primeiro projeta seu sistema ou site para celulares e depois o adapta para telas maiores. Uma das principais vantagens dessa estratégia é que você é obrigado a focar nas funcionalidades principais do sistema e retirar ou dar menos importância às secundárias.

Em cada funcionalidade que projetar, pense primeiro no que é mais importante para o usuário naquela tela. Reflita sobre cada informação apresentada. Será que é mesmo necessário definir 12 colunas em uma tabela se ele, 98% das vezes, precisa somente de três? Se as outras colunas são necessárias apenas 2% das vezes, por que não as tirar da tela e permitir que sejam acessadas por um link ou talvez por outra tela? Outro exemplo, será que é necessário ter esses quatro gráficos na tela conforme o usuário solicitou? Só um não resolveria? É necessário avaliar cada caso dentro do seu contexto, pois não existe receita pronta e cada situação deve ser pensada de acordo com os objetivos da aplicação.

Antes de adicionar algo em uma funcionalidade, faça as seguintes perguntas: isso simplifica o que o usuário faz? Isso vai agregar valor para o usuário? Se a resposta for "não", provavelmente você deveria deixar de lado e não adicionar esse algo ao produto.

Dependendo do sistema que você está desenvolvendo, os usuários podem não ter muita experiência com computadores. Um exemplo disso é um sistema para medicina do trabalho, onde um médico com 75 anos que começou a usar computador há pouco tempo só precisa preencher dados em uma tela do sistema. Diante disso, quanto mais simples for a solução, quanto mais visível for o botão *Salvar*, melhor. Se ele conseguir utilizar, outras pessoas com mais experiência no uso de computadores também conseguirão. Se você começar a adicionar funcionalidades que para você pareciam interessantes, talvez apenas polua a tela e dificulte o trabalho do seu amigo médico.

Além disso, não pode-se esquecer dos usuários avançados. Deste modo, deve-se pensar também em maneiras de atrair os usuários avançados, sem interferir na simplicidade para os usuários iniciantes.

Não me faça pensar

Steve Krug, renomado escritor e consultor na área de Usabilidade, escreveu um livro com esse nome e definiu como elemento principal da usabilidade que tudo seja claro, ou pelo menos, autoexplicativo. Sendo assim, cada funcionalidade deve estar evidente, isto é, quem acessá-la deve ser capaz de:

- **Compreendê-la:** Saber o que é e para que serve;
- **Realizar seu objetivo:** Conseguir realizar o que é proposto sem dificuldades.

Portanto, busque ser autoexplicativo. Faça com que o usuário entenda o que é cada funcionalidade e que cada clique do usuário seja intuitivo e não ambíguo, isto é, em cada ação o usuário deve ter certeza que está fazendo do jeito certo para chegar onde deseja.

Nota

O livro de Steve Krug "Não me faça pensar" explica, de forma simples e divertida, os conceitos-chave de usabilidade e é leitura recomendada para quem quer estudar mais sobre o assunto.

O usuário não deve se fazer perguntas enquanto navega no sistema. Ele não deve precisar pensar em coisas como: Por onde eu começo? Onde estou agora? Por que isso tem esse nome? Onde está tal informação? Eliminar as perguntas na cabeça do usuário é um dos objetivos da usabilidade.

Use padrões e convenções

Pode parecer óbvio, mas existem muitos sistemas que botões com o mesmo comportamento, em telas semelhantes, apresentam labels (textos) diferentes (*Salvar*, *Gravar*, *Enviar* e *Registrar*, por exemplo). Além disso, se esse botão está sempre à esquerda, mantenha-o à esquerda em todas as telas. A ideia é que após estabelecer um padrão, siga-o, a não ser que você tenha uma razão específica para não o seguir como, por exemplo, uma tela específica em que ficará mais claro para o usuário se o botão ficar à direita.

Existem muitos outros padrões que você pode adotar no sistema e até mesmo entre sistemas na mesma empresa. Padrões de menus, cores, filtros, paginação, fonte, ícones, etc. É muito comum, por exemplo, as empresas estabelecerem uma guideline que oriente os desenvolvedores sobre padronizações. Se sua empresa ou cliente ainda não possui, que tal sugerir ou até mesmo criar uma?

Além de criar e estabelecer padrões, você pode tirar proveito de convenções já existentes. A placa "Pare" é um bom exemplo, sendo muito parecida em todos os países. Assim, apenas de olhar o motorista já vai saber do que se trata (veja a **Figura 1**).

Figura 1. Placa "Pare" ao redor do mundo

Apesar disso, dificilmente você vai ter a placa "Pare" no seu sistema, não é? Mesmo assim, busque, sempre que possível, utilizar convenções existentes a fim de facilitar o entendimento dos seus usuários.

Algumas convenções comuns são:

- Ícones para vídeo ou para redes sociais. Tente não criar algo totalmente novo. Use o que as pessoas estão acostumadas;
- Logo da empresa no canto superior esquerdo, que ao clicado volta para a home do sistema;
- Campo de busca no topo da tela;
- Dados do usuário/conta no canto superior direito;
- Entre outros.

Cada usuário é único

Imagine você chegando em uma cafeteria e pedindo um cappuccino com canela por cima. No dia seguinte, ao chegar nessa

mesma cafeteria, a garçonete lembra de você, lembra do seu nome e, após te cumprimentar, pergunta: "Quer um cappuccino com canela hoje?". Esse simples comportamento fará você gostar do lugar, provavelmente voltar lá mais vezes, comprar mais coisas e ainda indicar aos amigos.

Isso se chama "estratégia de personalização" e nos últimos anos tem saído do mundo físico e aparecido no mundo virtual, como em e-commerces que apresentam produtos com base em interesses individuais, ou a Netflix, que tem um algoritmo para recomendar filmes que você gostaria de assistir. Isso também é chamado de marketing one-to-one, que é quando você faz algo personalizado e não uma campanha para atingir a todos.

Essa estratégia é muito utilizada em e-mail marketing, apresentação de anúncios e produtos recomendados e pode ser utilizada dessa e de outras formas para melhorar a experiência do usuário.

Por mais que você saiba quais são as seções mais acessadas no seu sistema, existem usuários que geralmente acessarão determinada funcionalidade que não é tão popular. Diante disso, por que não criar uma seção de favoritos ou de mais acessados por usuário para facilitar a busca daquilo que cada usuário mais acessa?

Segundo a ISO 9241, o sistema é capaz de individualização quando "a interface pode ser modificada para se adaptar as necessidades da tarefa, as preferências individuais e as habilidades dos usuários".

Em um supermercado você não precisa procurar todas as vezes onde é a seção das cervejas, pois após a primeira vez que encontrar a seção você sempre vai lembrar algo como: "Está no final do primeiro corredor, perto das carnes", porém na web não existe, de forma tão clara, essa lembrança por localização. Por isso, atalhos pessoais, como favoritos, tornam-se importantes.

Você pode também armazenar informações como as pesquisas mais realizadas pelo usuário, dados selecionados, valores preenchidos e o que mais constatar que pode auxiliar o uso do sistema no dia a dia. Em uma solução com janelas ou portlets, por exemplo, onde o usuário pode modificar a posição e tamanho dos elementos, esses dados alterados podem ser armazenados e reaplicados em um próximo acesso.

Um exemplo comum é o browser, que salva o último diretório de onde você selecionou o arquivo para fazer o upload. Assim, na próxima vez que for necessário fazer um upload, a janela já será aberta neste diretório. Note que muitas vezes o usuário pode nem perceber uma funcionalidade como esta. Ele simplesmente a usa. No entanto, independentemente disso, ela facilita bastante a usabilidade, aumentando a satisfação do usuário.

Em um sistema de contratos imobiliários, por exemplo, onde cada vez que o usuário faz login ele escolhe um contrato e começa a utilizar o sistema, você poderia armazenar o último contrato selecionado e na próxima vez que o usuário fizer login, o sistema automaticamente seleciona aquele contrato.

No entanto, caso em cada login o usuário selecione um contrato diferente, não faz sentido salvar essa informação. Talvez apenas sugerir o contrato ao invés de já selecioná-lo. Lembre-se, cada sistema ou funcionalidade deve ser pensado individualmente, para verificar quais preferências de usuário podem ser salvas.

A fim de garantir performance mesmo adicionando operações no banco de dados, como as de salvar as preferências e posteriormente recuperá-las, uma possibilidade é o uso de um banco de dados não relacional como o MongoDB, que pode possibilitar operações mais rápidas (veja o **BOX 2**).

BOX 2. MongoDB

O MongoDB é um banco de dados open source não-relacional orientado a documentos que provê alta performance, disponibilidade e fácil escalabilidade. Uma das vantagens do MongoDB é a manipulação de um grande volume de dados de forma mais simples e performática que um banco de dados relacional.

Ele é um dos bancos de dados não relacionais mais utilizados e como um dos seus diferenciais, possui muito material para estudo na internet e em artigos publicados na Java Magazine, sendo adotado em projetos pequenos, com apenas um desenvolvedor, a projetos de grande porte.

O MongoDB pode ser utilizado como uma alternativa ao banco de dados relacional, porém também é comum se deparar com sistemas que adotam bancos relacionais e também o MongoDB, em busca de performance para determinadas partes do sistema, como logs (registrando ações do usuário como login, logout, tela acessadas), auditoria de dados (registrando quem alterou determinado registro e quando) e também preferências do usuário.

Valorize o tempo das pessoas oferecendo performance

Alguns filósofos dizem que o tempo é o bem mais precioso das pessoas. Pensando nisso, não é interessante que o seu projeto leve 10 segundos para abrir uma determinada tela e quase um minuto para abrir um relatório. Problemas de performances podem ocorrer por vários motivos, no entanto existem inúmeras formas de identifica-los, monitorá-los e corrigi-los.

O site Browser Diet explica várias ações que você pode tomar para deixar seu sistema mais leve e rápido (veja a seção **Links**). Ele fala sobre:
- Otimização de imagens;
- Unificação e compressão de arquivos CSS e JS;
- Dicas de códigos HTML e JS;
- Ferramentas de diagnóstico;
- Entre outras coisas.

Uma tela que exibe um menu no lado esquerdo com os meses do ano e que quando você clica em um mês exibe as faltas dos funcionários pode ficar muito lenta se tudo for consultado de uma só vez (digamos que 10 segundos para abrir a tela pela primeira vez). Ao alterar a lógica para trazer as faltas apenas no mês selecionado pode tornar a abertura da tela mais rápida (caindo para um segundo, por exemplo). O ponto negativo é que esse um segundo acontecerá cada vez que o usuário clicar em um mês. No entanto, como o usuário geralmente vê apenas o mês atual ou o mês anterior, ele não vai ficar navegando nos meses o tempo inteiro. Não necessariamente você deve adotar esta solução sempre, mas ela é válida para mostrar que existem alternativas simples para resolver problemas de performance.

Diante disso, fazer testes de desempenho com JMeter ou ferramentas similares e monitorar o desempenho através de ferramentas como VisualVM ou JProfiler é o mundo ideal. A partir disso é possível identificar os gargalos da sua aplicação, que são os pontos que mais utilizam recursos computacionais (como memória e processador), além de verificar o tempo de resposta das suas páginas.

Certas funcionalidades são testadas pelo desenvolvedor utilizando apenas um usuário por vez, mas imagine a funcionalidade de acessar a folha de pagamento. Dia 5 de todo mês, às 15:00, o departamento de RH envia um e-mail para os 1.000 funcionários da empresa com o link para acessar a folha de pagamento. Quase que simultaneamente, centenas de funcionários clicarão no link e então o sistema ou vai ficar muito lento ou vai começar a falhar em uma porcentagem dos acessos, ou pior, o servidor pode cair e ninguém mais conseguir acessar a informação desejada.

Utilizando apenas o JMeter você pode identificar que essa funcionalidade funciona com até 100 usuários simultâneos, mas com mais do que isso o sistema começa a ficar lento e com 300 ou mais usuários simultâneos, ele fica praticamente inutilizável apresentando inclusive algumas falhas. Com essas informações você pode realizar algumas ações de tuning e testar novamente até atingir seu objetivo.

Existem inúmeras maneiras para melhorar e monitorar a performance de sua aplicação. Esses exemplos são apenas para mostrar que há um caminho e que é importante se preocupar com isso.

Pense nos filtros

Em sistemas corporativos é muito comum o desenvolvimento de telas de consulta, onde cada tela terá suas especificidades e filtros diferentes, porém, existem algumas estratégias que você pode adotar para aumentar a usabilidade proporcionar uma melhor experiência ao usuário.

O que o usuário precisa no primeiro acesso?

Pense sempre em porque o usuário precisará acessar a tela e a desenvolva baseado nessa resposta. Se for uma consulta de folha de pagamento, o usuário geralmente quer acessar sua última folha de pagamento, então isso deve estar fácil para ele. A consulta, por padrão, pode vir ordenada, deixando os dados mais atuais em cima. Além disso, não é necessário trazer num primeiro momento todas as folhas de pagamento. Imagine um usuário que está trabalhando há 10 anos na empresa. Essa tela ficaria lenta para ele.

Filtros prontos

É comum a existência de consultas que são executadas com maior frequência em cada tela. Sendo assim, descubra quais são e facilite o trabalho do usuário oferecendo atalhos para essas consultas. Por exemplo: em um sistema que tem um workflow de férias onde um usuário solicita suas férias para um determinado mês, depois essa solicitação deve ser aprovada pelo coordenador, pelo gerente, pelo RH e também alguém pode reprovar e devolver para o passo anterior. Na tela que lista todas as solicitações de férias, poderia haver um filtro chamado *Pendentes comigo* para listar as solicitações que devem ser aprovadas pelo usuário logado. Isso facilitaria bastante o dia a dia do usuário, que economizaria tempo ao não precisar pesquisar entre as solicitações aquelas que dependem da sua aprovação.

Ademais, ao invés de usar filtros de data com os campos "De" e "Até", é possível definir filtros prontos como *Solicitações abertas nessa semana*, *Solicitações abertas no último mês*. No entanto, lembre-se que isso vai variar em cada situação, sendo necessário entender a tela e validar se isso se aplica.

Filtro avançado

Suponha que um usuário solicitou 10 campos novos no filtro da tela que você criou. Ao invés de colocar esses 10 campos, poluir a tela e talvez até atrapalhar os usuários iniciantes ou que já estejam acostumados com a mesma, por que não inserir um link ou botão chamado *Filtro Avançado*, onde os 10 campos solicitados (ou mais) podem ser adicionados sem prejudicar a usabilidade do sistema? Dessa forma, você mantém a tela limpa e permite que usuários avançados tenham a opção de fazer consultas mais elaboradas.

Escrever é cortar palavras

John Ruskin, escritor inglês do século XIX, contou a história de um feirante que tinha uma placa com o texto "HOJE VENDO PEIXE FRESCO". Nesta história, um homem que passava pela feira comentou com o feirante que todo dia é sempre "HOJE", então ele poderia retirar essa palavra da placa e o feirante concordou. Depois disso, o mesmo homem disse que todos na feira estavam vendendo algo, que ninguém estava dando peixes de graça, portanto, a palavra "VENDO" também era dispensável. E como nunca se vende peixe congelado em uma feira, a palavra "FRESCO" poderia ser cortada também. Por fim, restou apenas a palavra "PEIXE". Contudo, quem olha para uma barraca cheia de peixes não precisa ler a placa para saber o que era vendido ali. Assim sendo, ao final a placa foi retirada.

A partir dessa história, pense em como diminuir ou eliminar cada texto, título, label, descrição, observação e ajuda que você colocar no sistema. Seu objetivo é deixar tudo, ou o máximo possível, autoexplicativo. Quanto mais elementos você colocar na página:

- Mais ruídos e interferências a página terá;
- Menos atenção o usuário dará ao que realmente importa;
- A página poderá ficar grande e pesada.

Se você está desenvolvendo uma tela que tem um questionário para os usuários falarem sobre itens a serem melhorados na empresa, seria muito comum colocar um texto explicativo no início do questionário. Se for necessário ter esse texto, ele deve:

- Ser curto;
- Falar o essencial;
- Evitar o óbvio;
- Não ter instruções que o usuário conseguirá descobrir sozinho ao longo do questionário;

• Não mostrar mais do que ele precisa saber para iniciar o questionário (aquele parágrafo de ajuda para responder a quinta questão não precisa ser lido antes de iniciar o questionário).

Ao não se preocupar com isso, você corre o risco de os usuários nem lerem o seu texto minuciosamente redigido.

Experimente através de Wireframes

Se você não tem o hábito de criar wireframes, que são uma representação de baixa fidelidade de um design (vide **BOX 3**), você está perdendo uma oportunidade de inovar, de experimentar. Após entender o problema que você precisa resolver com a funcionalidade, ou seja, o porquê dessa funcionalidade, é interessante definir como implementá-la antes de iniciar o desenvolvimento.

Os wireframes podem auxiliar nas fases iniciais de análise para validar com a equipe e até com o usuário final o que foi pensado para cada tela. Você pode desenhar wireframes no papel ou utilizar ferramentas que podem auxiliar neste momento, como é o caso da Balsamiq e do UXPin.

BOX 3. Diferenças entre wireframe, protótipo e mockup

É muito comum confundir wireframes com protótipos e mockups e até achar que os três são a mesma coisa, pois os três são uma representação do produto final, mas têm grandes diferenças e podem ser utilizados com objetivos diferentes.

Um wireframe é uma representação de baixa fidelidade de um design. Ele mostra o quê existirá na tela, como ela estará organizada e pode descrever de forma básica como será a interação do usuário. Por não ter uma alta fidelidade, isto é, não ser idêntico à versão final da tela, o que inicialmente pode ser visto como negativo, na verdade traz a vantagem do pouco tempo e do baixo custo necessários para serem criados.

Um protótipo, por sua vez, tem uma fidelidade bem maior, parecendo muito com o produto final e simula a interface de interação com o usuário. Ele é útil para testar o sistema experimentando o conteúdo e as interações da interface. Como ponto negativo, protótipos podem ser caros e demorados para construir. Uma dica é desenvolvê-los de uma forma que possam ser reaproveitados durante o desenvolvimento do projeto.

Assim como um protótipo, um mockup é uma representação de média a alta fidelidade de um design, mas de maneira estática. Ele é muito mais parecido com o produto final do que um wireframe, porém, não tem a interação que um protótipo viabiliza. Mockups são úteis quando você quer demonstrar o produto para um stakeholder e um wireframe seria simples demais.

Para mais informações, veja o artigo "Wireframe, protótipo e mockup – Qual a diferença?" de Marcin Treder e traduzido por Ana Rute no endereço indicado na seção **Links**.

Como fazer testes de usabilidade de forma simples e barata?

Testes de usabilidade consistem na observação de uma pessoa tentando usar algo para executar tarefas comuns com o objetivo de detectar e consertar as coisas que confundiram e frustraram esta pessoa.

Depois de algum tempo trabalhando em um projeto, você achará tudo simples e fácil no seu sistema. Os testes de usabilidade mostrarão o que é simples para você, mas talvez impossível para outra pessoa. Mostrarão também que aquele botão que você tinha certeza que chamaria muita atenção pode passar despercebido.

É importante ressaltar que quanto mais técnica for uma pessoa, mais difícil pode ser para ela expressar uma ideia para alguém leigo. Os testes ajudam a garantir que pessoas que pensam de forma diferente e que têm experiências diferentes das pessoas envolvidas no projeto conseguem utilizar a tela projetada.

Os testes mostram o que funciona, o que precisa ser melhorado e o que pode ser retirado. E se os testes forem executados desde o início do projeto e realizados com frequência, os ganhos podem ser inestimáveis.

Por muito tempo acreditava-se que testes de usabilidade: eram caros, envolviam muitas pessoas, aconteciam em uma sala de vidro, tinham muitas câmeras e geravam relatórios de 50 páginas. Testes dessa dimensão existem e podem ser úteis; no entanto, é possível fazer testes simples, frequentes e econômicos.

Assim sendo, você mesmo pode organizar e realizar testes de usabilidade em apenas uma manhã. Para isso, chame um ou dois amigos ou pessoas de outro departamento da empresa, explique o que você vai fazer e peça para a pessoa acessar seu sistema e dizer o que ele entende. Solicite a ele a realização de tarefas simples como cadastrar o produto X, gerar o relatório Y e então analise a forma que a pessoa realizará estas ações e perceba suas dificuldades. Se você conseguir fazer isso com usuários reais ou pessoas do perfil dos seus usuários, melhor.

Você também pode usar um software de compartilhamento de tela (como o Camtasia) e transmitir as imagens e o som para outra sala e envolver mais pessoas da equipe para acompanhar os testes, ou apenas gravar para poder visualizar novamente em outro momento, desde que o usuário permita essa gravação.

Feito isso, defina os principais problemas encontrados e foque na correção destes itens. Depois de algumas semanas, realize um novo teste com novas pessoas e siga sempre evoluindo.

No livro "Não me faça pensar" de Steve Krug, há um capítulo sobre como realizar testes simples e com orçamento reduzido, onde são explicados todos os detalhes, como desde o recrutamento das pessoas, um roteiro a ser seguido nos testes, como escolher as tarefas para passar ao usuário, como acompanhar os testes, como priorizar o que corrigir, entre outras coisas. Nesse livro também é fornecido um roteiro para ser seguido durante os testes que pode ser baixado no site da editora (veja a seção **Links**).

Caso você queira se aprofundar mais no assunto, Krug também escreveu o livro "Simplificando coisas que parecem complicadas" e Jacob Nielsen, no final da década de 80, abordou o assunto em seu artigo "Usability Engineering at a Discount".

Metodologias de desenvolvimento influenciam na usabilidade?

Falar sobre metodologias e processos também é um assunto muito amplo, porém, verifique se sua metodologia ou processo não está impactando na qualidade do seu produto, pois um produto com erros básicos, com alterações em uma funcionalidade que impactam outras prejudicará a experiência do usuário e a

usabilidade, pois ele não conseguirá realizar seus objetivos no sistema.

Sendo assim, você deve focar em feedbacks mais rápidos dos usuários, em testes automatizados, em integração contínua, clean code, entre outras estratégias para garantir a qualidade do seu projeto.

A partir desse artigo você já pode utilizar os conceitos aqui apresentados para tomar decisões focadas no usuário. O conhecimento obtido sobre usabilidade influenciará positivamente em seus projetos e caso queira se aprofundar ainda mais, algumas referências foram citadas e lhe possibilitarão a continuação dos estudos. Vale lembrar que existem também outros assuntos e técnicas não abordadas aqui que podem ser muito úteis no seu projeto, como a definição de personas, jornada do usuário, card sorting, uso do google analytics, entre outros.

É importante lembrar que não existem receitas ou soluções prontas. Tudo que foi apresentado serve para lhe dar ferramentas, técnicas e ideias para tomar suas próprias decisões dentro do contexto que os problemas aparecerem.

E que tal, agora, você abrir o site da sua empresa e avaliar a usabilidade dele? Será que se outra pessoa entrar saberá rapidamente o que é a sua empresa, o que ela faz e quais os seus diferenciais? Pense de forma crítica em suas próximas análises e desenvolvimentos e deixe seus usuários mais satisfeitos.

Autor

Adriano Schmidt
adriano@localhost8080.com.br – www.localhost8080.com.br
Arquiteto de software na JExperts em Florianópolis, programador Java desde 2007, formado em Administração de Empresas, fazendo MBA em Marketing Digital na FGV e é ator de teatro como hobby. Além disso criou o projeto "O Usuário está bêbado" para testar sites e sistemas de uma perspectiva de Usabilidade e UX após beber muita cerveja: www.localhost8080.com.br/o-usuario-esta-bebado.

Links:

BASTIEN, J.M. Christian., SCAPIN, Dominique L.. Ergonomic criteria for the evaluation of humam-computer interfaces. Rapport technique de l'INRIA. 1993.

KRUG, Steve. Não me faça pensar. Alta Books Editora, Rio de Janeiro, 2014.

NIELSEN, J. Usability Engineering. California: Morgan Kaufmann, 1993.

NIELSEN, J.; LORANGER, H. Prioritizing Web Usability. California: New Riders, California, 2006.

ROSENFELD, L.; MORVILLE, P. Information Architecture for the Word Wide Web. 3. ed. Sebastopol, CA: O'Reilly, 2006.

TEZZA, Rafael. Proposta de um constructo para medir usabilidade em sites de e-commerce utilizando a Teoria de Resposta ao Item. Dissertação (Mestrado) – UFSC. Florianópolis, 2009.

Browser Diet.
http://browserdiet.com/pt/

ISO 9241. Ergonomic requirements for office work with visual display terminals.
http://www.inf.ufsc.br/~cybis/pg2003/iso9241-11F2.pdf

Script para testes de usabilidade.
http://www.altabooks.com.br/index.php?dispatch=attachments.getfile&attachment_id=1955

Wireframe, protótipo e mockup – Qual a diferença?
http://anarute.com/wireframe-prototipo-e-mockup-qual-a-diferenca/

10 princípios de UX do Google – Gustavo Moura. Palestra no IXDSA11
https://www.youtube.com/watch?v=rrn1P-U8kMY

Você gostou deste artigo?

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

Ajude-nos a manter a qualidade da revista!

Desvendando o processo de Teste de Software

Um guia para planejar testes assertivos, com cobertura adequada e que realmente agreguem valor ao projeto

Fique por dentro

Este artigo tem como objetivo servir como um guia para o planejamento da fase de testes de um projeto. Para isso, serão analisados os artefatos que devem ser produzidos, atividades e definições essenciais do projeto para que os testes sejam assertivos, tenham a cobertura adequada e realmente agreguem valor ao projeto. Como público alvo tem-se os coordenadores de desenvolvimento de software, gerentes de projetos, analistas de negócio e, principalmente, desenvolvedores, testadores e analistas de teste de software.

Em um projeto de software é natural que os envolvidos foquem no resultado final, esperando que a solução funcione corretamente. Porém, antes de pensarmos no que seria uma solução funcionando corretamente, sugerimos primeiramente que você pense no contexto de negócio onde o sistema está envolvido. Por questões estratégicas, agilidade na tomada de decisões, redução de custos, entre outras características, grande parte dos sistemas estão integrados, o que aumenta a complexidade e consequentemente a necessidade de testes mais robustos para manter a qualidade.

Com base nisso, chegamos a um segundo pensamento que se faz necessário, apesar de já estar implícito: as soluções precisam funcionar, do ponto de vista técnico e de negócio, ou seja, não podemos nos preocupar apenas se o sistema está funcionando sem erros (visão técnica). Devemos levar em consideração também se ele está atendendo ao propósito para o qual foi desenvolvido (visão de negócio). No entanto, durante um projeto de desenvolvimento de software, muitas vezes estas visões estão separadas, e é até natural que estejam, pois costumam ser exercidas por áreas diferentes. Quando é indicado que um software apresenta problemas, não estamos nos referindo apenas à indisponibilidade de um site, lentidão do sistema ou a outra característica técnica. Podemos nos referir também ao fato de o software expor uma característica funcional/de negócio que não está correta.

Este é mais um ponto de atenção que é preciso ter para que no final do projeto possamos afirmar que o software possui qualidade. Antes de continuarmos, vale lembrar que erros, defeitos e falhas são conceitos distintos, porém, para simplificar o entendimento, iremos tratar como uma única definição: "inconformidade do software".

São pelas razões mencionadas que existem os Testes de Software e as equipes de teste e qualidade. Com base no que foi exposto, ao adotar os testes e garantir uma visão unificada (técnica e funcional) do software, dificilmente os objetivos e resultados esperados da solução não serão alcançados.

Portanto, o objetivo deste artigo é ressaltar a necessidade dos testes e discorrer sobre conceitos importantes, relacionados à fase de planejamento, a qual representa a base para as demais etapas (análise, modelagem, implementação, etc.).

Como um exercício para os desenvolvedores que não gostam de testar ou que acham que a etapa de testes deve ser menor ou praticamente não existir, sugerimos que reflitam sobre os impactos que podem ocorrer quando um software falha. Independentemente da área de seguimento, podemos listar alguns problemas em comum:

- Prejuízos financeiros;
- Perda de credibilidade;
- Perde de qualidade e retrabalho;
- Perda de confiança;
- Ações legais.

A partir desses itens, podemos perceber o quanto os testes precisam ser levados a sério e colocados em um patamar de discussão onde a redução ou eliminação da fase de testes para atingir o prazo e custo desejados seja uma possibilidade.

Para desmitificar a questão de que é possível ter economia de custos ao reduzir a fase de teste, apresentamos nas **Figuras 1** e **2** um cenário comum que podemos nos deparar por conta de retrabalho. Note que o cenário de propagação de erros por má/não execução de testes pode gerar custos e retrabalhos maiores ao longo do projeto, os quais certamente superariam os custos de uma fase de testes bem planejada e executada.

Como esperado, este cenário não é aceitável em um projeto, ainda mais quando lidamos com soluções corporativas que exigem precisão cirúrgica de custos e prazos. Portanto, os testes de software não podem ser encarados apenas como mais uma fase do projeto, já que ela pode ser crucial para o sucesso ou fracasso do mesmo. Ao contrário do que muitos pensam, os testes começam muito antes da "simples" execução de testes em cima do código desenvolvido.

Talvez a seguinte definição nos ajude a ampliar um pouco a nossa visão sobre os testes: "Teste é o processo que consiste em todas as atividades do ciclo de vida, tanto estáticas quanto dinâmicas, voltadas para o planejamento, preparação e avaliação de produtos de software e produto de trabalhos relacionados, a fim de determinar se elas satisfazem os requisitos especifi-

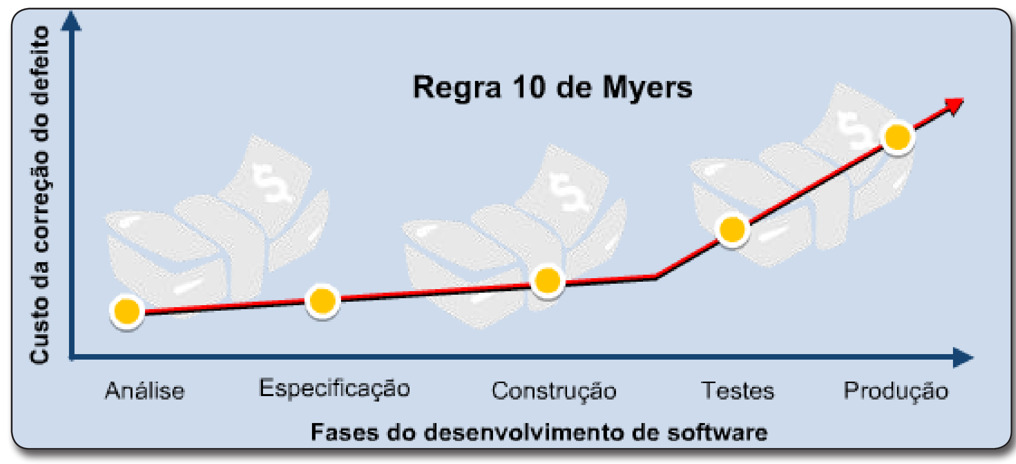


Figura 1. Fases de desenvolvimento de software x custo de correção

Nos requisitos	Na especificação	No código	Durante a execução do teste	Durante o teste de aceitação	Com o software em produção
- Alterar documento de requisitos.	- Alterar documento de requisitos. - Alterar a especificação.	- Alterar documento de requisitos. - Alterar a especificação. - Alterar o Código.	- Alterar documento de requisitos. - Alterar a especificação. - Alterar o Código. - Alterar o teste, reteste e teste de regressão.	- Alterar documento de requisitos. - Alterar a especificação. - Alterar o Código. - Alterar o teste, reteste e teste de regressão. - Refazer teste de aceitação.	- Alterar documento de requisitos. - Alterar a especificação. - Alterar o Código. - Alterar o teste, reteste e teste de regressão. - Refazer teste de aceitação. - Avisar usuários da falha e da solução de contorno. - O banco de dados do cliente pode necessitar de correções. - Montar um novo build do software pode levar horas.

Aumento do trabalho (Re-trabalho)

Figura 2. Atividades a serem realizadas nas fases do projeto diante dos erros encontrados

cados e demonstrar que estão aptas para sua finalidade e para a detecção de defeitos." (Glossário de termos do ISQB).

Pensando um pouco em projetos de software problemáticos (que falham), quais seriam os motivos que levam à falha? A seguir são indicados alguns motivos:

• Falta de definição de requisitos;
• Alteração de escopo de projeto não documentada;
• Falha de comunicação;
• Pouco tempo para testar, diminuindo assim a cobertura dos testes;
• Processo de desenvolvimento/teste imaturo.

Para aqueles que ainda não estão muito familiarizados com testes, saiba que os mesmos vão muito além dos testes dinâmicos, contribuindo e muito para definições de requisitos e escopo do projeto, o que é alcançado com os testes estáticos. Por exemplo: ao realizar inspeções ou revisões de documentações já produzidas, podem ser encontrados gaps no escopo do projeto. A **Figura 3** expõe uma rápida definição de testes dinâmicos e estáticos.

Princípios gerais dos testes

Ainda seguindo uma linha de contextualização para os leitores que estão começando a se envolver com o mundo dos testes, achamos importante destacar os sete princípios-chave do teste de software. São eles:

1. Teste demonstra a presença de defeitos: A equipe de teste precisa ter em mente que os testes não são realizados para provar que o software funciona, mas sim que o software não funciona. Um bom teste possui uma alta probabilidade de encontrar uma falha. Neste ponto vale ressaltar que mesmo que se tenha uma grande cobertura de testes, não é possível garantir que o software está 100% isento de erros;

2. Teste exaustivo é impossível: É importante aceitar que não é possível testar tudo. Diante disso, a melhor opção é garantir que sejam realizados testes nos trechos de código que oferecem mais riscos para o negócio;

3. Teste antecipado: Traz a ideia de iniciarmos os testes o quanto antes, considerando o ciclo de desenvolvimento de software. Um bom exemplo neste caso é a execução de testes estáticos – sabendo que os mesmos normalmente ocorrem na fase de análise do desenvolvimento – a partir do momento em que os requisitos funcionais já estiverem definidos;

4. Agrupamento de defeitos: Os defeitos sempre apresentam uma característica e a partir desta podemos classificá-los;

5. Paradoxo do pesticida: É recomendado criar uma quantidade variada de testes para um "alvo" que será testado, pois ao passo que os erros são corrigidos, alguns casos de teste deixarão de ser tão eficientes;

6. Teste depende do contexto: Os testes dependem de uma série de avaliações, por exemplo: riscos, orçamento, grau de cobertura

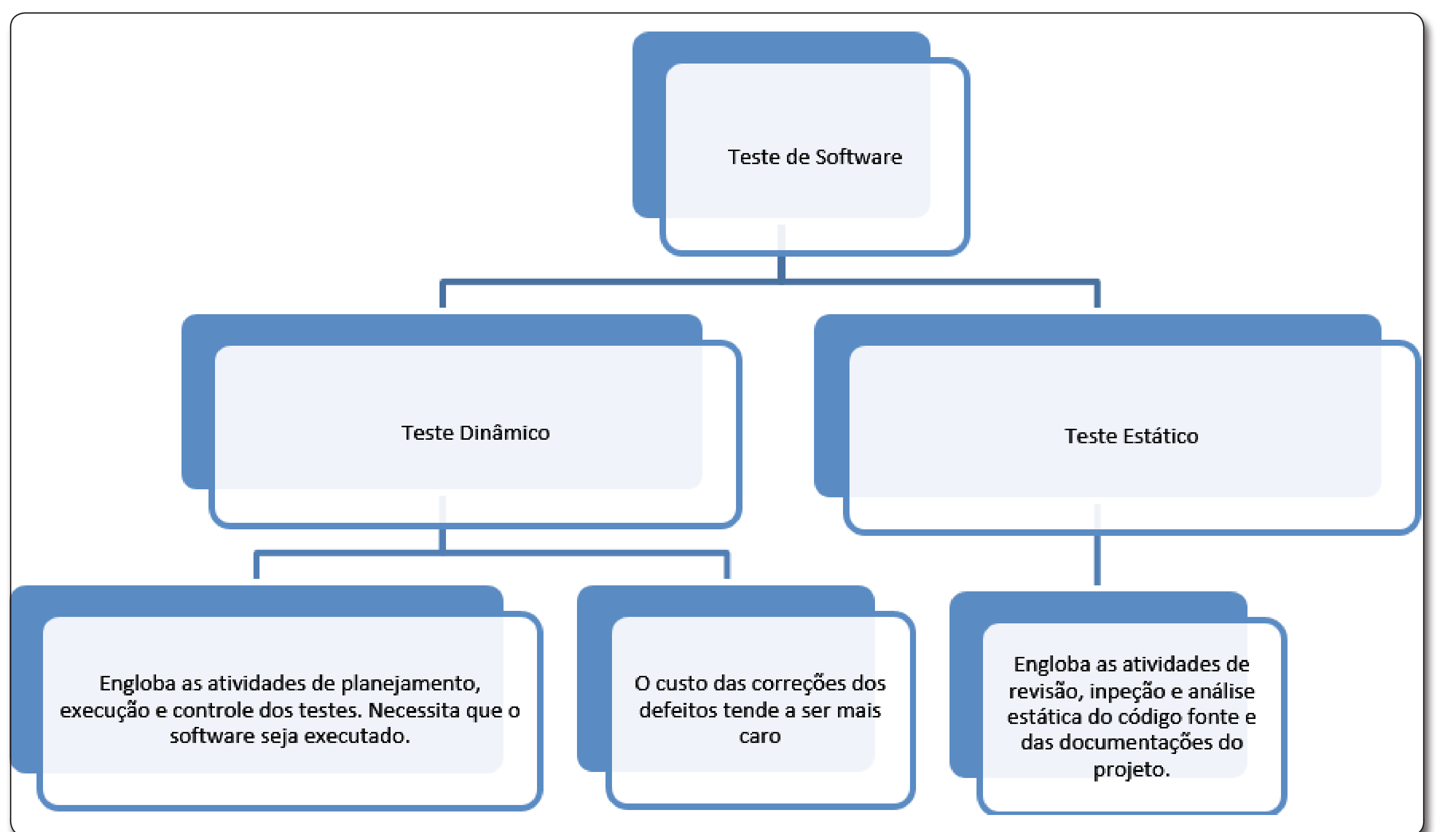


Figura 3. Testes dinâmicos e estáticos

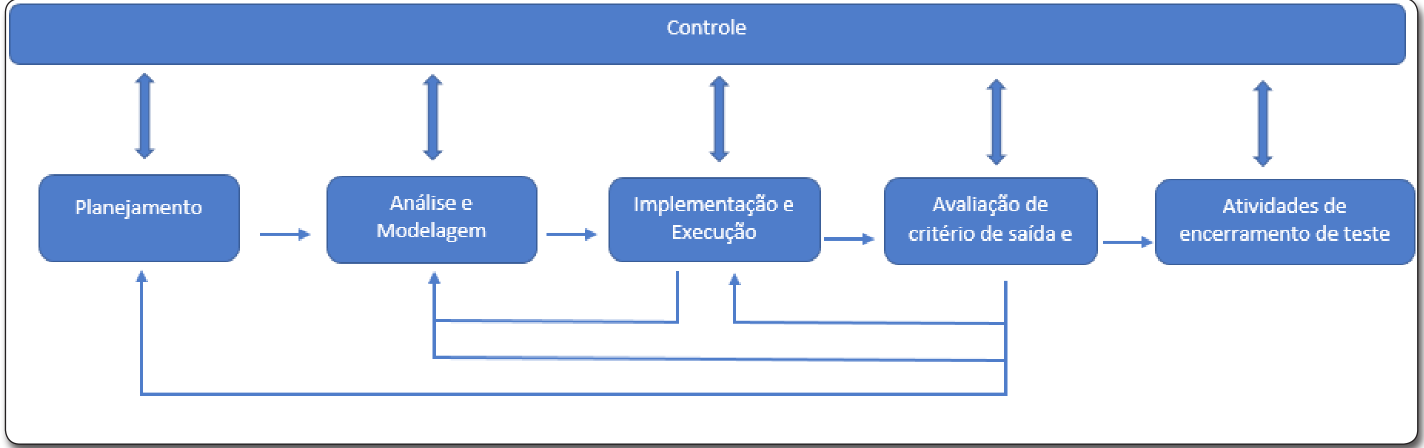


Figura 4. Fases do processo de teste

de testes, seguimento do negócio, entre outros fatores. Ex.: o teste de um sistema bancário será muito mais rigoroso se comparado ao teste de um sistema de empréstimo de livros;

7. A ilusão da ausência de erros: Podemos dividir este princípio em dois pontos para facilitar o entendimento:

a. Erros do sistema: realizar a execução de todos os testes não garante que o software esteja 100% sem erros. Isso porque, como vimos anteriormente, o teste exaustivo é impossível. O software pode estar executando normalmente, mas não quer dizer que internamente não exista um erro esperando para ser encontrado;

b. Erros do negócio: não ter erros que se apresentam tecnicamente não quer dizer que não existam erros relacionados às regras de negócio. Conforme mencionamos anteriormente, a qualidade depende da boa execução técnica e funcional do software. Só assim podemos dizer que o produto atinge as expectativas do cliente.

O Processo de Teste

É de grande valia que as pessoas e organizações assimilem a importância dos testes e saibam que estes não são uma simples fase de projeto devendo, portanto, serem encarados como um subprojeto com processo bem definido. A **Figura 4** nos ajuda a ter uma visão global do processo de teste e sua interação durante as fases de planejamento, análise, modelagem, implementação, etc.

Ao longo do artigo teremos uma explicação superficial de cada uma das fases deste processo, porém sempre com uma perspectiva da fase de planejamento.

Partindo do ponto inicial, exatamente a fase de planejamento, é preciso conhecer algumas definições importantes, também apresentadas no artigo, para montar um plano de testes consistente. A **Figura 5** mostra algumas atividades que precisarão ser consideradas dentro das fases que serão planejadas, já destacadas na **Figura 4**.

Um detalhamento mais específico das demais fases e atividades não será realizado, porém estarão cronologicamente destacados para o entendimento de tudo o que precisa ser considerado em um planejamento consistente de teste de software.

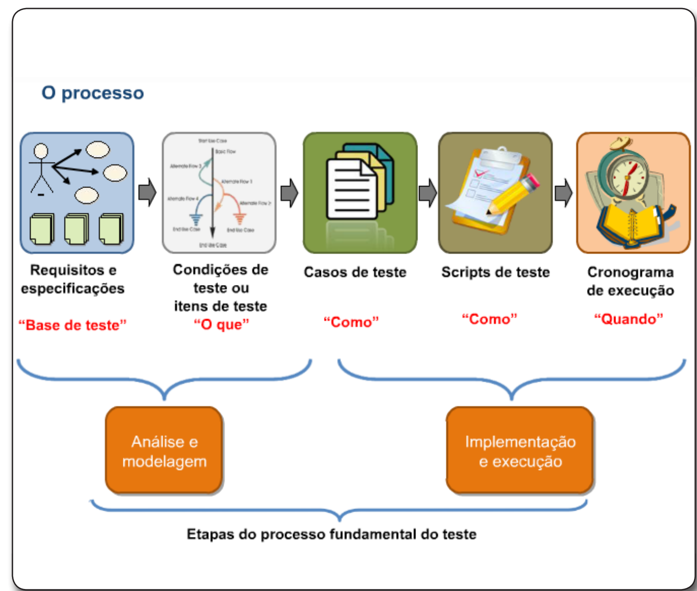


Figura 5. Etapas do processo de teste

Teste independente de desenvolvimento

Uma característica importante do processo de testes é que o mesmo é totalmente independente do modelo de desenvolvimento que está sendo considerado para o projeto. Desta forma, sua única preocupação no planejamento é saber como fazer a adaptação ao processo de desenvolvimento, ou seja, conhecê-lo para saber onde suas atividades ocorrerão. Na **Figura 6** apresentamos o processo de desenvolvimento em paralelo ao processo de testes, bem como suas interações.

É importante observar que algumas saídas das fases de desenvolvimento são o gatilho para iniciar algumas fases de teste. Entendendo este relacionamento, é possível adaptar o processo de teste a qualquer modelo de desenvolvimento, já que os mesmos ocorrem em paralelo.

O projeto de testes para o projeto de desenvolvimento

Para ter um direcionamento sobre os testes em um primeiro instante, é aconselhável que o gerente/coordenador/líder da equipe de testes busque entender quais são as estratégias e políticas de teste adotadas pela empresa. Vale a pena lembrar que a política de testes influencia diretamente na forma como o planejamento será feito, uma vez que esta define os processos e artefatos básicos considerados importantes para a empresa.

A partir da política de testes as organizações especificam um conjunto básico (mínimo) de procedimentos e atividades que precisam ser realizadas durante os testes, conhecido como "Plano de teste mestre", que servirá para os demais níveis de testes que serão utilizados.

Na **Figura 7** ilustramos esta divisão hierárquica entre a política de teste e os planos de teste.

Após o alinhamento com a política de teste da organização, podemos fazer a definição do "Plano de teste mestre" do projeto,

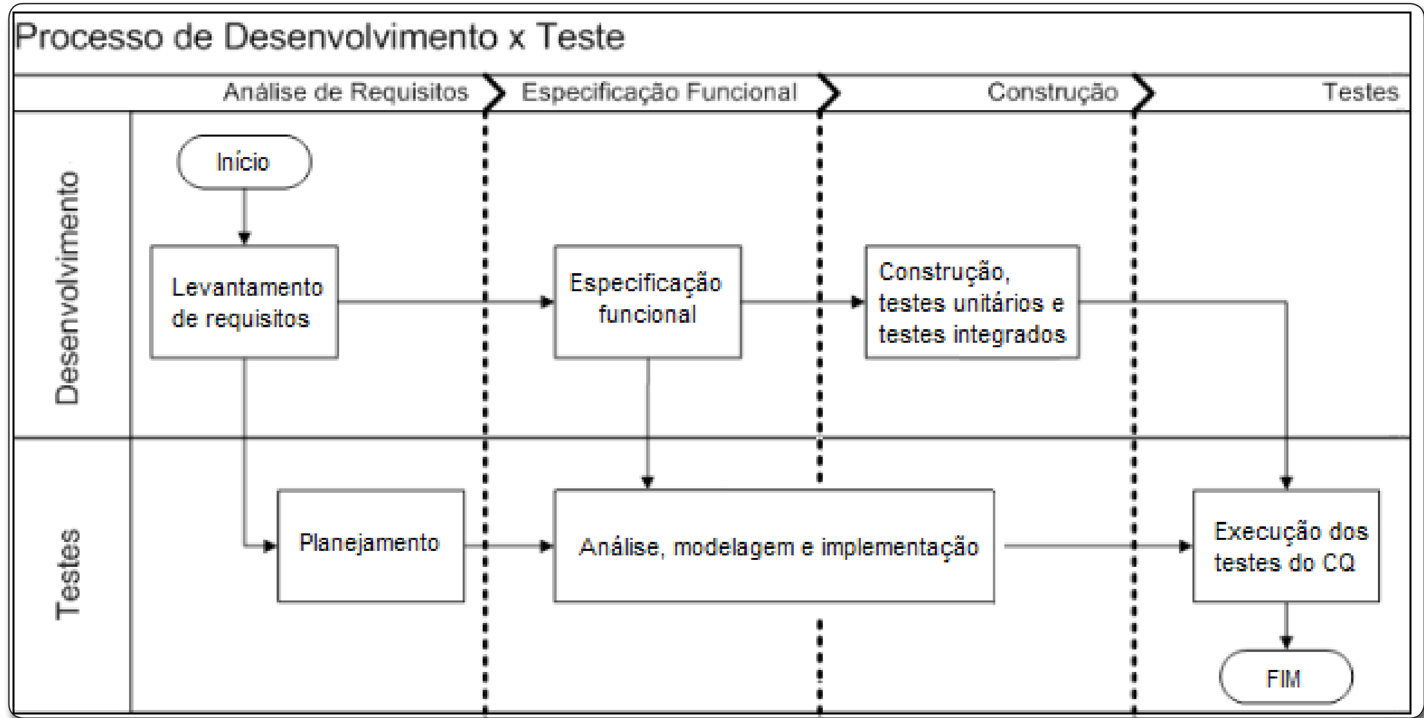


Figura 6. Processo de desenvolvimento em paralelo ao processo de teste

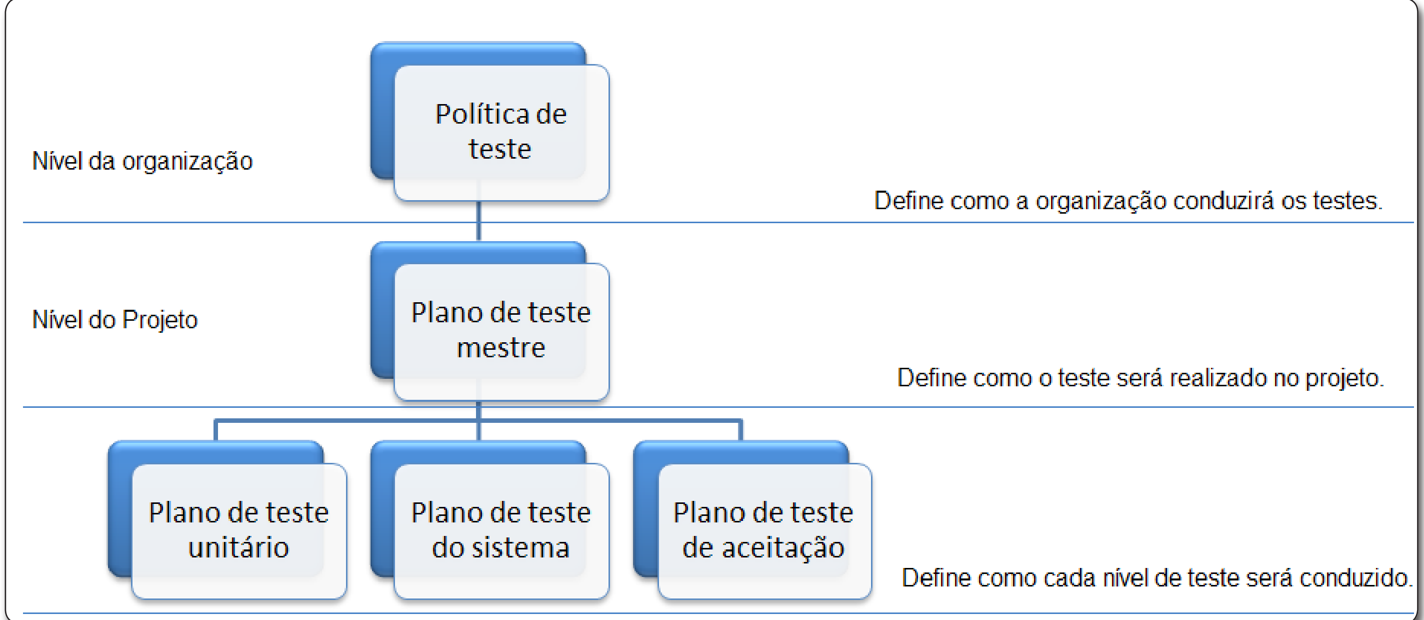


Figura 7. Estratégias e plano de teste

que contém basicamente todo o escopo de testes que se espera realizar. Lembre-se que este plano mestre poderá servir como referência para cada um dos níveis de teste que serão realizados ao longo do projeto.

Atualmente existe uma estrutura de Plano de Testes proposta pela norma IEEE 829 que é responsável pela definição de normas e padrões para o processo de teste e qualidade de produto. Também conhecida como Padrão 829, esta pode ser considerada como um template para as equipes de teste.

Como o plano de teste é a principal saída da fase de planejamento, a seguir listamos alguns dos pontos de maior relevância da estrutura da norma IEEE 829:

1. Identificação do plano de teste: Representa o identificador para o plano de testes, podendo ser um número, nome ou qualquer informação utilizada dentro da organização;

2. Introdução: A introdução pode trazer uma visão geral sobre o projeto como um todo, bem como o escopo do projeto de testes, destacando o que se espera com os testes, resumo dos requisitos, entre outros detalhes que se considere importante para uma introdução adequada ao projeto;

3. Itens de teste: Neste tópico devem ser destacados os itens do software que serão testados, incluindo também documentos que serão utilizados nos testes estáticos. Vale destacar que itens de software englobam um escopo geral, por exemplo: banco de dados, integrações, desempenho, etc.

4. Funcionalidades a serem testadas: Lista de todas as funcionalidades que serão testadas;

5. Funcionalidades que não serão testadas: Lista de todas as funcionalidades que não serão testadas. É importante destacar estes pontos, pois aqui está sendo fechado o escopo do projeto. As funcionalidades que serão ou não testadas nos ajudam a determinar os recursos requeridos para o projeto;

6. Abordagem: Este tópico descreve a abordagem que iremos adotar para os testes, podendo ser:

- **a. Analítica:** testes direcionados para as funcionalidades mais críticas para o sistema ou negócio;
- **b. Baseada em modelos:** os testes são realizados a partir de informações estatísticas sobre as ocorrências de erro, ou seja, se baseia em "lições aprendidas" com incidentes para direcionar os testes;
- **c. Abordagem metódica:** parecida com a estratégia "Baseada em modelos" (baseada em falhas), porém também leva em consideração alguns checklists e características de qualidade mais abrangentes;
- **d. Compatível com processos e padrões:** utiliza padrões de mercado para realização de teste como, por exemplo: metodologias ágeis para teste;
- **e. Dinâmica e heurística:** considera a experiência dos testadores e tem um caráter mais exploratório, levando

em conta que não se precisa ficar tão preso a casos de testes pré-planejados. Seria basicamente o "sair testando";

f. Baseada em conselhos: testes direcionados por conselhos de especialistas técnicos ou de negócio;

g. Regressão: considerar esta abordagem para os casos de correção e manutenção do sistema. Outro ponto que pode ser levado em conta junto com a *Regressão* é o *Re-Teste*.

7. Critérios de aprovação e reprovação: Critérios que definem se os itens testados passaram ou não no teste, por exemplo: número máximo de falhas encontrados, % de falhas encontradas, etc.;

8. Critérios de parada e retomada do teste: Define critérios que causariam a suspensão/retomada do teste, por exemplo: indisponibilidade do ambiente de testes do cliente e aprovação do plano de teste para início dos mesmos;

9. Entregáveis do teste: São as documentações que precisarão ser entregues, por exemplo: casos de teste, plano de testes, modelagem, dados (massa de teste), laudo (resultado), entre outros artefatos que sejam considerados como necessários durante o projeto;

10. Atividades do teste: Definição das atividades de responsabilidade da equipe de teste, por exemplo: definição dos casos de testes, cenários de teste, etc.;

11. Ambientes de teste: Expõe as configurações necessárias para o ambiente de testes, como ferramentas, características da máquina (espaço, desempenho, etc.), entre outras;

12. Responsabilidades: Neste tópico tem-se a definição de papéis e responsabilidades dentro do projeto de teste;

13. Recursos e treinamentos necessários: Aqui são especificados basicamente os recursos e conhecimentos necessários;

14. Cronograma: Define o plano de trabalho da equipe de testes;

15. Riscos e contingências: Este item tem como objetivo fazer a gestão de riscos dentro de qualquer projeto, o que nos ajuda a prevenir situações indesejáveis. Em um projeto de testes não seria diferente;

16. Aprovação: Registro da aprovação do plano de testes por parte dos responsáveis, que poderiam ser: o gerente da equipe de testes, gerente de projeto e até mesmo o próprio cliente (caso seja um plano compartilhado).

Com isso, concluímos a criação do plano de testes para termos uma visão do escopo de atuação do projeto de teste dentro do projeto de desenvolvimento. Outro fator importante a ser considerado no planejamento é a definição dos níveis de testes que serão aplicados durante o projeto.

Níveis de Testes

A busca pela qualidade de software está associada à aplicação de todos os níveis de testes, visto que estes possuem características e objetivos distintos. A **Figura 8** apresenta os quatro níveis de testes, analisados a seguir.

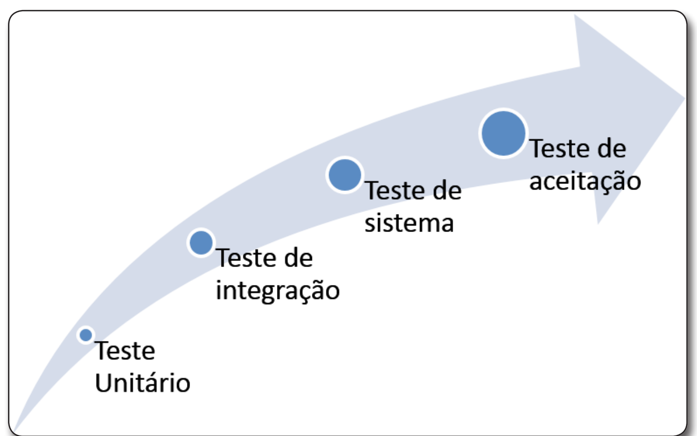


Figura 8. Níveis de teste

Teste Unitário

O teste unitário tem por objetivo validar – individualmente – itens menores da solução. Corresponde ao primeiro nível de teste e deve ser executado durante e após o termino da fase de codificação.

Este nível tem como objetivo garantir que o item a ser testado cumpra a função para a qual ele foi projetado e encontrar erros de lógica de programação.

Os testes unitários trazem como benefícios:

- Garantir que problemas sejam descobertos ainda durante o processo de desenvolvimento;
- Facilitar a manutenção de código, especialmente a refatoração;
- Simplificar a integração de módulos;
- Servir como artefato do projeto, apresentando os detalhes da funcionalidade.

Uma técnica de desenvolvimento de software muito conhecida em projetos que adotam metodologias ágeis e que está associada aos testes unitários é o TDD (*Test Driven Development*). Esta técnica prioriza a criação e automação dos testes e posteriormente a implementação do código fonte. É uma solução que possui uma abordagem essencialmente iterativa e baseada em ciclos de elaboração de casos de teste. Na **Figura 9** podemos ver as fases do TDD:

- Primeiro escreva um teste que falhe;
- Depois escreva um código que faça o teste passar;
- Melhore o código escrito;

Teste de Integração

Testes de integração têm o objetivo de encontrar erros na integração dos diferentes componentes de software. Consiste em testar a solução como um todo, observando o comportamento de cada um dos componentes e suas interações. Em relação à integração de componentes, há duas vertentes principais para o desenvolvimento de software: *bottom-up* e *top-down*.

Na abordagem *bottom-up* o programa é desenvolvido a partir de rotinas básicas que prestam serviços a rotinas de nível mais alto.

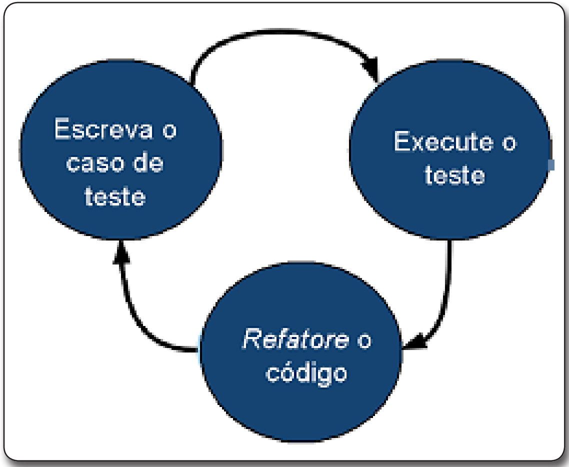


Figura 9. Fases do TDD

Por sua vez, na abordagem *top-down* faz-se o inverso. Para isso, o programador trabalha supondo que o código de baixo nível já esteja pronto.

Testes de integração acontecem de forma recursiva e geralmente são executados pelo desenvolvedor. Seguem os tipos de testes de integração existentes:

- **Teste de integração de componentes:** Testes associados à integração dos componentes de um sistema;
- **Teste de integração de sistemas:** Testes associados à integração entre sistemas pertencente à solução.

Teste de Sistema

O teste de sistema, também conhecido por teste de validação, é a fase em que o sistema, já completamente integrado, é verificado quanto a seus requisitos em um ambiente controlado que apresente características similares a um ambiente de produção. Está no escopo da técnica de teste de caixa-preta, e dessa forma não requer conhecimento da estrutura interna do sistema. É um teste mais limitado em relação aos testes de unidade e de integração, pois se preocupa somente com aspectos gerais do sistema [QATEST].

Este tipo de teste não se limita a testar somente requisitos funcionais, mas também não funcionais, como a expectativa do cliente, e por isso inclui técnicas não funcionais de teste.

Teste de Integração de Sistema

O teste integrado de sistema é o tipo de teste executado para garantir que o sistema funcione como um todo, ou seja, valida que a sua integração com outras interfaces esteja funcionando conforme o esperado.

Esta etapa, também conhecida como SIT (*System Integration Testing*), é a etapa do processo de testes que acontece antes da homologação, ou seja, onde o analista/testador encontra o maior número de defeitos possíveis, para então enviá-los para correção. Após os defeitos corrigidos é feita uma tarefa de reteste, para garantir que além das correções efetuadas não houve a introdução de novos defeitos no código. O teste integrado de sistema tem a finalidade de não permitir que erros sejam propagados para as etapas de homologação e produção [CEVIU].

Teste de Aceitação

O teste de aceitação é um tipo de teste associado às necessidades dos usuários e aos requisitos e processos de negócios. É realizado para verificar se o sistema satisfaz os critérios de aceite e para que o cliente tenha as informações e decida por aprovar o sistema. O objetivo deste teste é estabelecer a confiança no sistema.

Neste nível apenas algumas funcionalidades chaves são testadas pelo cliente e/ou usuários.

Existem três formas de teste de aceitação:

- **Teste de aceitação de usuário (UAT):** Corresponde a um teste de caixa-preta realizado por um grupo restrito de usuários finais antes da disponibilização do sistema. Tem por função verificar se o sistema atende aos seus requisitos originais e às necessidades do usuário;
- **Teste de aceitação operacional (OAT):** Também conhecido como teste de preparação. Procura garantir que os processos e procedimentos estejam prontos para que o usuário/ testador possa utilizá-los;
- **Alfa-teste e Beta-teste:** Alfa-teste é realizado pelo cliente no ambiente de desenvolvimento no período entre o término do desenvolvimento e a entrega. Beta-teste é realizado pelo usuário final nas instalações do cliente após o alfa-teste.

Teste de Mudanças

Abrange as mudanças identificadas durante o processo de implementação e implantação do sistema. Há dois tipos de testes relacionados há mudanças:

- **Reteste:** Teste que executa casos de teste reprovados durante sua última execução. Este procedimento é feito para verificar o sucesso das ações corretivas;
- **Teste de regressão:** Teste realizado em componentes do software que não sofreram alterações, mas que podem ter sido impactados pela manutenção realizada em outros módulos.

Tipos de Testes

Os tipos de testes escolhidos dependem das características e objetivos do projeto. Tem-se como principais tipos de testes os testes funcionais, não funcionais, estruturais e de regressão, analisados a seguir.

Testes funcionais

O teste funcional, também conhecido como teste de caixa-preta, analisa o item a ser testado sob o aspecto funcional, observando qual foi o resultado do teste e não como o mesmo foi obtido.

Tem como objetivo verificar o processo de *input* dos dados, o processamento e sua resposta, além da implementação apropriada das regras de negócio.

A execução dos testes funcionais ocorre em um ambiente separado e deverá seguir os procedimentos definidos nos casos de teste.

Testes não funcionais

Os testes não funcionais exploram como o sistema trabalha e podem ser realizados em todas as fases do projeto. Eles contemplam um conjunto de testes que procura garantir a operação correta do sistema. Seguem alguns tipos de testes não funcionais:

- Teste de performance;
- Teste de carga ou estresse;
- Teste de usabilidade;
- Teste de interoperabilidade;
- Teste de manutenibilidade;
- Testes de confiabilidade;
- Testes de portabilidade.

Testes estruturais

Testes estruturais, também conhecidos como testes de "caixa branca", têm o objetivo de analisar a estrutura interna e o comportamento do componente a ser testado, necessitando que o desenvolvedor entenda como o componente foi codificado. Pode ser executado em todos os níveis de testes.

Com esta opção é possível quantificar quanto do código foi testado (isto é, a cobertura de teste). Existem dois tipos de testes estruturais [GALEOTE]:

- **Teste estrutural de debug:** Tem como objetivo encontrar a causa raiz de um defeito e entender o comportamento de uma aplicação através da execução passo a passo;
- **Teste estrutural de execução:** Corresponde à execução do código e à análise dos dados coletados com o propósito de encontrar erros e melhorar a qualidade do mesmo.

Testes de regressão

Consiste em aplicar a cada nova versão do software ou a cada ciclo todos os testes que foram aplicados anteriormente ao sistema.

É recomendada a utilização de ferramentas de automação para aumentar a produtividade e viabilidade desse tipo de teste. Assim a nova versão ou ciclo de teste poderá ter todos os testes anteriores reexecutados com maior agilidade.

Ferramentas de teste

Apesar de existir um grande conjunto de ferramentas de teste, o grande desafio é aplicar a automação de teste dentro do ciclo de vida do software e identificar o quanto esta automação ajudará a reduzir custos e melhorar a produtividade dos profissionais envolvidos.

O processo de adoção e execução de ferramentas de teste contempla as seguintes atividades:

1. Avaliar junto ao mercado uma ferramenta que possa atender as necessidades do projeto;
2. Conhecer as características do ambiente de teste e prepará-lo para suportar as características da ferramenta de teste;
3. Configurar a ferramenta de teste segundo as características de seu projeto.

Seguem alguns tipos e nomes de ferramentas de teste conhecidos no mercado [TRUST]:

• Ferramenta de gestão de teste: TestLink;
• Ferramenta para automação de testes funcionais: Badboy;
• Ferramenta de gestão de defeitos: Mantis;
• Ferramenta para testes de performance, carga e stress: WebLOAD.

Planejando o processo de teste

Conforme indicado na **Figura 4**, o processo de teste pode ser dividido em etapas, sendo a primeira a base para todas as outras. Deste modo, ela é responsável por entender quais são as metas, riscos e escopo do projeto e, principalmente, gerar um planejamento para todo o projeto de teste. Nos próximos tópicos, o artigo detalhará as fases posteriores à etapa de planejamento.

Análise e Modelagem

O principal objetivo da etapa de análise e modelagem de teste é exercitar as condições de uso do software de uma maneira eficiente, alcançando o máximo de cobertura com o mínimo de casos de teste.

As fases de análise e modelagem estão bastante associadas, sendo frequentemente executadas ao mesmo tempo. Divididas por uma linha tênue, poderíamos diferenciá-las da seguinte forma: a análise está voltada para o escopo do projeto como um todo, enquanto a modelagem está focada nas atividades do projeto de testes.

A fase de análise/modelagem considera as seguintes atividades:

• **Definir as condições/restrições do teste;**
• **Desenhar os casos de teste;**
• **Definir o script de teste;**
• **Definir o cronograma de execução dos testes.**

Implementação

Nesta etapa serão utilizados os artefatos criados na fase de análise e modelagem para construção dos cenários de teste. Seguem as principais atividades da implementação:

• Criar os casos de testes;
• Priorizar os casos de teste para sua execução, ou seja, privilegiar os casos de teste com maior impacto;
• Criar a massa de dados;
• Criar os procedimentos para os testes;
• Escrever os scripts automatizados dos testes (se for o caso);
• Criar as suítes de testes a partir dos casos de teste para eficiente execução;
• Verificar se o ambiente de teste foi configurado corretamente.

Execução

Esta fase contempla a execução dos casos de teste definidos nas fases anteriores. As suas principais atividades são:

• Executar os testes (manualmente e de forma automatizada);
• Avaliar os critérios de sucesso de cada teste (parâmetros de entrada e saída);
• Evidenciar resultados e características;
• Comparar os resultados esperados;
• Reportar defeitos e enviá-los para os responsáveis pela correção.

Durante esta fase é fundamental que todos os envolvidos no projeto, como analistas/desenvolvedores, analistas de testes, tes-

tadores, gerente de projeto, cliente, entre outros, estejam cientes do status da execução.

Avaliação dos critérios de saída

Durante o planejamento dos testes é importante que sejam definidos os critérios de saída, também conhecidos como critérios de aceite, pois estes representam a base para análise de sucesso dos testes. Seguem alguns exemplos de critérios de saída:
• Percentual de casos de testes executados com sucesso;
• Número de erros que ainda não foram corrigidos;
• Número de erros de acordo com sua criticidade.

Atividades de encerramento

Assim como um projeto de desenvolvimento, um projeto de teste também possui atividades associadas à conclusão dos testes. Seguem alguns itens a serem abordados neste momento:
• Análise de lições aprendidas;
• Avaliar se todos os artefatos planejados foram entregues;
• Atualização dos artefatos;
• Análise do laudo/relatório de testes.

As atividades de encerramento de testes desempenham um papel de grande importância, já que poderão contribuir para a melhoria contínua dos processos e projetos de teste.

Conhecendo a Certificação CTFL

Caso o leitor deseje ter algum diferencial na área de teste de software, a certificação CTFL (*Certified Tester Foundation Level*) é uma ótima opção. Com um interessante custo benefício, ela possibilita um reconhecimento muito significativo do mercado e comunidade devido ao seu conteúdo abrangente. É organizada pela ISEB (*Information Systems Examination Board*) e ISTQB (*International Software Testing Qualifications Board*) e pode ser realizada no idioma inglês em qualquer centro de testes PROMETRIC disponível no Brasil, com custo de US$ 232,00. O exame exige que o candidato tenha 65% de assertividade para 40 questões de múltipla escolha.

Vale ressaltar que as empresas estão valorizando cada vez mais os profissionais que atuam diretamente com o processo de obtenção de qualidade, pois esta área está relacionada diretamente ao custo de retrabalho, imagem perante o mercado e satisfação do cliente, por exemplo. Levando isso em consideração, podemos chegar à conclusão que todos os envolvidos em um projeto deveriam estar preocupados com a qualidade do produto e consequentemente se candidatar ao processo de certificação.

A certificação visa garantir que os seguintes fundamentos do processo de teste de software sejam compreendidos [BSTQB]:
• Usar uma linguagem comum para uma comunicação eficiente e eficaz com todos os testadores e participantes do projeto de teste;
• Compreender os conceitos e abordagens de teste;
• Projetar e priorizar testes usando técnicas estabelecidas e analisar as especificações funcionais e não-funcionais em todos os níveis de teste;
• Executar testes de acordo com os planos de teste aprovados e analisar e informar sobre os resultados obtidos;
• Escrever relatórios de incidentes claros e compreensíveis;
• Participar de revisões de projetos;
• Estar familiarizado com os diferentes tipos de ferramentas de teste.

É importante citar ainda que esta certificação não será a única responsável por trazer o sucesso profissional, mas com certeza fornecerá uma boa base para que o profissional consiga se destacar.

Copa do Mundo de Teste de Software

Um diferencial curioso e estimulante da área de testes é a Copa do Mundo de Testes (*Software Testing World Cup*), organizada anualmente por *Díaz & Hilterscheid*. O evento tem o objetivo de promover e divulgar a cultura da necessidade dos testes em um projeto, além de premiar os mais bem classificados. A fase final da última edição, realizada em 2014 na cidade de Berlin-Alemanha, teve como campeão o time brasileiro, que foi representado pelos profissionais do Centro de Estudos e Sistemas Avançados do Recife (CESAR).

A primeira fase do campeonato consiste em competições eliminatórias em cada continente. Na fase final os campeões continentais são avaliados por jurados em rodadas de testes de software. Na avaliação são considerados o processo de identificação de erros, relatórios e ensaios. Os softwares analisados são previamente escolhidos pelo júri e as equipes dispõem de três horas para análise e criação de uma estratégia de testes, reporte de falhas e relatório final sobre a qualidade do mesmo.

Nos últimos anos a busca pela qualidade de software tem crescido em todo o mundo. Anteriormente, no entanto, a cultura da qualidade era praticada comumente apenas por empresas especializadas neste segmento.

De acordo com uma pesquisa feita pelo Gartner Group, a ausência de um escopo bem definido é um dos principais motivos de fracasso em um projeto de software.

Isso ocorre porque frequentemente a fase de definição de escopo é subestimada e o resultado é um grande número de correções feitas durante a fase de desenvolvimento. A pesquisa reporta que normalmente o tamanho da fase de definição de escopo é inversamente proporcional ao da fase de correções, pois quanto mais bem definida for a fase inicial, menores serão as discussões desnecessárias na etapa final do projeto. Ainda de acordo com estes relatórios, atualmente esse problema ocorre em 25% dos projetos.

Com o objetivo de minimizar distorções na definição do produto em um processo de desenvolvimento de software, a equipe de qualidade necessita interagir com as demais equipes desde a fase de levantamento de requisitos e especificações. Quando existe uma sinergia entre a equipe de análise/desenvolvimento e a equipe de teste, aumentam-se as chances de o software apresentar boa qualidade, pois o conflito de visões/experiências entre elas agrega valor ao produto.

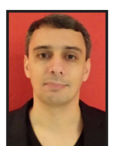
Com o intuito de facilitar a sinergia entre as equipes, é interessante a adoção de ferramentas automatizadas. Como diferencial, algumas dessas ferramentas possuem integração total a todas as etapas do projeto, desde o desenho até o lançamento, reunindo todas as equipes em uma mesma plataforma.

Tal integração agrega valor ao produto, pois quanto mais tempo se leva na detecção de um erro, mais caro o produto se torna. Se for detectado no cliente, por exemplo, esse custo pode ficar até 80% mais alto, uma vez que envolve deslocamentos, a credibilidade da empresa, entre outros fatores.

Enfim, seja uma área interna ou laboratório de software que cria suas próprias soluções, a sua reputação no mercado pode ser afetada sempre que um produto for lançado.

Por isso a preocupação com a qualidade de software por toda a empresa e em todos os momentos de um projeto torna-se cada vez mais evidente em qualquer empresa/profissional que queira se destacar no mercado.

Autor



Haroldo Pereira Nascimento
haroldo.nascimento@gmail.com
Graduado em Ciência da Computação pela UNESP, mestre em Informática pelo ITA e dourando pela USP. Certificado PMP, Cobit, Itil, CSPO, SCM e OCEB. Trabalha como gerente de projetos em TI na empresa Software Express. Grande defensor da importância do processo de qualidade de software em projetos.

Autor

Francisco Carlos de Matos Jr
primo.matos@gmail.com
Graduado em Ciência da Computação e Pós-Graduado em Gestão de Projetos. Certificado em Itil, Cobit, Ctfl, Scrum Master (PSM), ISO 27002 e Green IT Citizen. Trabalha como Gerente de Projetos em TI na empresa Software Express.

Links:

LAUBE, Klaus. TDD: Desenvolvimento Orientado a Testes.
http://klauslaube.com.br/2011/01/27/tdd-desenvolvimento-orientado-testes.html

CEVIU, Equipe. Teste de Sistema (System Integration Test). Portal Ceviu.
http://blog.ceviu.com.br/info/artigos/teste-de-sistema-system-integration-test

GALEOTE, Sidney. Tipos de testes de software.
http://www.galeote.com.br/blog/2011/06/tipos-de-testes-de-software

TRUST, Target. Ferramentas para Automação de Teste de Software.
http://www.targettrust.com.br/blog/desenvolvimento/curso-de-ferramentas-para-automacao-de-teste-de-software

BSTQB. Sobre a Certificação Foundation.
http://www.bstqb.org.br/?q=node/28

QATEST. Auditoria de Qualidade.
http://qatest.com.br

Portal IBM: Teste e Qualidade de Software
https://www.ibm.com/developerworks/community/blogs/tlcbr/entry/teste_e_qualidade_de_software

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!

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 codigo (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