Proposta de um Framework para aplicações web

Proposal for a Framework for web applications

Artigo Original

Original Paper

Francisco de Oliveira Dantas Filho [1]
Hevanderson da Silva [1]
Angelo Márcio de Paula [2]

Palavras-chaves:

Web

Software

Padrões de Projeto

Arquitetura

Resumo

Aplicações envolvendo internet são denominadas aplicações web e requerem requisitos especiais de arquitetura de software para um correto desenvolvimento e funcionamento. Este trabalho tem como proposta apresentar as técnicas de software livre, formando uma arquitetura para construir aplicações web que funcionem na internet ou intranet. Todo framework foi desenvolvido na plataforma Java com o uso de Servlet, JSP e JavaBean. Ao final, o processo realizado apresentou alta produtividade, pois retirou do desenvolvedor a responsabilidade de programar o código de comunicação envolvendo o protocolo HTTP e as trocas de mensagens entre as camadas da aplicação. Desta forma, o programador se concentrará em programar somente a regra de negócio do sistema proposto.

Abstract

Applications involving Internet are called web applications and require specific requirements for software architecture for a correct development and functioning. This work has a proposal present the techniques of free software forming architecture to build web applications operating on the internet or intranet. Finally, the processor conducted showed high productivity, because withdrew from the developer the responsibility to set the code of communication involving the HTTP protocol and the exchange of messages between application tiers. Thus, the developer will focus on programming only the business rule of the proposed system.

Key words:

Web

Software

Design Pattern

Architecture

1. Introdução

Aplicações de *software* são complexas porque modelam a complexidade do mundo real. Atualmente, aplicações típicas são muito grandes e complexas para que um único indivíduo consiga compreendê-la. Sobre essas circunstâncias, a construção e a manutenção de grandes aplicações tornam-se um sério problema.

A demanda por mais serviços, novos produtos, pela melhoria da qualidade e menores preços vêm, já há um bom tempo, impulsionando o estudo de técnicas de desenvolvimento de software que possam ser efetivamente utilizadas para a obtenção desses objetivos. Além do aspecto puramente técnico, encontramos ainda enorme dificuldade em gerenciar o processo de desenvolvimento.

Este trabalho reúne as técnicas denominadas *Padrões de Projetos* para construção de aplicações que usam a *internet/intranet* como meio de comunicação, formando um *framework*, que atinja desde o cliente, executando uma página no navegador, até o dado,

[1] Discente do Curso de Sistema de Informação - Centro Universitário de Volta Redonda - UNIFOA
[2] Coordenação de Sistema de Informação e Engenharia Elétrica - Centro Universitário de Volta Redonda - UNIFOA – Volta Redonda-RJ

sendo gravado no banco de dados, ou seja, uma proposta de uma arquitetura para desenvolvimento *web* em uma totalidade razoável.

Todos os testes foram realizados com produtos *open source*, destacando-se a linguagem de programação *Java* através das tecnologias *Servlet* e *JSP* que são especialidades para desenvolvimento *web*.

Ao final, é apresentado um *framework* com uma arquitetura madura, pronta para ser usada na construção de aplicações *web* por qualquer equipe de desenvolvimento, a qual tenha conhecimentos básicos na linguagem *Java*.

2. Revisão Bibliográfica

2.1 Trabalhos Realizados

Entre os trabalhos pesquisados no meio acadêmico e na comunidade de desenvolvedores de código aberto (*open source*), destacam-se sob o ponto de vista acadêmico e contribuíram para o desenvolvimento desta pesquisa:

- *Framework Apache Struts*, desenvolvido por *Struts* (2009);
- *Framework JavaServer Faces*, desenvolvido por SDN (2009).

O *Apache Struts* é um projeto popular desenvolvido por *The Apache Software Foundation* (STRUTS, 2009). Trata-se de um projeto *open source* para desenvolvimento de aplicações *web* usando a linguagem de programação *Java*. O *Struts* é empregado onde há necessidade de criação de *web sites* que possuem respostas dinâmicas. Aplicações *web* baseadas em JSP (*Java Server Pages*), ASP (*Active Server Pages*) e PHP (*Hypertext Preprocessor*) misturam dados, controle e visualização em apenas um componente e, na prática, verifica-se uma dificuldade na manutenção destes *sites*, principalmente, quando se trata de sistemas de médio para grande porte. Então, umas das preocuções em uma aplicação é ter uma clara separação das camadas, o que é alcançado com a utilização da arquiteura MVC (*Model-View-Controller*). O *Struts* foi concebido para ajudar os desenvolvedores criar aplicações *web* que utilizam a arquitetura MVC. Esse projeto é mantido por um grupo voluntariado que colabora entre si desenvolvendo o *framework* e mantendo-o dentro das características de código aberto.

O *framework JavaServer Faces* também é usado para desenvolvimento de aplicações *web* dinâmicas visando à separação das camadas. É mantido pelo *Java Community Process* (JSR-314) e estabelece o padrão do lado servidor para a construção de interfaces com o usuário. Com as contribuições do grupo de especialistas, o JSF foi concebido de forma a ser aproveitado por ferramentas as quais irão tornar ainda mais fácil o desenvolvimento de aplicações *web*.

A Facilidade de uso é a principal meta do JSF, pois ele define claramente uma separação entre a aplicação lógica e a camada de apresentação tornando fácil a ligação entre a camada de apresentação e a lógica do negócio. Esse projeto permite, a cada membro da equipe de desenvolvimento, concentrar-se no desenvolvimento da lógica de negócio, ficando a cargo do JSF as complexidades de ligação entre as camadas. Por exemplo, o desenvolvedor não precisa ser especialista em HTML, pois o JSF possui TAG´s que encapsulam a regra de apresentação através de passagem de parâmetros, sem a necessidade de escrever qualquer *script* (SDN ,2009).

2.2 Revisão Tecnológica

2.2.1 Java Beans

JavaBeans são componentes de *software* escritos em *Java*. A API de *JavaBeans* foi criada pela *Sun Microsystem* com a cooperação da indústria e determina as regras que o programador de *software* deve seguir a fim de criar componentes de *software* reutilizáveis e independentes. Assim como muitos outros componentes de *software*, os *Beans* encapsulam tanto estado quanto comportamento. Ao usar a coleção de tags relacionadas a *Beans* de JSP nas suas páginas da *web*, os programadores de conteúdo podem alavancar o poder de *Java* para adcionar elementos dinâmicos às suas páginas sem escrever uma única linha de código *Java*.

Então, um *JavaBean* é uma classe que segue certo padrão de projeto. O padrão tem

uma semântica bem definida para permitir que a estrutura do *JavaBean* seja descoberta através da introspecção de sua classe compilada, usando a API de reflexão *Java*. Os requisitos fundamentais de um *JavaBean* são os seguintes: Ser uma classe *Java public*, possuir um construtor sem nenhum argumento e, normalmente, possuir uma ou mais propriedades (ou atributos) que podem ser manipulados através de métodos públicos (DEITEL E DEITEL, 2006).

2.2.2 Servlet

Um *Servlet Java* é um programa no lado do servidor que é ativado pelo servidor *web* para atender aos pedidos HTTP. Um *Servlet* é executado em uma máquina virtual *Java* no servidor da *web* e, normalmente, realiza alguma computação para gerar o conteúdo da resposta HTTP (DEITEL E DEITEL, 2006). Em comparação a outras técnicas de programação no lado do servidor, os *servlets* representam uma alternativa mais madura e confiável.

2.2.3 JSP

Devido ao grande sucesso que é o ASP (*Active Server Page*) da Microsoft, a *Sun Microsystem* decidiu construir uma versão tão poderosa quanto ao ASP da Microsoft, o *JavaServer Pages* (JSP). A tecnologia JSP é incorporado ao modelo de *Servlet*. A ideia básica das JSP é permitir que o código *Java* seja misturado com modelos HTML e XML estáticos. O propósito da lógica *Java* é gerar conteúdo dinâmico na página, enquanto a linguagem de marcação manipula a estruturação e a apresentação dos dados. (DEITEL E DEITEL, 2006).

3. Materiais e Métodos

3.1 Arquitetura Aplicação

Os dois *frameworks* (*Struts e JSF*) dominam o mercado de desenvolvimento *web*, porém, eles apenas tratam a camada que faz a interface com os pedidos HTTP. Além do mais, como eles cobrem praticamente todos os pontos relacionados ao desenvolvimento *web*, os mesmos se tornam complexos, exigindo um conhecimento extra além dos *JSP*, *Servlets* e *JavaBean*.

A proposta desta pesquisa é apresentar um *framework* para desenvolvimento *web* extendido até a camada da regra de negócio e a camada de dados sem ter que apresentar conhecimentos além do trivial (*JSP, Servlet e JavaBean*).

A arquitetura da aplicação deve ser projetada de modo que determinadas situações típicas não se tornem problemas os quais impedirão a aplicação de atender aos requisitos do sistema. São consideradas as seguintes situações como críticas:

- Sistemas envolvendo interfaces, servidor de aplicação remoto e banco de dados.
- Aplicações distribuídas.
- Aplicações que acessam dados em mais de uma fonte de dados.

3.1.1 Sistemas envolvendo interfaces, servidor de aplicação remoto e banco de dados

As seguintes situações estão presentes nesse caso:

- O sistema tem que aceitar dados do usuário, atualizar o banco de dados e retornar, mais tarde e em outro cenário, o dado para o usuário.
- A existência de várias formas em que o dado pode ser aceito e apresentado no sistema do usuário.
- Dados que alimentam um sistema em um formulário deverão ser restauráveis em outro formulário.

Se o sistema desenvolve um simples componente que interage com o usuário e também mantém o banco de dados, então um requerimento para suportar um novo tipo de apresentação necessitará o reprojeto do componente.

No estudo de caso proposto, a clínica fornece a facilidade de acompanhamento dos resultados. Quando o usuário se autenticar no sistema, se ele for o paciente, ele verificará o resultado de um determinado procedimento; se ele for o administrador, verificará a quantidade de procedimentos realizados em um

período de tempo; se ele for o fisioterapeuta, dará um parecer no procedimento realizado. Ou seja, a mesma informação pode ser visualizada de várias formas, de acordo com o perfil do solicitante. Aqui, o mesmo dado que representa o procedimento é visto em múltiplas formas, mas é controlado por uma única entidade na aplicação. Assim, três tarefas ocorrem ao mesmo tempo na aplicação:

- Gerenciar a interação do usuário com o sistema.
- Gerenciar o dado atual.
- Formatar o dado em múltiplas formas.

Dessa forma, um simples componente que faz todas as tarefas pode ser dividido dentro de três componentes independentes. Todas três tarefas podem ser tratadas por diferentes componentes. Então, a solução segundo Deshmukh e Malavia (2003) é separar os dados da apresentação dos dados de manutenção e ter um terceiro componente coordenador. Esses três componentes são chamados "Modelo", "Visão" e "Controlador". Eles formam o básico do padrão MVC. A Figura 1 mostra o relacionamento entre os componentes do modelo MVC.

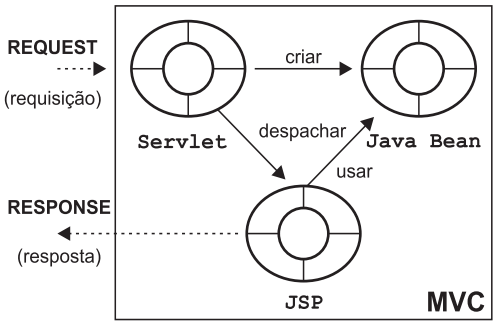


Figura 1. Modelo-Visão-Controlador

O modelo é responsável por manter o dado ou o estado da aplicação. Ele também gerencia o armazenamento e recuperação do dado armazenado. O componente *JavaBean* em conjunto com o *framework Hibernate* que realiza persistência objeto relacional foi aplicado nessa camada.

A visão contém a lógica de apresentação. Ela mostra o dado contido no modelo para o usuário. Ele também permite o usuário interagir com o sistema e notificar o controlador às ações do usuário. Os componentes *web JavaServer Page* (JSP) e o *JSP Standard Tag Library* (JSTL) foram aplicados na construção dessa camada.

O Controlador gerencia o modelo e a visão. Ele instancia e associa o modelo e a visão. Dependendo do requerimento da aplicação, ele pode instanciar múltiplas visões e pode associá-las com o mesmo modelo. Ele escuta as ações do usuário e manipula o modelo de acordo com a regra de negócio. Um *Servlet* foi aplicado nessa camada.

Desenvolver uma aplicação seguindo o padrão *model-view-controller* permite criar múltiplas visões para um mesmo dado. Mudanças ocorrem nos componentes do modelo ou nos componentes de visão de forma independente. A visão permanece a mesma. Isso aumenta a manutenibilidade e a extensabilidade do sistema.

3.1.2 Em aplicações distribuídas

Os componentes do cliente e os componentes do servidor residem em locais remotos e a comunicação ocorre sobre a rede de comunicação. Sendo assim:

- O banco de dados fica no lado do servidor
- O servidor fornece métodos *get's (getImagem( )*, por exemplo) para que o mesmo possa chamá-lo um por um para reaver os dados.
- O servidor fornece métodos *set's (setImagem(byte image[ ]),* por exemplo, para o cliente, para que o mesmo possa chamá-lo um por um para atualizar os dados no banco de dados.

Cada chamada entre o cliente e o servidor é uma chamada de métodos remotos com uma substancial sobrecarga na rede. Se a aplicação cliente chama os métodos *get's* e *set's* para reaver ou atualizar simples valores de atributos, ocorrerão muitas chamadas remotas para esse atributo. Essa chamada remota gera muita sobrecarga na rede e, consequentemente, uma diminuição do desempenho do sistema.

A camada de negócio acessará o banco de dados diretamente ou via a camada de recursos, e colocará o mecanismo de acesso dentro de um conjunto de entidades e *beans*

controladores. Esses componentes expõem o dado via interfaces remotas. As aplicações *desktop`s*, os *servlets* e páginas JSP na camada de apresentação precisam acessar esses dados de negócio, eles fazem isso chamando os métodos das interfaces remotas implementadas por *Javabeans*.

Para exemplificar, vamos considerar que desejamos incluir dados de uma paciente na base de dados e se trata de uma aplicação distribuída. Os dados são: identificador, nome e imagem (foto), o qual está encapsulado em um *Javabean* chamado *Laudo*. O acesso para a informação está encapsulado pela aplicação da camada de negócio com a ajuda de um *Javabean* controlador chamado *LaudoFacade*. Esse controlador expõe métodos para o cliente remoto, tais como: *getId()*, *setId(...)*, *getNome()*, *setNome(...)*, *getImagem()*, *setImagem(...)*. Os *servlets* e páginas JSP então têm que chamar cada método um por um remotamente no servidor. Conforme a Figura 2.

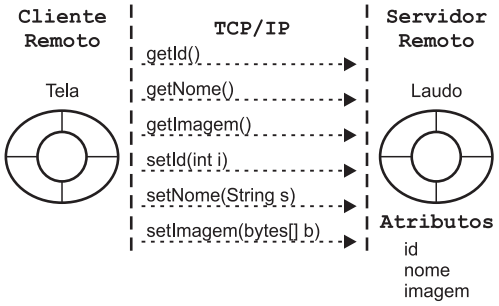


Figura 2 – Sobrecarga de chamadas em objetos remotos

A Figura 2 apresenta alguns problemas:

- Um simples objeto de negócio tem muitos atributos
- Na maioria das vezes, o cliente requer vários atributos ao mesmo tempo e quase nunca apenas um atributo.
- A utilização dos métodos get's é muito maior que os métodos set's

Dessa forma, é preciso criar um objeto para encapsular todos os atributos que são requeridos pela aplicação cliente. Esse objeto é chamado de Value Object. Quando o cliente requer o dado do servidor, o componente do lado do servidor extrai os valores localmente e constrói o Value Object. Esse Value Object é então enviado para o cliente por valor e não por referência, o que significa que esse objeto é serializado e transferido pela rede bit a bit. O cliente do outro lado reconstrói o objeto localmente com todos os valores intactos. Uma vez que o objeto está local, o cliente consulta todos os atributos sem gerar sobrecarga na rede.

Agora, considerando o exemplo anterior, ao invés de fazer múltiplas chamadas remotas no objeto Laudo para reaver todos os atributos para formar um laudo, o cliente chama um simples método getLaudo no LaudoFacade, este vai extrair os dados do objeto Laudo, criar o LaudoVO e retorná-lo pela rede com todos os atributos estruturados em um objeto próprio para trafegar pela rede. Assim, o cliente acessará todas as informações em uma única vez localmente. A Figura 3 está ilustrando a solução.

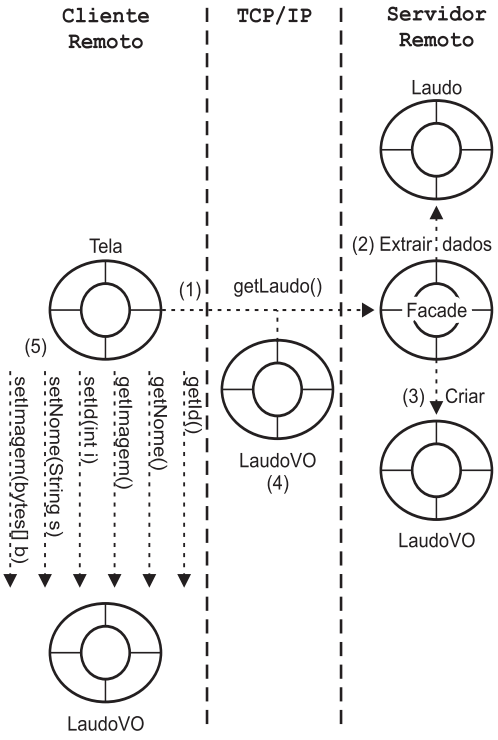


Figura 3 – Aplicação dos padrões Facade e Value Object

O cliente pode ser uma página *web* JSP, um *servlet*, uma *applet Java*, um *JPanel* no formato *desktop* que faz chamadas remotas para o objeto de negócio na camada de negócio no servidor. O objeto de negócio é um componente que cria o *Value Objetct*. O cliente sempre fará chamadas para o objeto de negócios.

A interface remota é simplificada porque múltiplos métodos retornando um simples valor são simplificados dentro de um simples método, retornando um grupo de múltiplos valores. A redução do número de chamadas, por meio da rede, melhora o tempo de resposta da

aplicação. Se o cliente deseja atualizar o valor do atributo, ele primeiro atualiza o valor no *Value Object* local e, então, o envia para o servidor para que possa ser persistido o novo valor. Tudo isso acontece usando o mecanismo de passagem por valor.

O Value Object pode ficar desatualizado, isto é, se o cliente adquiriu um *Value Object* por um longo período de tempo, há possibilidade de a informação ter sido atualizada por outro cliente. No caso de *Value Object* mutável, requisições de atualização de dois ou mais clientes podem resultar em conflito de dados.

Resumindo, um *Value Object* é um pequeno objeto *Java* serializado que é usado para conduzir grupo de dados sobre a rede de um componente residente em outra camada de uma aplicação multicamadas distribuídas. O seu principal propósito é reduzir sobrecarga de comunicação, reduzindo o número de chamadas remotas entre os componentes distribuídos.

3.1.3 Em aplicações que acessam dados em mais de uma fonte de dados

As aplicações envolvem atualizações e consultas de fontes de dados diferentes: banco de dados relacionais, sistemas de arquivos, arquivos *XML*, assim por diante. As seguintes situações ocorrem:

- As aplicações envolvem atualizações e consultas de fontes de dados iguais, como um banco de dados, mas bancos de dados são fornecidos por diferentes fabricantes, por exemplo, Oracle, SQLServer da Microsoft, DB2 da IBM e outros, onde cada fabricante construiu o seu próprio mecanismo de acesso.

- Alguns bancos de dados permitem manipulação de dados via store procedure. O processo de chamada da store procedure são dependentes da implementação do sistema do banco de dados.

A Figura 4 mostra a dificuldade que a camada de negócio encontra para acessar dados de diferentes fontes, visto que cada fonte de dados possui o seu próprio mecanismo de acesso. Outro problema que existe também é quando a camada de negócio precisa acessar apenas uma fonte de dados, mas o banco de dados precisa mudar de endereço físico em algum momento, logo, todos os componentes da camada de negócio que acessam a fonte de dados diretamente serão alterados. Então, quando temos várias fontes de dados ou quando a mesma não está corretamente definida, apresentará um grande problema de manutenção e de baixa flexibilidade do sistema.

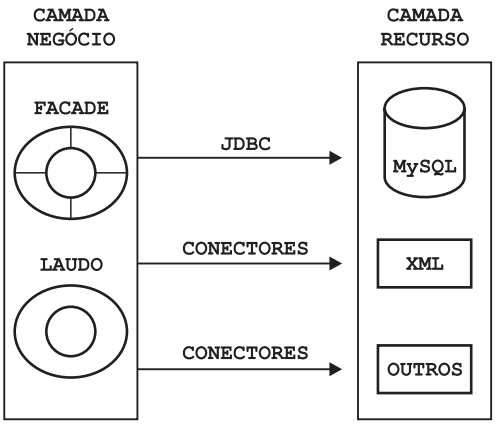


Figura 4 – Negócio acessando base de dados

Dessa forma, a lógica de negócio implementada pelos objetos de negócio não deve depender do tipo do armazenador de dados que é usado. O algoritmo de negócio deve ser separado do código que acessa o banco de dados. A lógica de negócio implementada pelo objeto de negócio não deverá depender da forma em que o dado armazenado é acessado.

A solução apresentada na Figura 5 cria um objeto especial que se ocupa em acessar os bancos de dados. O DAO é uma classe abstrata ou uma interface que tem métodos como *select(..)*, *upadate(..)*, *delete(...)* e *insert(..)*, as classes derivadas deverão então implementar esses métodos na forma específica que é requerida pelo banco de dados individualmente. Então os *Dao's* encapsulam a lógica de acesso aos dados, e o objeto de negócio que usa as classes Dao's não precisa conhecer sobre algum conjunto de código *SQL* ou as *API's* de baixo nível fornecidas pelo fabricante do banco de dados.

O padrão *DAO* fornece uma camada de abstração entre as camadas de negócio e a fonte de dados. Objetos da camada de negócio acessam a fonte de dados via *Data Access Object*. O *DAO* encapsula os detalhes de persistência e fornece um conjunto padrão de interface para acessar o dado. Esse objeto realiza todas as ope-

rações relacionadas à fonte de dados, tais como, consultas SQL. Isso limita o impacto de mudanças da fonte de dados para apenas as classes que compõe o *DAO*, isso melhora a flexibilidade e manutenibilidade dos objetos de negócio.

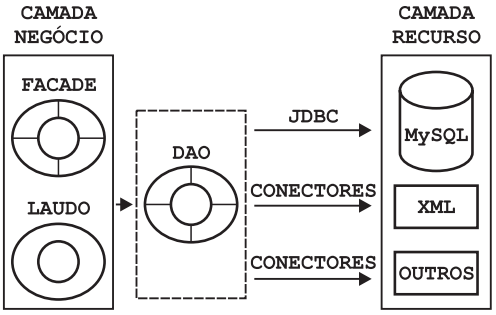


Figura 5 – Exemplo do padrão DAO

O objeto de negócio usa o *DAO* para alcançar o dado armazenado, o próprio *DAO* abstrai e encapsula as características da fonte de dados que a camada de negócio precisa para acessar os dados. Dessa forma, a fonte de dados será qualquer espécie de banco de dados ou sistema legado, e a aplicação vai selecionar dinamicamente a fonte de dados plugando o apropriado *DAO*.

3.2 Arquitetura da Aplicação proposta

A aplicação será construída com componentes para as camadas cliente, apresentação, negócio, integração e dados e foi influenciada por Alur, Crupi e Malks (2001), Deshmukh e Malavia (2003) e Paula (2007). A Figura 6 mostra os padrões cooperando entre si, formando a arquitetura.

A camada do cliente permitirá ao cliente interagir com a aplicação, essa pode ser uma aplicação *web* rodando em um *browser*, uma aplicação *desktop* ou até uma aplicação *Wirelles* rodando em um *Palm*. A camada de apresentação define o nível de permissão do usuário, encapsula a lógica da *interface*, gerencia a sessão do usuário e recebe e envia respostas para o usuário. A camada de negócios é o módulo responsável pela realização das regras de negócio da aplicação. A camada de integração é responsável pela comunicação com os recursos externos. A camada de recursos é responsável por manter os dados persistidos.

Dessa forma, o cliente *web* fará sempre uma requisição para um *Front Controller*, que delegará a um *Dispatcher* a função de criar um *action* para atender a solicitação e encaminhar a resposta para um *View* formar a saída para o cliente, essa funcionalidade também é mostrada na Figura 7. O cliente *Desktop* acessará o *Facade* diretamente através de uma chamada remota. O padrão *Service Locator*, faz a localização dos objetos remotos.

O Facade é um controlador da regra de negócio. Ele conhece o fluxo de atendimento da aplicação, sua principal função é acessar a base de dados através dos *Dao`s* criados pelos *Abstracts Factory e Factory*, executar a regra de negócio encapsulada pelos objetos de negócios e garantir a resposta para o cliente criando *Value Objets* apropriados para cada solicitação. O sistema possuirá *Implementation Dao* para cada meio de armazenamento no sistema. Os meios mais comuns são XML, Sistema de arquivo e banco de dados relacional.

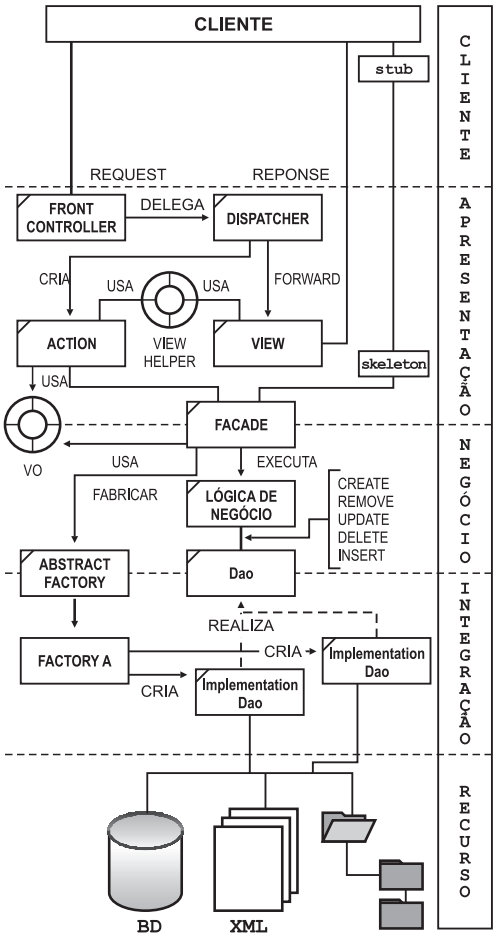


Figura 6– Arquitetura da aplicação segundo Paula (2007)

4. Resultados

A Figura 7 mostra as classes envolvidas no controlador da camada de apresentação. As classes são empacotadas em um arquivo com extensão *jar*, nesse caso, a sua reutilização é

feita adicionando o arquivo de nome *controle.jar* no *path* da aplicação e a devida configuração do *web.xml*.

Então, quando o cliente disparar uma URL (Exemplo: *pesquisar.do*) o *servlet FrontController* será acionado pelo servido *web* e várias ações ocorreram no servidor:

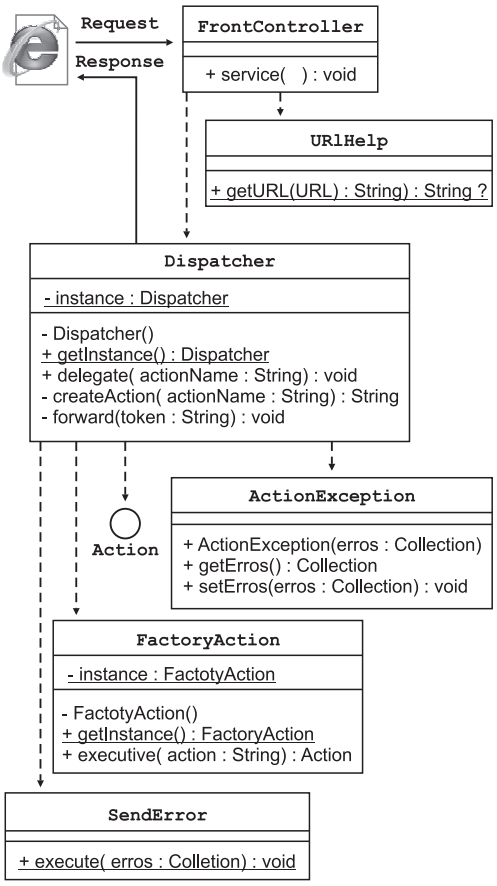


Figura 7– Controlador da camada de apresentação

1. Uma *Thread FrontController é disparado*.
2. A objeto *URIHelp* reconhece a *URI* solicitante.
3. O *FrontController* solicita uma instância do objeto *Dispatcher*.
4. O *FrontController* delega as demais responsabilidades para o objeto *Dispatcher*.
5. O *Dispatcher* solicita uma instância do objeto *FactoryAction*.
6. O *Dispatcher* pede ao *FactoryAction* o action presente na *URI*.
7. O *Dispatcher* executa o objeto *Action* (Exemplo: *pesquisarAction.class*)
8. O *Dispatcher* faz um *forward* (encaminha o pedido) para o documento (JSP, HTML, etc..) conforme o retorno do *Action*.

A Figura 8 evidencia o protótipo construído para testar o framework.

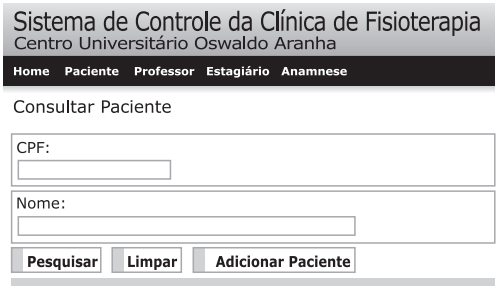
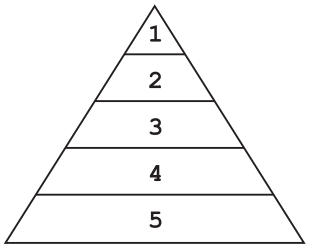


Figura 8– Estudo de caso

5. Discussão

Temos, ao longo dos anos, vários casos que indicam insucesso quando se trata de desenvolvimento de software. De forma empírica, constatamos as seguintes situações relatadas na Figura 9. Temos que unir esforços para atingirmos o topo da pirâmide e com um custo, qualidade e prazo dentro da normalidade.



1. Produção com sucesso
2. Retrabalho
3. Grande retrabalho
4. Entregue, mas usuário não usa
5. Pagos, mas não entregues

Figura 9– Crise do software

Vários fatores influenciam no desenvolvimento do software e destacamos os mais importantes: alterações nas metas, falha no gerenciamento de riscos, complexidade do software e projeto da arquitetura. A nossa pesquisa é aplicada ao *Projeto da Arquitetura* e pretende-se, dessa forma, uma arquitetura que permita atender aos princípios básicos de uma aplicação: Performance; transparência; flexibilidade; confiabilidade e escalabilidade. Todos os membros da equipe precisam codificar da mesma forma, facilitando a rotatividade dos profissionais, fato concreto na atual escassez de profissionais de TI. Com isso, a nossa proposta contribuirá tanto no desenvolvimento - devido à padronização imposta na manutenção

do sistema devido ao uso dos padrões de projeto- quanto na sedimentação do sistema, pois está construído com boas regras de codificação, conhecidas e usadas mundialmente.

6. Conclusão

Concluímos que, devido à importância alcançada pelas aplicações *web* e a forma que a *internet* invadiu o nosso cotidiano, seja necessário realizar pesquisa para auxiliar na construção dos *softwares*, através de soluções reutilizáveis.

Percebemos também que no decorrer do desenvolvimento de uma aplicação, é comum nos depararmos com problemas que precisam de conhecimentos estratégicos para serem resolvidos. Para cada problema que encontramos, imediatamente, é considerado inúmeras formas de resolvê-lo, incluindo soluções de sucesso já vividas ou soluções completamente novas. O segredo é documentar cada solução aplicada e à medida que reusamos uma solução com sucesso, considera-se que temos uma solução para resolver um determinado problema.

Dessa forma esse *framework* apresentado poderá ser evoluído para atender a novas especificações, desde que devidamente documentado e compartilhado na comunidade acadêmica de desenvolvedores.

Finalizando, esse *framework* está maduro para ser usado em desenvolvimento de aplicações *web*, pois foi desenvolvido com técnicas já testadas mundialmente.

7. Referências

ALUR, D.; CRUPI, J.; MALKS, D.; **Core J2EE Patterns:** Best Practices and Design Strategies, Califórnia, Sun Microsystems Press, 2001.

DEITEL, H.M.; DEITEL, P.J.; **Java: Como Programar**, 6 ed. São Paulo, Pearson, 2006.

DESHMUKH, H.; MALAVIA, J.; SCWCD, **Exam Study Kit: Java Web component Developer Certification,** Greenwich, 2003.

PAULA, A.M.; **Gerenciamento, Análise e Transmissão de imagens**, On-Line, Via TCP/IP, Dissertação de Mestrado, Mestrado em Tecnologia, CEFET, Rio de Janeiro, RJ, Brasil, 2007.

SDN Sun Developer Network: JavaServer Faces Technology, 2009. Disponível em:< http://java.sun.com/javaee/javaserverfaces/> Acesso em: 18 Mai. 2009.

STRUTS The Apache Software Foundation, 2009. Disponível em: <http://struts.apache.org> Acesso em: 18 Mai. 2009.

Endereço para Correspondência:

Angelo Márcio de Paula
Coordenação de Sistema de Informação
e Engenharia Elétrica
angelo.paula@foa.org.br

Centro Universitário de Volta Redonda
Campus Três Poços
Av. Paulo Erlei Alves Abrantes, nº 1325,
Três Poços - Volta Redonda / RJ
CEP: 27240-560

Informações bibliográficas:
Conforme a NBR 6023:2002 da Associação Brasileira de Normas Técnicas (ABNT), este texto científico publicado em periódico eletrônico deve ser citado da seguinte forma: FILHO, Francisco de Oliveira Dantas; SILVA, Hevanderson da; PAULA, Angelo Márcio de. Proposta de um Framework para aplicações web. **Cadernos UniFOA**. Volta Redonda, ano IV, n. 11, dezembro 2009. Disponível em: <http://www.unifoa.edu.br/cadernos/edicao/11/19.pdf>