

CENTRO PAULA SOUZA
FACULDADE DE TECNOLOGIA DE MOCOCA
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E
DESENVOLVIMENTO DE SISTEMAS

Fernando Teixeira Alves de Araujo

***DESIGN PATTERNS INTEGRANDO WEB SERVICE COM
FRAMEWORKS: IMPLEMENTANDO PROJETOS NAS
PLATAFORMAS WEB E DESKTOP QUE INTEGRAM-SE POR WEB
SERVICE SOAP***

Mococa-SP
Dezembro/2015

FERNANDO TEIXEIRA ALVES DE ARAUJO

**DESING PATTERNS INTEGRANDO WEB SERVICE COM
FRAMEWORKS: IMPLEMENTANDO PROJETOS NAS
PLATAFORMAS WEB E DESKTOP INTEGRAM-SE POR WEB
SERVICE SOAP**

Trabalho de Conclusão de Curso
apresentado à FATEC - Faculdade de
Tecnologia de Mococa, para obtenção do
título de Tecnólogo no Curso Superior de
Tecnologia em Análise e
Desenvolvimento de Sistemas.

Orientador: **Prof.^a Ms. Fabrício Gustavo
Henrique**

**Mococa-SP
Dezembro/2015**

Dedico este trabalho à Deus por me possibilitar a oportunidade concluir esse trabalho, aos meus irmãos, as minhas cunhadas, a meus amigos, e todos que tiveram participação na minha vida acadêmica.

AGRADECIMENTOS

- ✓ À Deus por sempre me dar as possibilidades de seguir os melhores caminhos;
- ✓ À minha família, especialmente aos meus irmãos e minhas cunhadas, Fabiano e Aline que me incentivaram e apoiaram a iniciar o curso o ensino superior, e Fagner e Silene que me incentivam a continuar e concluir o curso;
- ✓ Ao meu orientador, professor Fabricio, que me apoiou na realização dessa conclusão e que sem ele muito dificilmente teria terminado esse trabalho de forma eficiente e eficaz;
- ✓ Ao professor Wladimir que também me auxiliou na orientação desse trabalho;
- ✓ À todas as empresas que acreditaram no meu trabalho (NTS Informática, Alpha Softworks, Discover Technology e Magna Sistemas) que acreditaram no minha competência e me fizeram eu me tornar um grande profissional;
- ✓ Às todos meus colegas de acadêmicos, especialmente para a Janine e Carlos Hernani que além de colegas, foram grandes amigos.

“Para o pessimista, eu sou bastante otimista” (Hayley Williams).

RESUMO

Observando problema decorrente da necessidade de alteração de clientes de *Web Services* para aplicar funcionalidades ao mesmo, foi realizado um estudo de possibilitar que, a partir do cliente ser alterado, não necessite de ser feito um retrabalho após uma nova geração do cliente pelo WSIMPORT. Para viabilizar essas alterações, esse trabalho propõe a ideia de criação de um *proxy* para que as alterações não sejam realizadas diretamente nos códigos do cliente gerado, e sim por meio de uma camada, que é o *proxy*. Para exemplificar essa situação, foi criando um ambiente semelhante ao real interligando dois sistemas, um *web* e outro local, por intermédio de um terceiro sistema que é responsável por acessar o banco de dados do sistema local e sincronizar os dados com o sistema da *web* via *Web Service*, e aplicando a ideia, foi coletado os resultados e apresentado nesse trabalho.

Palavras-chave: Padrões de Projeto. Web Service. Frameworks, Proxy, SOAP, Integração.

ABSTRACT

Noting problems arising from the need for Web Services clients change to apply the features, a study was conducted to enable, from the client to change, need not be made a rework after a new client generation by WSIMPORT . To make these changes, this paper proposes the idea of creating a proxy for the changes are not made directly in the client code generated, but by a layer, which is the proxy. To illustrate this situation, it was creating an environment similar to real connecting two systems, one in web and other local one, through a third system that is responsible for accessing the local system database and synchronizes the data with the web system via Web Service, and applying the idea was collected and the results presented in this paper.

Keywords: Design Patterns. Web Service. Frameworks, Proxy, SOAP, Integration.

LISTAS DE FIGURAS

	Página
Figura 1 – Integração entre um cliente de <i>Web Service</i> e um <i>Web Service</i>	17
Figura 2 – Estrutura do <i>SAOP Message</i>	18
Figura 3 – Interação com os <i>handlers</i> na chamada do serviço.	23
Figura 4 – Composição dos conteúdos acessados pelos <i>handlers</i>	24
Figura 5 – Envolvendo um Objeto <i>Proxy/Decorator</i>	29
Figura 6 – Código de adição do <i>handler</i> em um Cliente de <i>Web Service</i>	31
Figura 7 – Código de adição do <i>handler</i> em um Cliente de <i>Web Service</i> com <i>Proxy</i>	32
Figura 8 – Simples instanciação de um Cliente <i>Web Service</i>	33
Figura 9 – Instanciação de um <i>Proxy</i> de Cliente <i>Web Service</i>	33
Figura 10 – Aplicação do <i>handler</i> com <i>Proxy</i> em um Cliente de <i>Web Service</i>	34
Figura 11 – Sobrecarga de um método com tipo não compatível com <i>Web Service</i>	35
Figura 12 – Método de atribuição de valor em objeto composto.	35
Figura 13 – Aplicação do <i>Proxy</i> em um Artefato do Cliente do <i>Web Service</i>	36
Figura 14 – Componentes do diagrama entidade-relacionamento.	37
Figura 15 – DER do Sistema “Plano de Saúde” – Plataforma <i>Web</i>	44
Figura 16 – Tela de Consulta do “Plano de Saúde” – Plataforma <i>Web</i>	44
Figura 17 – DER do Sistema “Plano de Saúde” – Plataforma <i>Desktop</i>	47
Figura 18 – Tela de Consulta do “Plano de Saúde” – Plataforma <i>Desktop</i>	48
Figura 19 – Tela de Cadastro/Edição do “Plano de Saúde” – Plataforma <i>Desktop</i>	49

LISTA DE ABREVIATURAS E SIGLAS

AOP	<i>ASPECT-ORIENTED PROGRAMMING</i>
API	<i>APPLICATION PROGRAMMING INTERFACE</i>
B2B	<i>BUSINESS TO BUSINESS</i>
DER	<i>DIAGRAMA ENTIDADE-RELACIONAMENTO</i>
EJB	<i>ENTERPRISE JAVABEANS</i>
HTTP	<i>HIPERTEXT TRANSFER PROTOCOL</i>
IDE	<i>INTEGRATED DEVELOPMENT ENVIRONMENT</i>
JAR	<i>JAVA ARCHIVE</i>
JAXB	<i>JAVA ARCHITECTURE FOR XML BINDING</i>
JAX-RPC	<i>JAVA API FOR XML – REMOTE PROCEDURE CALL</i>
JAX-WS	<i>JAVA API FOR XML – WEB SERVICE</i>
JCP	<i>JAVA COMMUNITY PROCESS</i>
JDK	<i>JAVA DEVELOPMENT KIT</i>
JDBC	<i>JAVA DATABASE CONNECTIVITY</i>
JMS	<i>JAVA MESSAGE SERVICE</i>
JPA	<i>JAVA PERSISTENCE API</i>
JRE	<i>JAVA RUNTIME ENVIRONMENT</i>
MER	<i>MODELO ENTIDADE-RELACIONAMENTO</i>
PAC	<i>PRÁTICO, ACESSÍVEL E CONFIÁVEL</i>
RMI	<i>REMOTE METHOD INVOCATION</i>
SEDEX	<i>SERVIÇO DE ENCOMENDA EXPRESSA NACIONAL</i>
SEI	<i>SERVICE ENDPOINT INTERFACE</i>
SOAP	<i>SIMPLE OBJECT ACCESS PROTOCOL</i>
SQL	<i>STRUCTURED QUERY LANGUAGE</i>
UF	<i>UNIDADE FEDERATIVA</i>
URI	<i>UNIFORM RESOURCE IDENTIFIER</i>
XML	<i>EXTENSIBLE MARKUP LANGUAGE</i>
W3C	<i>WORLD WIDE WEB CONSORTIUM</i>
WS	<i>WEB SERVICE</i>
WSDL	<i>WEB SERVICE DESCRIPTION LANGUAGE</i>

SUMÁRIO

1 INTRODUÇÃO	12
2 REFERENCIAL TEÓRICO.....	14
2.1 Java	14
2.2 Web Service	15
2.2.1 SOAP (<i>Simple Object Access Protocol</i>).....	18
2.3 Frameworks	19
2.3.1 JAX-WS (<i>Java for XML – Web Service</i>).....	20
<u>2.3.1.1 WSIMPORT.....</u>	<u>21</u>
<u>2.3.1.2 Handler.....</u>	<u>22</u>
2.3.2 Spring Framework.....	24
<u>2.3.2.1 Spring Framework JDBC</u>	<u>26</u>
2.4 Design Patterns	27
2.4.1 Proxy/Decorator	29
2.5 Integração	30
2.5.1 Integração de Web Service com SOAP Message Handler	31
2.5.2 Integração de Web Service com Spring Framework JDBC	34
2.6 Modelagem de Dados.....	36
2.6.1 Diagrama Entidade-Relacionamento	37
3 PROCEDIMENTOS METODOLÓGICOS.....	40
4 DESENVOLVIMENTO DO PROJETO “PLANO DE SAÚDE”	41
4.1 Sistema “Plano de Saúde” – Plataforma Web	41
4.1.1 Diagrama Entidade-Relacionamento	41
4.1.2 Interfaces	44
4.2 Sistema “Plano de Saúde” – Plataforma Desktop.....	45
4.2.1 Diagrama Entidade-Relacionamento	45
4.2.2 Interfaces	48
4.3 Sistema “Plano de Saúde” – Integração	49
5 RESULTADOS E DISCUSSÃO	51
6 CONSIDERAÇÕES FINAIS	53
REFERÊNCIAS.....	54

1 INTRODUÇÃO

O setor de desenvolvimento de sistemas é uma área que procura buscar um bom desempenho produtivo, pois isto impacta em custo de desenvolvimento.

Há muitas metodologias para melhorar a produtividade de uma equipe de desenvolvimento de software, a partir de gerenciamento de projeto por metodologias ágeis. Mas algumas vezes os gestores pensam muito mais em metodologias, e se esquecem de que é possível ganhar muita produtividade seguindo e inovando com boas práticas.

Atualmente as únicas equipes que desenvolvem manuais de boas práticas, e recomendações que buscam melhorar produtividade, são apenas os poucos arquitetos de *software* dentro de uma empresa, ou as equipes que desenvolvem as tecnologias e *frameworks*. Mas é dever das comunidades de desenvolvimentos compartilharem as melhores práticas para que possa alcançar ainda uma produtividade ainda maior.

Para alcançar a melhor produtividade, é necessário fazer experiências, e observar se os resultados são satisfatórios. Se forem satisfatórios, essas experiências podem servir de manual de melhores práticas para auxiliar no desenvolvimento.

Esse trabalho visa mostrar a viabilidade de usar um *proxy* para integrar *frameworks* com o cliente de *Web Service* SOAP em JAX-WS, apresentando duas situações. Uma situação *Web Service* integrando com o *Handler* e o outro integrando com *Spring Framework* JDBC.

Com os resultados satisfatórios, esse trabalho pode se tornar um manual de boas práticas.

Para alcançar o objetivo, esse trabalho está estruturado em 6 capítulos, sendo o primeiro a introdução e os demais com o desenvolvimento do trabalho, como descrito a seguir.

O segundo capítulo procura colocar todas as referências utilizadas no trabalho, como as linguagens de programação usada, conceitos e especificações, *frameworks* utilizados, conceitos de padrões de projeto, modelos de dados e suas formas de documentações.

No terceiro capítulo é exibido os procedimentos utilizados para desenvolver os estudos e o projeto para exemplificar a integração entre *frameworks* e *Web Service*.

No quarto capítulo é apresentado o projeto “Plano de Saúde”, o projeto foco do trabalho, e também são mostrados suas funcionalidades e suas características, além de diagramas referenciados a cada sistema presente nesse projeto.

O quinto capítulo apresenta os resultados obtidos com os estudos e com o projeto, e comenta também sobre os resultados positivos que foram consequências no decorrer do desenvolvimento do projeto.

E no sexto capítulo é feita as considerações finais para mostrar os resultados obtidos através dos estudos e as possíveis continuações que esse trabalho poderá ter.

2 REFERENCIAL TEÓRICO

2.1 Java

O Java é uma linguagem de programação para o desenvolvimento de sistemas computacionais. Sua principal característica é o uso da POO (programação orientada a objetos), que vem a ser a programação feita a partir da abstração dos objetos como são vistos no mundo real.

Em 1991, a Sun Microsystems financiou o desenvolvimento do Java com propósito inicial de fazer uma linguagem de programação voltada para microprocessadores, posto que os estes tinham uma expectativa de crescimento. Porém com o baixo avanço dos microprocessadores, e com o grande *Boom* da *Internet*, o projeto Java tomou outro rumo com outro propósito. Em maio de 1995, a Sun lançou o Java oficialmente em uma conferência do setor (DEITEL, 2010). Em 2009, a Oracle comprou a Sun com todos os seus projetos, incluindo o Java, por aproximadamente 7,4 bilhões de dólares (ORACLE, 2015).

O projeto Java resultou em uma linguagem baseada em C++ que foi criada por James Gosling. Foi chamado inicialmente de *Oak*, em homenagem a uma árvore de carvalho que era vista da janela do criador do Java. Porém como já existia uma linguagem com o mesmo nome, uma equipe do projeto em uma cafeteria alterou o nome para Java em homenagem a uma cidade produtora de um café importado (DEITEL, 2010).

O Java chamou a atenção da comunidade de negócio devido ao enorme interesse na *Web*. O Java é utilizado para do desenvolvimento de aplicativos corporativos de grande porte, melhorar as funcionalidades de servidores *Web*, desenvolver aplicações de uso popular e muitos outros propósitos (DEITEL, 2010). A fatia maior do desenvolvimento está no desenvolvimento de aplicações *Web*. Para dispositivos móveis cresceu muito após o advento dos *smartphones*, mais especificamente após o lançamento do Sistema Operacional *Android*. Nos últimos anos, a Oracle vem fazendo um grande investimento com o JavaFX para liderar o mercado de desenvolvimento *Desktop* (GUI – *Graphical User Interface*).

As Principais características que seduzem desenvolvedores para a linguagem de programação Java são a orientação a objetos, o gerenciamento de memória e principalmente a portabilidade (SIERRA e BATES, 2005). WORA, “*write once, run anywhere*” (escreva uma vez, rode em qualquer lugar) é o *slogan* do Java, é a principal promessa e a que mais atraem desenvolvedores e investidores de sistemas, onde com um mesmo código é possível rodar em qualquer dispositivo.

Após o lançamento do Java 8 e JavaEE 7, o Java voltou a liderar a lista de linguagens mais usadas entre os desenvolvedores, segundo a TIOBE, com cerca de 19,5% de todos os desenvolvedores de sistemas. Isso se deve principalmente a participação mais ativa das comunidades Java com a Oracle.

A orientação a objeto, principal característica do Java, busca abstrair os objetos do mundo real. Segundo GUERRA, um objeto é definido como uma entidade concreta enquanto a classe é definida como uma abstração de um conceito. Por exemplo, uma classe poderia representar uma Pessoa, um conceito abstrato, posto que não esteja definido qual é a pessoa em questão, podendo ser qualquer pessoa; E um objeto, criado a partir da abstração Pessoa, poderia ser eu ou você, tendo informações como nossos documentos, peso e altura, e podendo ter, inclusive, comportamentos como andar e falar.

2.2 Web Service

Com o grande crescimento da tecnologia, e com cada vez mais sistemas surgindo para resolver os problemas mais distintos, foi necessário ter uma tecnologia que integrasse os sistemas diferentes com seus propósitos comuns ou complementares. Por exemplo: o sistema de vendas tem que se comunicar com o sistema de controle de estoques, que mesmo sendo independentes, por cada um

manipular um conjunto de dados distintos, eles ainda tem um acoplamento, mesmo sendo baixo.

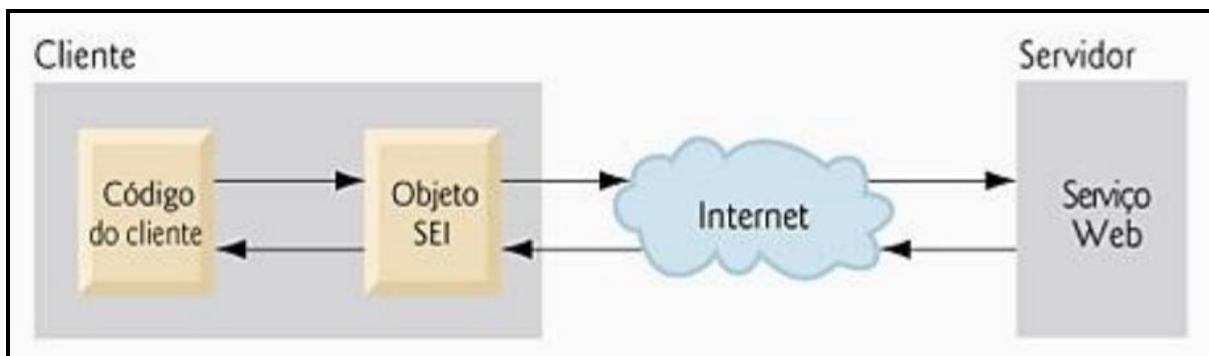
Observando essa necessidade, o Java tornou-se uma tecnologia muito utilizada para escrever softwares distribuídos em rede do tipo cliente/servidor (DEITEL, 2010). As aplicações do tipo cliente/servidor são aplicações que centralizam o processamento dos dados em uma única instalação que é o servidor, e disponibiliza os dados processados para as instalações que solicitaram o processamento, que são os clientes.

Um *Web Service* é um pedaço de lógica de negócios, localizado na *Internet*. Através do *Web Service* é possível realizar tarefas simples como uma autenticação em um site e tarefas complexas como facilitar a negociação entre múltiplas organizações (CHAPPELL e JEWELL, 2002). O *Web Service* pode ser utilizado para integrar aplicações de uma mesma organização ou organizações distintas com a intenção de fornecer serviços genérico. A Empresa Brasileira de Correios e Telégrafos utilizam *Web Service* para calcular o preço e prazo de um item pode ser enviado por SEDEX, e-SEDEX e PAC, esse serviço é um processamento genérico que pode ser usado por qualquer empresa que quer conhecer os valores e prazos dos serviços dos Correios de acordo com os endereços de origem e destino (CORREIOS, 2015).

Os *Web Services* costumam enviar os dados referente à integração por XML e via HTTP por serem a estrutura e protocolo mais aceitos, porem os *Web Services* não estão restritos a enviar seu dados apenas via HTTP podendo também ser enviada via RMI e JMS (SAUDATE, 2012). E também pode utilizar JSON como linguagem de marcação para armazenar seus registros enquanto estão sendo enviados.

O fluxo de processamento do *Web Service* começa com um aplicativo (cliente) enviando uma solicitação por intermédio de um rede (*Internet*), enviando os dados necessários para o processamento, para um *host* de *Web Service* (Servidor) que processa e retorna os dados processados (SAUDATE, 2012). Podemos ver a representação desse fluxo na Figura 1.

Figura 1 – Integração entre um cliente de *Web Service* e um *Web Service*.



Fonte: DEITEL, 2010.

É possível observar na Figura 1 que temos o objeto SEI (*Service Endpoint Interface*), também chamada como classe *Proxy*, é utilizada para realizar a comunicação com o *Web Service*. Um aplicativo invoca os métodos do serviço via SEI (DEITEL, 2010). Esse método é a abstração do *Web Service* para o aplicativo cliente.

Para especificar as regras que o *Web Service* utilizará para a entrada/saída do provedor de dados, o *Web Service* fornece um elemento chamado WSDL (SAUDATE, 2012).

A partir do WSDL é possível identificar regras que estão definidos nas seguintes seções:

- “*Types*”: seção que contém formatos dos elementos com seus respectivos atributos que trafegam na solicitação e na resposta do serviço. Os atributos podem ser simples como *string*, *int*, *boolean* e atributos complexos como outro elemento declarado;
- “*Message*”: seção que conecta os elementos presentes na seção *types* com a operação, presente na sessão *portType*, a ser realizada pelo *Web Service*;
- “*PortType*”: sessão que contém todas as operações disponíveis para serem executadas;
- “*Binding*”: sessão que especifica as configurações do *Web Service* como protocolos de envio que serão utilizados.

- “*Service*”: sessão que especifica o *host* de *endpoint* que o cliente acessa quando consumir o *Web Service*.

A partir da leitura do WSDL, o cliente sabe como trabalhar os métodos expostos pelo *Web Service*.

O modelo B2B (*business to business*) utiliza muito essa tecnologia quando é necessário integrar serviços de corporações diferentes. (DEITEL, 2010). Com o mercado cada vez mais competitivo, é necessário que as corporações sejam ágeis para lidar com os processo, e o *Web Service* provê tal agilidade entre a troca de informações entre sistemas diferentes e desacoplados.

2.2.1 SOAP (Simple Object Access Protocol)

O SOAP é um protocolo independente de plataforma é baseado em XML para a comunicação remota, geralmente em HTTP, que permite *Web Services* e Clientes se comunicarem, mesmo que o cliente seja desenvolvido em outra linguagem (DEITEL, 2010). É o protocolo mais comum no uso de *Web Services*.

Criado e mantido pela W3C (SAUDATE, 2012), que também é a empresa que especifica seus padrões de uso atuais e coordena as mudanças.

O SOAP tem uma estrutura chamada de SOAP *Message* que é uma estrutura feita em XML que é usada para trafegar os dados em uma solicitação e resposta do *Web Service* que é legível por computadores e humanos (DEITEL, 2010). Os dados são encapsulados no XML dentro da estrutura do SOAP *Message* que, segundo SAUDATE em 2012, utiliza uma estrutura semelhante ao do Java.

A Figura 2 mostra a estrutura básica do SOAP *Message*.

Figura 2 – Estrutura do SAOP *Message*.

```

1 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
2   <soapenv:Header>
3     <!-- Elementos do cabeçalho SOAP Message -->
4   </soapenv:Header>
5   <soapenv:Body>
6     <!-- Elementos do corpo SOAP Message -->
7   </soapenv:Body>
8 </soapenv:Envelope>

```

Fonte: Autoria própria, 2015.

Na estrutura do SOAP *Message* há um elemento externo denominado *Envelope*, e internamente há os elementos *Header* e *Body*, todos os com o *namespace* do SOAP (SAUDATE, 2012). É comum haver *namespaces* referentes aos pacotes dos objetos que são convertidos do Java para o XML.

Os elementos do SOAP *Message* têm os seguintes propósitos:

- “*Envelope*”: é apenas um container que armazena as tags *Header* e *Body*;
- “*Header*”: é um container de metadados, podendo ser referente à autenticação, endereço de retorno da mensagem entre outros;
- “*Body*”: é o container que armazena a especificação da operação a ser realizada no *Web Service*, os parâmetros da operação, e o retorno que essa operação resultará.

Em uma solicitação de um cliente e um *Web Service*, posto que o cliente já conheça as regras de estrutura de dados e métodos do *Web Service*, o cliente organiza a operação que será executado no *Web Service* juntamente com seus respectivos parâmetros, adiciona no *Envelope* SOAP e transmite para o servidor do *Web Service*. Quando o *Web Service* recebe o *Envelope* SOAP, é realizado a conversão dos parâmetros, e executado o método especificado na operação. Ao realizar o processamento, o *Web Service* organiza o objeto de resposta, converte em XML, inclui no *Envelope* SOAP de resposta e retorna para o cliente solicitante (DEITEL, 2010).

2.3 Frameworks

O *framework* é o nome dado para uma implementação que quando adicionado no projeto, procura facilitar o desenvolvimento do mesmo com rotinas consideradas genéricas para a implementação de uma funcionalidade.

Todo *framework* utiliza o conceito de “inversão de controle”, que é a transferência de responsabilidade de ocorrências de eventos para um sistema para um sistema maior, e com mais experiência em uma funcionalidade específica. A utilização de *frameworks* traz benefícios como menos linhas de código no projeto e códigos mais focados em regras de negócio (WEISSMANN, 2012).

Todos os projetos de sistemas desenvolvidos utilizam *frameworks*, podendo ser de terceiros ou pertencente à própria linguagem.

Os *frameworks* mais utilizados nos sistemas são aqueles que têm as funcionalidades mais genéricas que possibilita o uso nos sistemas mais distintos, dentre eles estão os *frameworks* de persistência de dados¹, injeção de dependência², desenvolvimentos de telas e criação e manipulação de serviços expostos.

2.3.1 JAX-WS (Java for XML – Web Service)

O JAX-WS é um *framework* de *Web Service*, desenvolvido com o núcleo do projeto Metro da Comunidade Glassfish e fornece diversas ferramentas para simplificar o desenvolvimento de *Web Services* (GLASSFISH, 2015).

A partir da versão do Java 6, o JAX-WS foi incorporado na JRE (THE APACHE, 2015), melhorando os problemas de incompatibilidades do *framework* com o JRE. Se JAX-WS for utilizado para versões anteriores ao Java 6, é necessário incluir os JARs *framework* ao projeto.

O padrão de comunicação do JAX-WS é o protocolo SOAP sobre HTTP (SAUDATE, 2012), porém é possível fazer consumir serviços utilizando o protocolo RMI como seu antecessor JAX-RPC.

O JAX-WS origina-se a partir do JAX-RPC. Com o surgimento e vasto uso do SOAP, como padrão de comunicação entre aplicações através de serviços, a Comunidade Java desenvolveu um *framework* para simplificar o uso desses serviços. A primeira versão foi a JAX-RPC 1.0. Alguns meses depois, enquanto a equipe JCP escrevia a especificação, percebeu-se que seriam necessários alguns ajustes. E então veio a surgir o JAX-RCP 1.1. Após um ano do lançamento da versão JAX-RCP 1.1, a equipe JCP quis desenvolver uma versão melhor observando as tendências da indústria. Essa versão deveria ser a JAX-RPC 2.0, porém a indústria não estava usando apenas Web Services RPC, mas estavam usando também Web Service orientado a serviço. Observando essa tendência, o JAX-RPC não apenas mudou a versão, mas também mudou o nome para JAX-WS

¹ Termo utilizado para expressar o armazenamento de dados em dispositivo físico como um disco rígido.

² Termo utilizado para expressar a forma de interligar componentes mantendo o menor nível de acoplamento.

2.0, substituindo RPC por WS para mostrar o principal propósito do *framework* (BUTEK e GALLARDO, 2015).

Com o JAX-WS, é possível realizar chamadas síncronas e assíncronas aos serviços do *Web Service* criando (THE APACHE, 2015) *bindings* que mapeiam esses os serviços que serão consumidos.

O JAX-WS utiliza anotação de classe Java como metadados para indicar que uma classe é um serviço e informar suas características. (THE APACHE, 2015). E juntamente com o JAX-B, que também utiliza anotação para realizar o mapeamento dos objetos com os dados que trafegarão pelo serviço.

O JAX-WS depende do JAX-B para executar as rotinas de geração de XML de *request* e *response* do SOAP *Message*, gerar, ler e interpretar WSDL de acordo com o metadados das classes Java (THE APACHE, 2015).

A partir do Java EE 5, ao implementar um *Web Service*, o container de aplicação se responsabiliza de criar e iniciar os recursos do *Web Service* a partir da injeção de dependência (THE APACHE, 2015). A partir da anotação, indicando quais são os *Web Services*, o container se responsabiliza pela exposição e consumo desse serviço.

O principal objetivo do *framework* JAX-WS, é simplificar a implementação de *Web Services*, mudando a responsabilidade da manipulação do *Web Service* da aplicação para o *framework*.

Com os mesmo JARs, o JAX-WS fornece as ferramentas necessárias para aprimorar o desenvolvimento de serviços e clientes de *Web Service* (THE APACHE, 2015). Em muitos casos, pode ser utilizado as mesmas anotações do JAX-WS como do JAX-B tanto no serviço como no cliente.

2.3.1.1 WSIMPORT

A arquitetura do uso de um *Web Service* consiste em expor as chamadas para as rotinas dos serviços e o consumo desse serviço. A exposição às rotinas de um serviço é implantada em um servidor e geralmente utiliza o protocolo HTTP nos padrões SOAP. O Consumo do serviço é realizado por uma aplicação separada, e os componentes usados são chamados de cliente.

Para gerar um cliente, existem diversas formas, quase todas são feitas por meio de IDE's, mas a maneira mais fácil ainda é o uso do comando WSIMPORT localizado na pasta "bin" da JDK (SAUDATE, 2012).

A execução do comando WSIMPORT é formada pela seguinte expressão: `wsimport [parâmetros] <wsdl_URI>`.

A primeira parte é o nome do arquivo binário que contém as rotinas de geração de cliente. Os parâmetros são opcionais e pode ser composto de várias opções que vão dar características específicas ao cliente gerado. E por ultimo temos a URI do arquivo WSDL com todas as informações referente ao Web Service.

De acordo com parâmetros informados no comando, é possível gerar três tipos de retorno, sendo esses arquivos com a extensão CLASS, JAVA e JAR. Os arquivos CLASS são arquivos compilados, que podem ser adicionados no projeto após a sua compilação. Os arquivos JAVA são arquivos que contém o código fonte em Java, que pode ser adicionados ao projeto junto com todo o restante do código fonte. Um JAR é uma dependência que contém todos os arquivos CLASS compactados no mesmo, como são dependências, o seu projeto pode referenciar o JAR como uma dependência Java.

Esses arquivos que são adicionados ao seu projeto, são denominados como cliente do *Web Service* e compõe um cliente JAX-WS completo. (SAUDATE, 2012)

Os artefatos do cliente que são criados usando a ferramenta WSIMPORT são: Service Endpoint Interface (SEI); Classes dos serviços; Classes de exceção que é mapeado da classe `wsdl:fault`, se houver exceção mapeada; Valores dos tipos gerado por JAXB que são classes Java mapeados dos tipos do schema XML. (THE APACHE, 2015). A partir de que se tem todos esse artefatos, o cliente já consegue comunicar com o serviço do *Web Service*.

2.3.1.2 Handler

Em uma implementação *Web Service*, algumas vezes são necessários monitoramento dos dados que são transmitidos e criação de autenticação para validar a requisição no serviço. Para implementar essas funcionalidades ao *Web Service*, é necessário implementar interceptadores nos serviços que são expostos ou consumidos.

Segundo SAUDATE (2012), em JAX-WS, é possível interceptar as mensagens de entrada e saída dos serviços a partir de *handlers*.

Handlers são interceptadores de mensagem que podem ser facilmente conectados ao JAX-WS para fazer o processamento adicional de entrada e saída de mensagens. No processamento adicional é possível validar, modificar e monitorar os

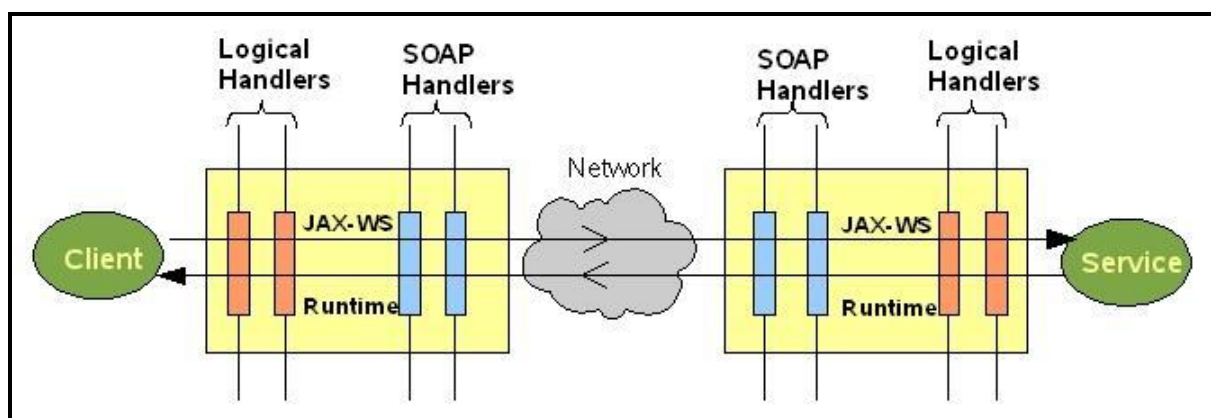
dados que estão sendo transmitidos e recusar a requisição em um caso de autenticação (GLASSFISH, 2015).

JAX-WS define dois tipos de *handlers*, *Logical Handlers* e *Protocol Handlers*. *Protocol Handlers* são especificados para um protocolo e pode ser acessar ou mudar os aspectos do protocolo específico de uma mensagem. *Logical Handlers* são agnósticos de protocolo e não pode mudar quaisquer partes específicas do protocolo (como headers) da mensagem. *Logical Handlers* age somente no *payload* da mensagem. (GLASSFISH, 2015)

Payload de uma mensagem refere-se apenas a parte da mensagem que contem os dados principais que estão sendo transmitidos.

A Figura 3 ilustra o funcionamento e fluxos de entrada e saída dos interceptadores SOAP *Handler*, que é um *Protocol Handler*, e *Logical Handler*.

Figura 3 – Interação com os *handlers* na chamada do serviço.



Fonte: GLASSFISH, 2015.

É possível interceptar a entrada e saída das mensagens tanto no lado do serviço exposto como no cliente que está consumindo o serviço.

Uma mensagem, tanto de entrada como de saída, segue a sequência no fluxo de interceptadores. Quem envia a mensagem vai executar a seguinte sequência de interceptadores: Primeiro executa o *Logical Handler* e depois o *SOAP Handler*. Quem recebe a mensagem, a sequência de execução é primeiro o *SOAP Handler* e depois o *Logical Handler*.

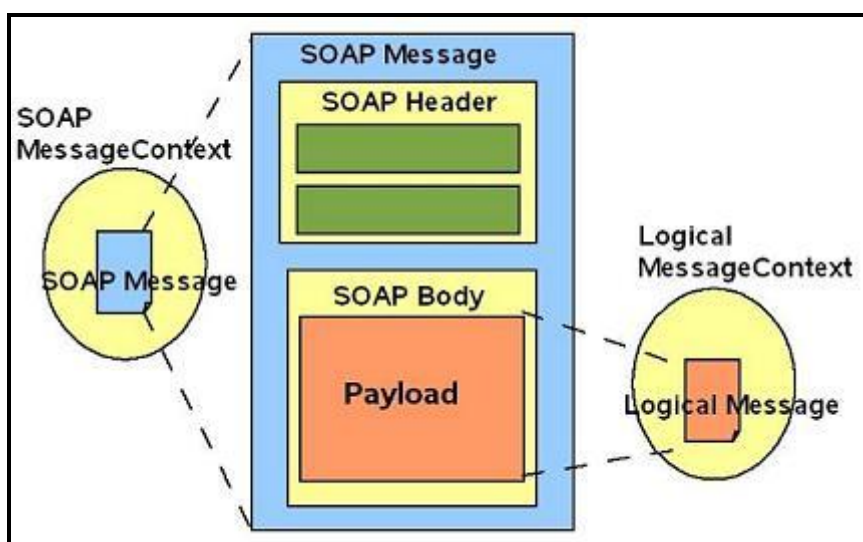
Para acessar o conteúdo das mensagens pelos interceptores, existem as implementações do *MessageContext* para cada *handler*.

A implementação do *MessageContext* para o *SOAPHandler*, é o *SOAPMessageContext* e é possível capturar o conteúdo da mensagem através do método *getMessage* que retornará um objeto do tipo *SOAPMessage* com todo o conteúdo da Mensagem (GLASSFISH, 2015).

A implementação do *MessageContext* para o *LogicalHandler*, é o *LogicalMessageContext* e é possível capturar o conteúdo da mensagem através do método *getMessage* que retornará um objeto do tipo *LogicalMessage* que fornece o *payload* da mensagem (GLASSFISH, 2015).

A Figura 4 mostra o conteúdo que cada *MessageContext* pode acessar e modificar a mensagem.

Figura 4 – Composição dos conteúdos acessados pelos *handlers*.



Fonte: GLASSFISH, 2015.

O *SOAPMessageContext* do *SOAPHandler* abrange um conteúdo maior para ser acessado e modificado do que o *LogicalHandler*, porém o *LogicalHandler* provê mais segurança por acessar só as informações principais que estão sendo transmitidas, e por ser genérico, pode ser usada para qualquer protocolo.

2.3.2 Spring Framework

O *Spring Framework* é um *framework* que se originou a partir de deficiências que o EJB 2.1 apresentava (WEISSMANN, 2012), que veio a ser um substituto mais eficiente, sendo que o *Spring Framework* supriu essas deficiências.

Segundo WEISSMANN, as deficiências que havia no EJB 2.1 eram:

- Excesso de código na implementação e configuração na aplicação que utilizava EJB 2.1 que prejudicava a produtividade de desenvolvimento e gerava falta de conhecimento do desenvolvedor que não sabia qual era a funcionalidade da maioria do código que estava sendo inserido.
- Incompatibilidade de implementações da API do EJB 2.1 entre os Servidores de Aplicação Java.
- Impossibilidade ao criar testes unitários de integração devido ao EJB 2.1 não rodar fora do Servidor de Aplicação.
- Dificuldade de implementação injeção de dependência e Programação Orientada a Aspectos com baixo acoplamento.

Com tantas deficiências e com o surgimento do *Spring*, o EJB 2.1 perdeu competitividade e correu o risco de virar uma API legada (WEISSMANN, 2012). Porém o EJB que já está na versão 3.1, que está presente no Java EE 6, utilizou muitas características presente no *Spring Framework* e voltou a ser tão competitivo quanto ao *Spring Framework*. Entretanto, o *Spring Framework* ainda domina o mercado voltado para desenvolvimento de aplicações que são executadas fora de Servidores de Aplicação Java.

O principal público alvo do *Spring Framework* é o mercado de desenvolvimento de aplicações corporativas para a plataforma Java e é baseado nos conceitos de inversão controle e injeção de dependência (WEISSMANN, 2012), sendo pioneiro na injeção de dependência de forma oculta para o desenvolvedor.

A injeção de dependência utiliza um container *Spring* para gerenciar os objetos instanciados pelo *Spring Framework*.

O *Spring Framework* pode ser executado em qualquer plataforma de desenvolvimento baseado em Java e fornece um amplo modelo de programação e configuração para o desenvolvimento de aplicações (SPRING, 2015).

E além de fornecer tantos recursos auto gerenciáveis, o *Spring Framework* suporta integrações com outros *Frameworks* do *Spring* de forma simples e sem necessidade de muita configuração.

A versão básica do *Spring Framework* é composta por seis componentes: o container de inversão de controle, suporte a AOP, instrumentação, acesso a dados/integração, suíte de testes e *web* (WEISSMANN, 2012). O acesso a dados

engloba o *Spring Framework* JDBC, e o componente *Web* está presente entre os componentes básicos do *Spring Framework* devido a programação Java ser consideravelmente maior na plataforma *web*.

O *Spring Framework* fornece as principais características que a arquitetura de uma aplicação necessita para constar na maioria do código apenas questões relacionadas a regras de negócio.

2.3.2.1 Spring Framework JDBC

O *Spring* disponibiliza uma grande quantidade de frameworks que se integram com o *Spring Framework*. Muitos desses *frameworks* não estão disponíveis nos componentes básicos do *Spring Frameworks*, e esses *frameworks* necessitam de ser adicionado no seu projeto como um componente apartado do componente principal do *Spring*, *Spring Frameworks*.

Porem, como falado anteriormente, alguns *frameworks* já estão incluídos no *framework* principal do *Spring*, e uma delas é o *Spring Framework* JDBC que faz parte dos *frameworks* de acesso a dados.

Segundo WEISSMANN (2012), além do *Spring Framework* fornecer um rico suporte nativo a tecnologias de acesso a dados como JDBC, também é oferecido um suporte a tecnologias como *Hibernate* e *JPA*.

Todos os *frameworks* de acesso a dados do *Spring*, necessitam que seja configurado um *data source* a partir da classe *javax.sql.DataSource* com as informações da conexão com o componente responsável por gerir os dados (WEISSMANN, 2012). Após o *data source* configura e com o objeto criado, é de responsabilidade do *Spring* de gerir as transações.

O principal componente do *Spring Framework* JDBC é o *JdbcTemplate*, que é responsável por repassar para o *framework* as instruções que a aplicação solicitar para ser executada no componente de dados, que vem a ser o banco de dados (WEISSMANN, 2012).

Com o *JdbcTemplate*, o trabalho com SQL de banco de dados relacional e JDBC é fácil (SPRING, 2015), sem a necessidade de muito código que é acrescentado mas não agrega valor para a regra de negócio, como códigos tratativas de transação e preparação e execução de instruções SQL.

Outro componente importante, é o *NamedParameterJdbcTemplate* que torna mais fácil informar os parâmetros da instrução SQL a partir do nomes antecedido por

dois pontos :) ao invés do velho conhecido ponto de interrogação (?) (SPRING, 2015).

A proposta do *Spring Framework* JDBC é propor um modelo clássico de informar instruções SQL de baixo nível, sem muita configuração de abstração de banco, como o JPA, e sem nenhuma tratativa nas instruções passadas para o *framework* (SPRING, 2015). Com pouca configuração, o sistema se torna mais simples e mais acoplado ao banco de dados no qual está sendo executada a aplicação.

Com *JdbcTemplate*, o desenvolvedor não precisa se preocupar com a manipulação da conexão com a obtenção e liberação de recursos devido o *framework* já oferecer essas funcionalidades (WEISSMANN, 2012). O desenvolvedor só deverá se preocupar em colocar em seu código as regra de negócio.

O *JdbcTemplate* permite o uso de *thread safe*, que quando configurado, possibilita que todos os beans que necessitarem desse serviço na aplicação utilize a mesma instância do *JdbcTemplate* (SPRING, 2015), que torna o uso de memória, mais inteligente não criando objetos repetidos desnecessariamente na memória.

Normalmente, códigos que utiliza JDBC são repletos de recursos de aquisição, gerenciamento de conexão, manipulação de exceção, e verificação geral de erro (SPRING, 2015). O objetivo o *Spring Framework* JDBC é remover todo esse código repetido da aplicação, tornando aplicação muito mais simples e legível.

2.4 Design Patterns

Design Patterns, também conhecido como Padrões de Projetos (GUERRA, 2013), é uma forma de deixar a arquitetura de software mais flexível, genérica e desacoplada.

As primeiras linguagens de programação de alto nível foram linguagens de programação estruturada, e como evolução da programação estruturada, surgiram as linguagens de programação orientada a objeto (GUERRA, 2013).

Com as linguagens e programação orientada a objeto, tendo o principal fundamento de abstrair o mundo real, tornando cada objeto uma unidade compostas de outros objetos, percebeu-se que era necessário desenvolver padrões para que as soluções dos problemas seja fácil de ser compreendida e alterada por qualquer desenvolvedor.

Observando essa necessidade, decidiu-se documentar as soluções para problema de modelagem de objeto (GUERRA, 2013), surgindo os padrões de projeto.

Um padrão pode ser considerado uma solução que foi implementada com sucesso de forma recorrente em diferentes contextos (GUERRA, 2013). Sucesso nas implementações e desacoplamento é fundamental para que uma solução seja considerada um padrão de projeto e não estando a um contexto específico, pode ser usado em qualquer lugar, qualquer projeto.

Segundo GAMMA et al (1995), os padrões de projeto são especificações de como comunicar objetos e classes com o intenção de resolver um problema de desenho geral do problema em um contexto particular do problema.

Para se resolver um problema utilizando um ou vários padrões, é necessário ter conhecimento de qual é a solução do problema, e ter um conhecimentos de muitos padrões para conseguir escolher o padrão que se adequa ao problema (GUERRA, 2013).

Segundo GAMMA et al (1995), os padrões de projeto são divididos em três categorias:

- “Padrão de Criação”: é um padrão que tem a proposta de abstrair o processo de instanciação e inicialização de objetos independentes de como foram criados, compostos ou representados. Alguns exemplos de padrões de criação são: *Abstract Factory*, *Builder*, *Singleton* entre outros;
- “Padrão Estrutural”: é um padrão que tem a proposta de abstrair objetos com grandes estruturas com heranças e composições. Alguns exemplos de padrões de criação são: *Composite*, *Facate*, *Proxy* entre outros;
- “Padrão Comportamental”: é um padrão que tem a proposta de abstrair a execução de algoritmos e atribuir responsabilidade entre os objetos. Padrão comportamental não apenas descreve como os objetos são compostos, mas também como eles se comunicam. Alguns exemplos de padrões de criação são: *Command*, *Iterator*, *Observer* entre outros.

É comum que um padrão utilize características de outros padrões, pois há muito genéricos, outros com suas características um pouco mais específicas que busca realizar uma funcionalidade.

2.4.1 Proxy/Decorator

Os padrões *proxy* e *decorator* são padrões do tipo estrutural, que é utilizado em objetos com herança e composição.

Proxy e *decorator* são dois padrões que tem sua estrutura muito parecida, suas diferenças são apenas na questão de motivação e contexto onde os padrões serão utilizados (GUERRA, 2013).

Para a entender como funciona um *proxy/decorator*, é necessário entender dois conceitos da programação orientada a objetos. São elas encapsulamento e polimorfismo.

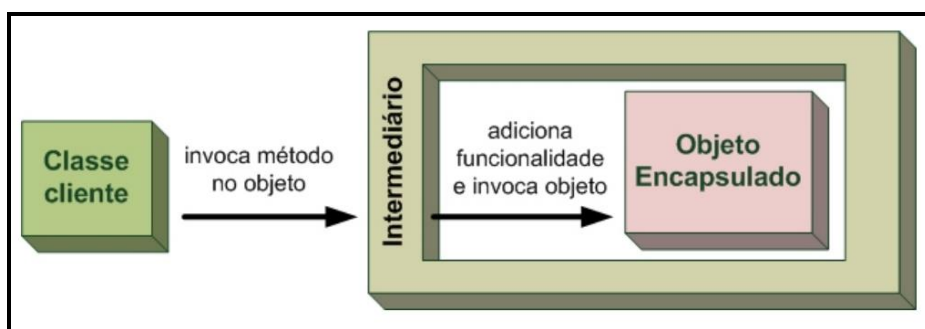
A partir do encapsulamento, o cliente que interagir com esse objeto precisa apenas conhecer a sua interface, abstraindo sua implementação interna (GUERRA, 2013).

Já o polimorfismo permite que uma instância de uma classe possa ser atribuída a uma variável desde que obedeça a abstração utilizada (GUERRA, 2013).

A partir desses conceitos, é possível que uma classe envolva uma instância sem que a cliente saiba que está lidando com outro objeto. Dessa forma, ela servirá como intermediária entre a classe cliente e a classe alvo. Com isso, ela obtém o controle da execução antes e depois de cada invocação de método. Isso permite que, por exemplo, funcionalidades sejam adicionadas e que validações sejam realizadas (GUERRA, 2013).

A Figura 5, mostra de maneira gráfica a arquitetura e o funcionamento um *proxy/decorator*.

Figura 5 – Envolvendo um Objeto *Proxy/Decorator*.



Fonte: GUERRA, 1997.

O “objeto encapsulado”, como descrito na Figura 5, representa o objeto com que armazena os dados do objeto e suas ações, que no Java são representados por métodos. Envolvendo o objeto “objeto encapsulado”, há o objeto “intermediário”, que nada mais é do que o *proxy/decorator*, que adicionará funcionalidades extras quando as ações dos métodos encapsulados forem chamadas. A “classe cliente” é a classe que interage com o *proxy/decorator* da mesma forma que interagiria com o “objeto encapsulado”.

O propósito do *proxy/decorator*, é interagir com a “classe cliente”, fazendo pensar que ela está lidando direto com o “objeto encapsulado” (GUERRA, 2013), como se não houvesse nenhum intercessor entre a “classe cliente” e o “objeto encapsulado”.

Segundo GAMMA et al (1995), o *proxy/decorator* fornece um substituto para controlar suas ação com outros objetos que acessa-lo.

Uma situação no qual o *proxy/decorator* pode trazer vantagens ao projeto e tardar o máximo a criação de um objeto, e cria-lo apenas quando for usar (GUERRA, 2013), essa situação é utilizada muitas vezes para objetos que consomem muitos recursos para a sua criação, dessa forma, não há instâncias desnecessárias de objetos que consomem muito recurso,

As diferenças conceituais entre *proxy* e *decorator* estão relacionadas as suas motivações. As motivações que pode permitir a implantação de um *decorator* está relacionado à adição de funcionalidades, quando se adiciona funcionalidades, a ideia que o *decorator* passa é realmente de decorar uma implementação, como expressa seu nome em inglês (GUERRA, 2013). Já as motivações de um *proxy* é de servir como intermediário entre o objeto cliente e o objeto principal de forma transparente de forma a manter a segurança no acesso do objeto principal (GUERRA, 2013).

2.5 Integração

Integração é um conceito que define a união de duas unidades distintas. Na área de desenvolvimento de sistemas, temos diversos tipos de integração, entre algumas delas, temos:

- “Integração entre sistemas”: são integrações entre sistemas desacoplados, como sistemas de empresas diferentes;

- “Integração de microssistemas”: são integrações de módulos independentes de um sistema, como por exemplo, integrar microssistemas de recursos humanos, financeiro, produção e *marketing*;
- “Integração de componentes externos”: são integrações com outras ferramentas que não estão acopladas diretamente ao sistema, como banco de dados, sistemas de arquivos, GPS entre outros;
- “Integração de componentes internos”: são integrações entre componentes no sistema, como *frameworks* e rotinas.

Todo sistema tem integrações de diversos tipos, melhorar a qualidade no qual essas integrações são feitas, a fim de deixar o sistema mais desacoplado, permite melhor manutenibilidade, flexibilidade e maior confiança no processamento.

2.5.1 Integração de Web Service com SOAP Message Handler

Como dito anteriormente, *SOAP Message Handler* é uma forma de interceptar as informações que são trafegadas entre o serviço e o cliente do *Web Service*.

Para implementar essa funcionalidade no cliente do *Web Service*, é preciso realizar uma alteração na classe que é herdada da classe *javax.xml.ws.Service* adicionando um metadata com a anotação *javax.jws.HandlerChain* como vemos na Figura 6.

Figura 6 – Código de adição do *handler* em um Cliente de *Web Service*.

```
@HandlerChain(file = "config/handler-chain.xml")
@WebServiceClient(name = "AtualizacaoCadastralWebServiceService")
public class AtualizacaoCadastralWebServiceService
    extends Service {

    private final static URL ATUALIZACAOCADASTRALWEBSERVICESERV
    private final static WebServiceException ATUALIZACAOCADASTRAL
    private final static QName ATUALIZACAOCADASTRALWEBSERVICESE
        "AtualizacaoCadastralWebServiceService");
```

Fonte: Autoria própria, 2015.

Além de fazer a anotação, é necessário criar ajustar a classe *handler* com as rotinas de interceptação, e criar um XML com as configurações dos *handlers*

que faram as interceptações do consumo do *Web Service* e que estará no atributo “*file*” da anotação *javax.jws.HandlerChain*.

Entretanto essa classe é uma classe gerada pelo comando *WSIMPORT* do *JAX-WS*, e altera-lo colocando a anotação *javax.jws.HandlerChain*, não é muito interessante devido à perda dessa alteração em caso de geração do mesmo cliente após alteração. Isso gera retrabalhos desnecessários, prejudicando a produtividade de desenvolvimento.

Para que não haja retrabalho em caso de nova geração do mesmo cliente, é interessante a criação de um *proxy*. O *proxy* faz com que os códigos referentes à configuração dos *handlers* sejam desacoplados dos códigos do cliente do *Web Service* gerado pelo *WSIMPORT*.

Então, para implementar o *proxy* e desacoplar da classe que estende a classe *javax.xml.ws.Service*, é necessário uma nova classe estendendo-a na classe que estende o *javax.xml.ws.Service* que foi gerado pelo *WSIMPORT* e adicionar a anotação nessa classe como exemplifica a Figura 7.

Figura 7 – Código de adição do handler em um Cliente de *Web Service* com *Proxy*.

```
@HandlerChain(file = "config/handler-chain.xml")
public class AtualizacaoCadastralWebServiceServiceProxy
    extends AtualizacaoCadastralWebServiceService {
}
```

Fonte: Autoria própria, 2015.

Apenas criar o *proxy*, não vai tornar automaticamente o uso desse *proxy* como padrão no consumo do *Web Service*.

Na Figura 8, é possível identificar como funciona a inicialização do objeto do cliente do *Web Service* para poder consumir o serviço do *Web Service* sem o *proxy*, apenas instanciando a classe que estende da classe *javax.xml.ws.Service*.

Figura 8 – Simples instanciação de um Cliente *Web Service*.

```
public AtualizacaoCadastralWebService  
getWSAtualizacaoCadastralWebService() {  
  
    return new AtualizacaoCadastralWebServiceService().  
               getAtualizacaoCadastralWebServicePort();  
  
}
```

Fonte: Autoria própria, 2015.

Então, quando é instanciado um cliente do *Web Service*, é necessário instanciar o *proxy* no lugar da classe que estende *javax.xml.ws.Service* e usar os métodos da sua classe pai, já que o *proxy* só tem a anotação.

Na Figura 9, pode-se observar que a instanciação no *proxy*, e usado de forma transparente deixando a impressão de que o não há *proxy* e como se estivesse trabalhando diretamente na classe cliente do *Web Service*.

Figura 9 – Instanciação de um *Proxy* de Cliente *Web Service*.

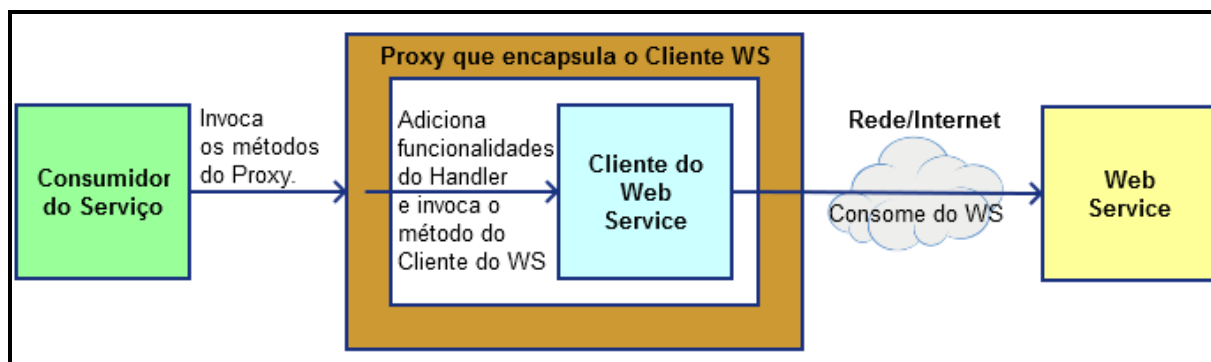
```
public AtualizacaoCadastralWebService  
getWSAtualizacaoCadastralWebService() {  
  
    return new AtualizacaoCadastralWebServiceServiceProxy().  
               getAtualizacaoCadastralWebServicePort();  
  
}
```

Fonte: Autoria própria, 2015.

Após estar instanciados o *proxy*, as classes que irão utiliza-los para consumir, o *Web Service* não terão alteração de código, isto é, pode haver muitas partes no projeto que utilizam esse consumo que não haverá implicação nenhuma nesses códigos, já que ele continua tratando o cliente do *Web Service* como o original criando pelo *WSIMPORT*.

A Figura 10 exemplifica de forma gráfica como o funcionamento do *proxy* no ambiente citado para adicionar as funcionalidade do *handler* de interceptação no cliente do *Web Service*.

Figura 10 – Aplicação do *handler* com *Proxy* em um Cliente de *Web Service*.



Fonte: Autoria própria, 2015.

Na Figura 10, há o “Consumidor do Serviço” abstrai todos os objetos que trabalharam com a solicitação de requisição do cliente do *Web Service*, e trabalha com a resposta desse serviço. O “Cliente do *Web Service*” é o componente que comunica via rede ou/e *Internet* com o serviço exposto do “*Web Service*”. O “*Proxy* que encapsula o cliente do *Web Service*” é o objeto que tem as funcionalidades adicionais de interceptação do *Web Service* e execução das rotinas do *handler*.

2.5.2 Integração de *Web Service* com *Spring Framework JDBC*

Como mostrado anteriormente, *Spring Framework JDBC* é um *framework* que auxilia na integração entre o banco de dados com os objetos no Java.

A proposta na integração do *Web Service* com *Spring Framework JDBC*, é permitir que os dados já sejam recuperados do banco de dados preenchendo os objeto dos artefatos do cliente do *Web Service* de forma que ele consuma o serviço sem intervenção das alterações das classes que preencher esse objeto.

Porém o *Spring Framework JDBC* não lida muito bem com alguns tipos de variáveis, como *javax.xml.datatype.XMLGregorianCalendar*, e também não consegue preencher objetos compostos na classe do artefato.

Então, deve criar a classe *proxy* dessas classes com o objetivo de preencher esses tipos de valores, se houver, que será atribuído as variáveis da classe direto pelo *Spring Framework JDBC*.

Para criar a classe *proxy* é necessário apenas criar uma classe herdando da classe do artefato do cliente do *Web Service*, e criar sobrecargas de métodos de atribuição de valores para fazer conversões de tipos de atributos diferentes ou/e criar métodos para atribuição dos atributos compostos.

A Figura 11 mostra uma implementação de sobrecarga com uma conversão de *Date*, que é o tipo que o *Spring Framework JDBC* retorna ao recuperar do Banco de dados para *XMLGregorianCalendar* que é o tipo que o *WSIMPORT* gera.

Figura 11 – Sobrecarga de um método com tipo não compatível com *Web Service*.

```
public void setDataNascimentoDate(Date dataNascimento) {
    if (super.getDataNascimento() == null) {
        super.setDataNascimento(new br.com.planosaude.batch.client.atualizacaocadastral.Date());
    }
    super.getDataNascimento().setDate(DateUtils.parseXMLGregorianCalendar(dataNascimento));
}
```

Fonte: Autoria própria, 2015.

Nesse código, as lógicas de conversão, estão na rotina do método *DateUtils.parseXMLGregorianCalendar()*.

Outro método que pode ser escritos em um *proxy* do artefato do cliente do *Web Service* é a atribuição de valores aos objetos compostos que na Figura 12 é mostrado.

Figura 12 – Método de atribuição de valor em objeto composto.

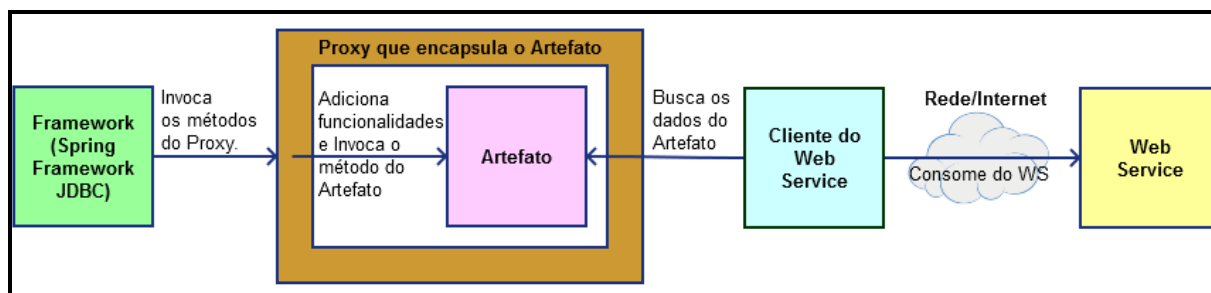
```
public void setTelefoneString(String telefone) {
    if (super.getTelefone() == null) {
        super.setTelefone(new Telefone());
    }
    this.getTelefone().setTelefone(telefone);
}
```

Fonte: Autoria própria, 2015.

Nesse caso, no objeto do *proxy* há um atributo com o objeto complexo do telefone com o atributo *telefone* de tipo *String*. Dessa forma não haveria necessidades de query adicionais nem rotinas de atribuição de valores dessa *query* para esses objetos após a consulta.

O gráfico da Figura 13 abstrai a visão de como seria a relação do *proxy* com seu componente encapsulado interagindo com os outros componentes como o *Spring Framework JDBC* e o Cliente do *Web Service*.

Figura 13 – Aplicação do *Proxy* em um Artefato do Cliente do *Web Service*.



Fonte: Autoria própria, 2015.

O “*Proxy* que encapsula o Artefato” intercepta as chamadas do “*Spring Framework JDBC*” e adiciona novas funcionalidade, seja de conversão de tipos de variáveis como atribuição a valores compostos, e dessa forma preenche todo o “Artefato”. Após o “Artefato” estiver preenchido, o “Cliente do *Web Service*” recupera esse artefato com os dados preenchidos, e transmite esses dados para o “*Web Service*” via rede e/ou *Internet*.

Essa prática pode ser utilizada em qualquer rotina que integra o banco dados, por meio do *Spring Framework JDBC* e o cliente *Web Service*.

2.6 Modelagem de Dados

O conceito de modelagem de dados tem o objetivo de definir uma estrutura de como os dados são estruturados em um repositório de dados.

Segundo ELMASRI e NAVATHE (2011), Modelagem de dados é uma técnica tradicional de concentrar nas estruturas e restrições de banco de dados durante seu projeto conceitual.

A partir dos dados coletados na análise de requisitos, é necessário desenvolver a análise dos dados que serão armazenados, e nessa análise, o propósito é definir uma abstração de como os dados serão armazenados no repositório. Essa abstração tem como propósito criar uma estrutura que poderá ser implementada em qualquer repositório de banco de dados.

Muitas vezes, os dados não costuma estar de forma clara para estruturar, e nem havendo nomes com fácil definição de suas funcionalidades.

Ao projetar um esquema de banco de dados, a escolha de nomes para tipos de entidade, atributos, tipos de relacionamentos e (particularmente) funções nem sempre é simples. É preciso escolher nomes que transmitam tanto quanto o possível, os significados conectados às diferentes construções no esquema (ELMASRI e NAVATHE, 2011).

Então, para abstrair o entendimento das estruturas de dados para, existem várias modelos e diagramas para facilitar a definição e documentação da estrutura dos dados. Dentre os modelos disponíveis temos alguns, entre eles temos: Modelo entidade-relacionamento, diagrama entidade-relacionamento, diagrama de classes, entre outros.

2.6.1 Diagrama Entidade-Relacionamento

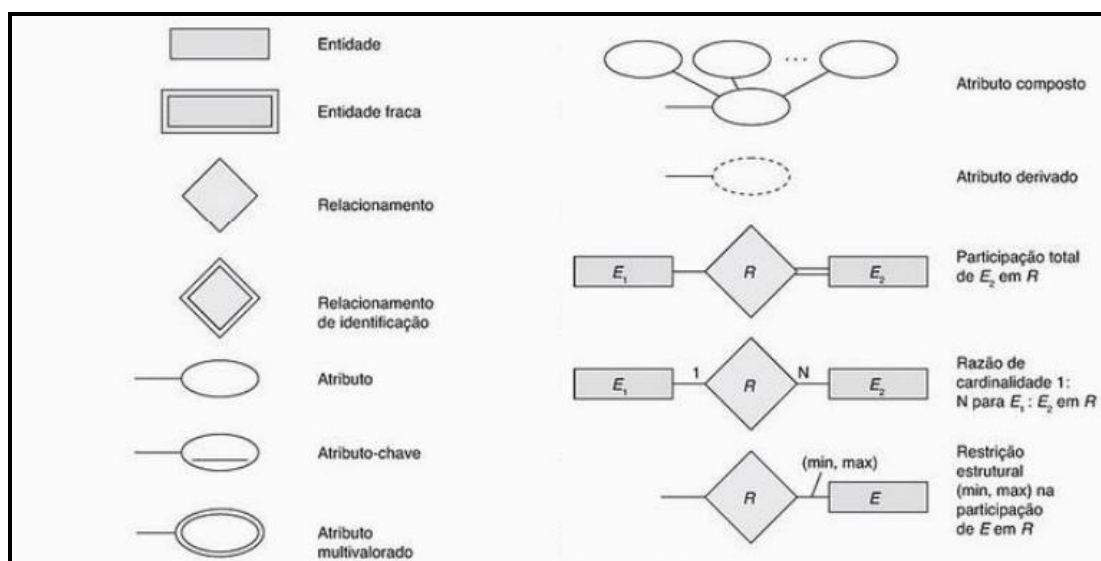
O diagrama entidade-relacionamento, mais conhecido como DER, é um diagrama que tem o propósito de abstrair a organização estrutural dos dados de uma aplicação.

DER é um diagrama muito popular e muito usado em projetos conceitual de aplicações de banco de dados (ELMASRI, NAVATHE, 2011). Por ter sua definição gráfica simples tendo formas geográficas simples, se tornou muito popular para definir e documentar estruturas de bancos.

Com o DER, definem-se entidades, atributos das entidades, e relacionamentos entre entidades de forma simplificada e genérica, sem depender de um banco específico, sendo assim, podendo ser usado em qualquer banco de dados e qualquer ambiente.

Na Figura 14, há um explicação de cada componente usado no diagrama entidade-relacionamento.

Figura 14 – Componentes do diagrama entidade-relacionamento



Fonte: ELMASRI e NAVATHE, 2011.

A entidade é representada por um retângulo simples, e em seu interior é composto por seu nome, sendo um substantivo no singular com todas as letras em maiúsculo (ELMASRI e NAVATHE, 2011). A entidade representa uma tabela no banco de dados, onde será gravado um conjunto de dados.

A entidade fraca é representada por um retângulo com uma borda em linha dupla. Apesar de ser uma entidade fraca, tem as mesmas características que tem a entidade simples. A entidade fraca também é uma tabela, só que é uma tabela que depende da outra principal. Se não houver registros na entidade principal, não haverá motivo para existir na entidade fraca, já que eles estariam totalmente dependentes desse registro da tabela principal.

O relacionamento é representado por um losango, a partir do losango é possível abstrair o relacionamento entre entidades. Em seu interior deve constar um verbo com todas as letras em maiúsculas mostrando a forma que uma entidade interage com a outra (ELMASRI e NAVATHE, 2011). O relacionamento é uma abstração de uma chave estrangeira no banco de dados.

O relacionamento de identificação é representado por um losango com uma borda em linha dupla, e tem as mesmas características que o componente de relacionamento. Sua única diferença é que ele é usado para relacionar uma entidade principal com uma entidade fraca.

O atributo é representado por um círculo (algumas vezes sendo oval), em seu interior deve ser informado seu nome que é composto por um substantivo que tem sua primeira letra em maiúscula e as demais letras em minúsculas (ELMASRI e NAVATHE, 2011). Um atributo abstrai uma coluna de uma tabela em um banco de dados.

O atributo-chave é representado por um círculo e é bem semelhante ao atributo, tendo características iguais, porém com a diferença de que o atributo chave tem seu texto em seu interior sublinhado. Para o banco de dados, o atributo-chave abstrai uma coluna que é a chave primária da tabela.

O atributo multivalorado é formado por um círculo com uma borda com linha dupla, tendo suas outras características iguais ao do atributo. O atributo multivalorado é um atributo que contém um conjunto de informações em um mesmo atributo.

O atributo composto é representado por um círculo ligando outros demais círculos. O círculo central representa o conjunto de atributos que ele está ligando.

Para o banco de dados, o círculo central não existe, é como se esses atributos que ligam esse atributo central estivessem ligados diretamente na entidade.

O atributo derivado é representado por círculo com a borda tracejada. Esse atributo simboliza um valor que pode ser obtido com os outros atributos. Pode ser que ele nem exista no banco de dados, porém deve haver todos os atributos necessários para a sua obtenção.

A participação total é representada por uma linha dupla que interliga uma entidade a um relacionamento. Essa ligação simboliza a ligação entre uma entidade e outra entidade fraca, diferentemente da linha simples que simboliza a união entre as entidades.

A cardinalidade é representado por N ou 1. Sendo assim, em um relacionamento entre duas tabelas, há três possibilidades.

- 1 x 1: representa quem apenas um registro de uma entidade será ligado com outro de outra entidade com quem se relaciona;
- 1 x N: representa que um registro de uma entidade pode ligar com um ou mais registros da outra entidade com quem se relaciona;
- N x N: representa que um ou mais registros de entidade pode se relacionar com um ou mais registros da outra entidade com quem se relaciona.

A restrição estrutural na participação é representada pela expressão “(mínimo, máximo)”, onde o mínimo é a quantidade de registros mínima que deve existir para que o relacionamento seja adequado, e o máximo é a quantidade de registros máxima que pode existir para que o relacionamento seja adequado.

A partir de todos esses componentes, é possível montar um diagrama entidade-relacionamento completo abstraindo toda a estrutura de dados da aplicação no qual o banco está sendo desenvolvido.

3 PROCEDIMENTOS METODOLÓGICOS

Desenvolvimento e sistemas de pesquisas bibliográficas sobre a linguagem Java, conceitos de *Web Service* e SOAP, conceitos e formas de uso de *frameworks* como JAX-WS, *Handlers* (que está incluído ao JAX-WS), *Spring Framework* e *Spring Framework JDBC*.

Aplicação de integrações entre *Web Service* com JAX-WS e *Handlers* e entre *Web Service* com JAX-WS e *Spring Framework JDBC* e verificação dos resultados de produtividade e integridade alcançados.

Desenvolvimento de três projetos para demonstração de como seria em um caso real onde um sistema local grava informações para ser sincronizado em outro sistema *web*, e essa sincronização entre os sistemas local e *web* ocorre por intermédio de outro projeto *batch*.

4 DESENVOLVIMENTO DO PROJETO “PLANO DE SAÚDE”

O Projeto “Plano de Saúde” é composto por três sistemas, um com sistema que local que é roda direto na JRE sistema operacional do computador do usuário, um sistema *web* que roda no navegador, e executado em um servidor de aplicação Java, e um sistema *batch* que tem sincroniza os dados entre o sistema local e o sistema *web*.

O foco dos três sistemas é na realização de adição e alteração dos conveniados e dependentes do Plano de Saúde.

O código do sistema está aberto e pode ser baixado no link: “<https://github.com/FernandoTAA/TCC-ADS>”.

4.1 Sistema “Plano de Saúde” – Plataforma *Web*

O sistema “Plano de Saúde” na plataforma *web* tem o propósito de centralizar todos os conveniados do plano de saúde que são cadastrados nas instalações locais do sistema “Plano de Saúde” na plataforma *desktop* (sistema local).

Além de centralizar todos os dados dos conveniados do plano de saúde em bancos de dados, o sistema da plataforma *web* tem um *Web Service* para receber novos conveniados para inclusão e conveniados já cadastrados para realizar a edição.

4.1.1 Diagrama Entidade-Relacionamento

A modelagem do sistema “Plano de Saúde” na plataforma *web* é composta de quatro entidades. Essas entidades compõe toda a parte de cadastro dos conveniados.

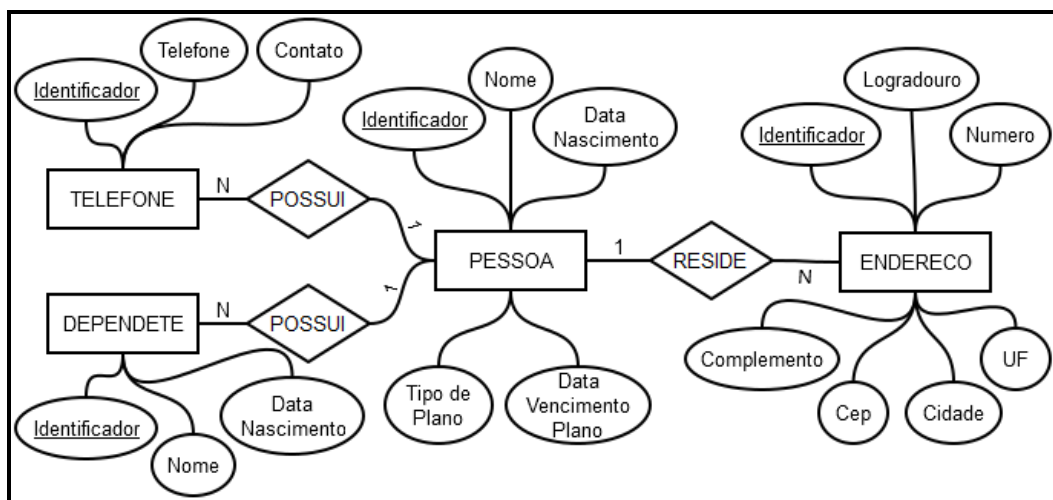
As entidades disponibilizadas para manter os dados centralizados no Sistema “Plano de Saúde” – Plataforma Web são:

- “PESSOA”: é a entidade que armazena as informações de uma pessoa conveniada. Os atributos da entidade são:
 - “Identificador”: é o código da entidade PESSOA e a chave primária da tabela;
 - “Identificador da entidade ENDERECO”: é o código da entidade ENDERECO e a chave estrangeira ligando PESSOA com ENDERECO;
 - “Identificador da entidade TELEFONE”: é o código da entidade TELEFONE e a chave estrangeira ligando PESSOA com TELEFONE;
 - “Nome”: é o nome da pessoa conveniada;
 - “Data de Nascimento”: é a data de nascimento da pessoa conveniada;
 - “Tipo do Plano”: é o tipo de plano no qual a pessoa conveniada possui;
 - “Data de Vencimento do Plano”: é a data de vencimento do plano da pessoa conveniada.
- “ENDERECO”: é a entidade que armazena o endereço do conveniado. Os atributos da entidade são:
 - “Identificador”: é o código da entidade ENDERECO e a chave primária da tabela;
 - “Logradouro”: é o nome da rua/avenida/travessa/prça do endereço da pessoa conveniada;
 - “Número”: é o número da residência da pessoa conveniada;
 - “Complemento”: é o complemento que identifica a residência da pessoa conveniada;

- “Cep”: é o código de endereçamento postal do endereço da pessoa conveniada;
- “Cidade”: é a cidade do endereço da pessoa conveniada;
- “UF”: é a unidade federativa (Estado) do endereço da pessoa conveniada;
- “TELEFONE”: é a entidade que armazena o telefone do conveniado e as informações de contato. Os atributos da entidade são:
 - “Identificador”: é o código da entidade TELEFONE e a chave primária da tabela;
 - “Telefone”: é o número de telefone ligado à pessoa conveniada;
 - “Contato”: é o nome pessoa responsável pelo telefone cadastrado;
- “DEPENDENTE”: é a entidade que armazena os dados dos dependentes relacionados a um conveniado. Os atributos da entidade são:
 - Identificador: é o código da entidade DEPENDENTE e a chave primária da tabela;
 - Identificador da entidade PESSOA: é o código da entidade PESSOA e a chave estrangeira ligando DEPENDENTE com PESSOA;
 - Nome: é o nome da dependente que está ligado a pessoa conveniada;
 - Data de Nascimento: é a data de nascimento do dependente que está ligado a pessoa conveniada.

Na Figura 15, o Diagrama Entidade-Relacionamento é mostrado de forma gráfica.

Figura 15 – DER do Sistema “Plano de Saúde” – Plataforma Web.



Fonte: Autoria própria, 2015.

Nesse DER, a entidade PESSOA é a entidade principal e a se concentra todas as rotinas do sistema.

4.1.2 Interfaces

A única interface que o sistema na plataforma web tem é uma listagem com todos os conveniados do plano de saúde cadastrados.

A figura 16 mostra essa tela de listagem com todos conveniados do plano de saúde que estão cadastrados.

Figura 16 – Tela de Consulta do “Plano de Saúde” – Plataforma Web.

Id	Nome	Tipo de Plano	Data de Vencimento do Plano	Cidade	UF
5	Jairany Nunes	INDIVIDUAL	31/01/2016	Rio de Janeiro	RJ
2	Fernando Teixeira Alves de Araujo	INDIVIDUAL	31/12/2015	São Paulo	SP
3	Fagner Teixeira Alves de Ajaujo	EMPRESA	01/01/2015	Mococa	SP
4	Fabiano Teixeira Alves de Araujo	FAMILIA	05/01/2016	Mococa	SP

Fonte: Autoria própria, 2015.

Nessa listagem há apenas algumas colunas para ser observada, são elas: Identificador, Nome do Conveniado, Tipo de Plano (Individual, Empresarial e Familiar), Data de Vencimento do Plano, Cidade e UF da residência do Conveniado.

A partir desses dados é possível ter um resumo dos conveniados que estão cadastrados no sistema.

4.2 Sistema “Plano de Saúde” – Plataforma *Desktop*

O sistema “Plano de Saúde” na plataforma *desktop* tem o propósito de possibilitar que os responsáveis por realizar consultas e cadastros dos conveniados do plano de saúde.

4.2.1 Diagrama Entidade-Relacionamento

A modelagem do sistema “Plano de Saúde” na plataforma *desktop* é composta de quatro entidades. Essas entidades compõe toda a parte de cadastro dos conveniados. Os registros inseridos permaneceram apenas local até que ocorra a integração com o sistema *web* que está nas rotinas do sistema *batch*.

As entidades disponibilizadas para manter os dados salvo temporariamente no Sistema “Plano de Saúde” – Plataforma Web até a execução da rotina batch são:

- “PESSOA”: é a entidade que armazena as informações de uma pessoa conveniada. Os atributos da entidade são:
 - “Identificador”: é o código da entidade PESSOA e a chave primária da tabela com geração automática de código;
 - “Identificador da entidade ENDERECO”: é o código da entidade ENDERECO e a chave estrangeira ligando PESSOA com ENDERECO;
 - “Identificador da entidade TELEFONE”: é o código da entidade TELEFONE e a chave estrangeira ligando PESSOA com TELEFONE;
 - “Nome”: é o nome da pessoa conveniada;
 - “Data de Nascimento”: é a data de nascimento da pessoa conveniada;

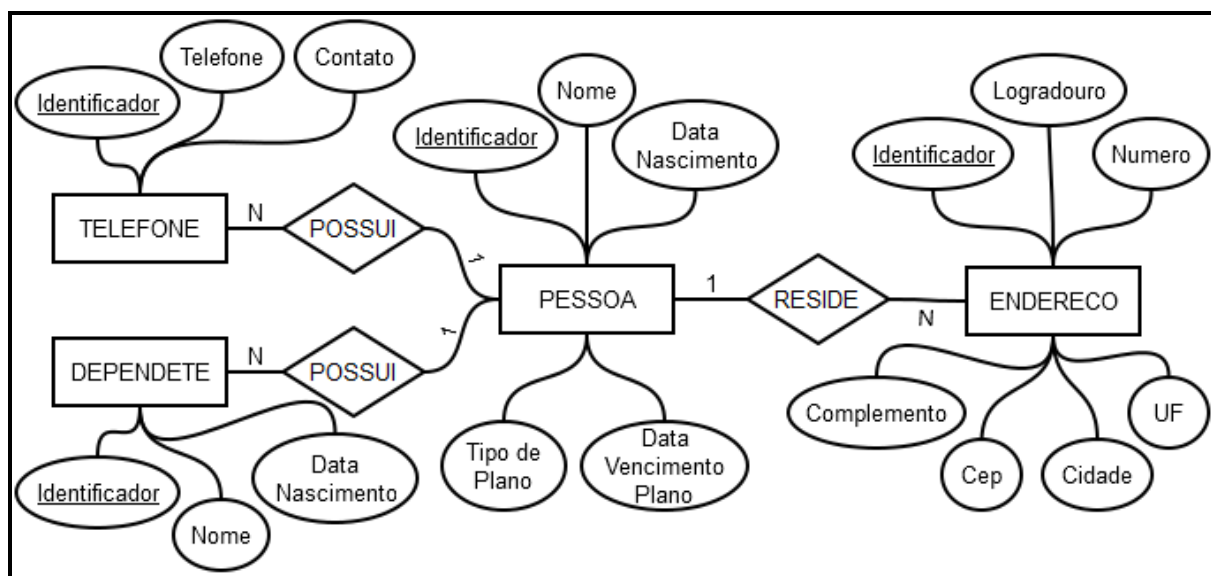
- “Tipo do Plano”: é o tipo de plano no qual a pessoa conveniada possui;
- “Data de Vencimento do Plano”: é a data de vencimento do plano da pessoa conveniada.
- “ENDERECO”: é a entidade que armazena o endereço do conveniado. Os atributos da entidade são:
 - “Identificador”: é o código da entidade ENDERECO e a chave primária da tabela com geração automática de código;
 - “Logradouro”: é o nome da rua/avenida/travessa/praça do endereço da pessoa conveniada;
 - “Número”: é o número da residência da pessoa conveniada;
 - “Complemento”: é o complemento que identifica a residência da pessoa conveniada;
 - “Cep”: é o código de endereçamento postal do endereço da pessoa conveniada;
 - “Cidade”: é a cidade do endereço da pessoa conveniada;
 - “UF”: é a unidade federativa (Estado) do endereço da pessoa conveniada;
- “TELEFONE”: é a entidade que armazena o telefone do conveniado e as informações de contato. Os atributos da entidade são:
 - “Identificador”: é o código da entidade TELEFONE e a chave primária da tabela com geração automática de código;
 - “Telefone”: é o número de telefone ligado à pessoa conveniada;
 - “Contato”: é o nome pessoa responsável pelo telefone cadastrado;
- “DEPENDENTE”: é a entidade que armazena os dados dos dependentes relacionados a um conveniado. Os atributos da entidade são:

- Identificador: é o código da entidade DEPENDENTE e a chave primária da tabela com geração automática de código;
- Identificador da entidade Pessoa: é o código da entidade PESSOA e a chave estrangeira ligando DEPENDENTE com PESSOA;
- Nome: é o nome da dependente que está ligado a pessoa conveniada;
- Data de Nascimento: é a data de nascimento do dependente que está ligado a pessoa conveniada.

Na Figura 17, o Diagrama Entidade-Relacionamento é mostrado de forma gráfica.

Caso ocorra a desinstalação do sistema local, todos os registros que não foram sincronizados são perdidos.

Figura 17 – DER do Sistema “Plano de Saúde” – Plataforma *Desktop*.



Fonte: Autoria própria, 2015.

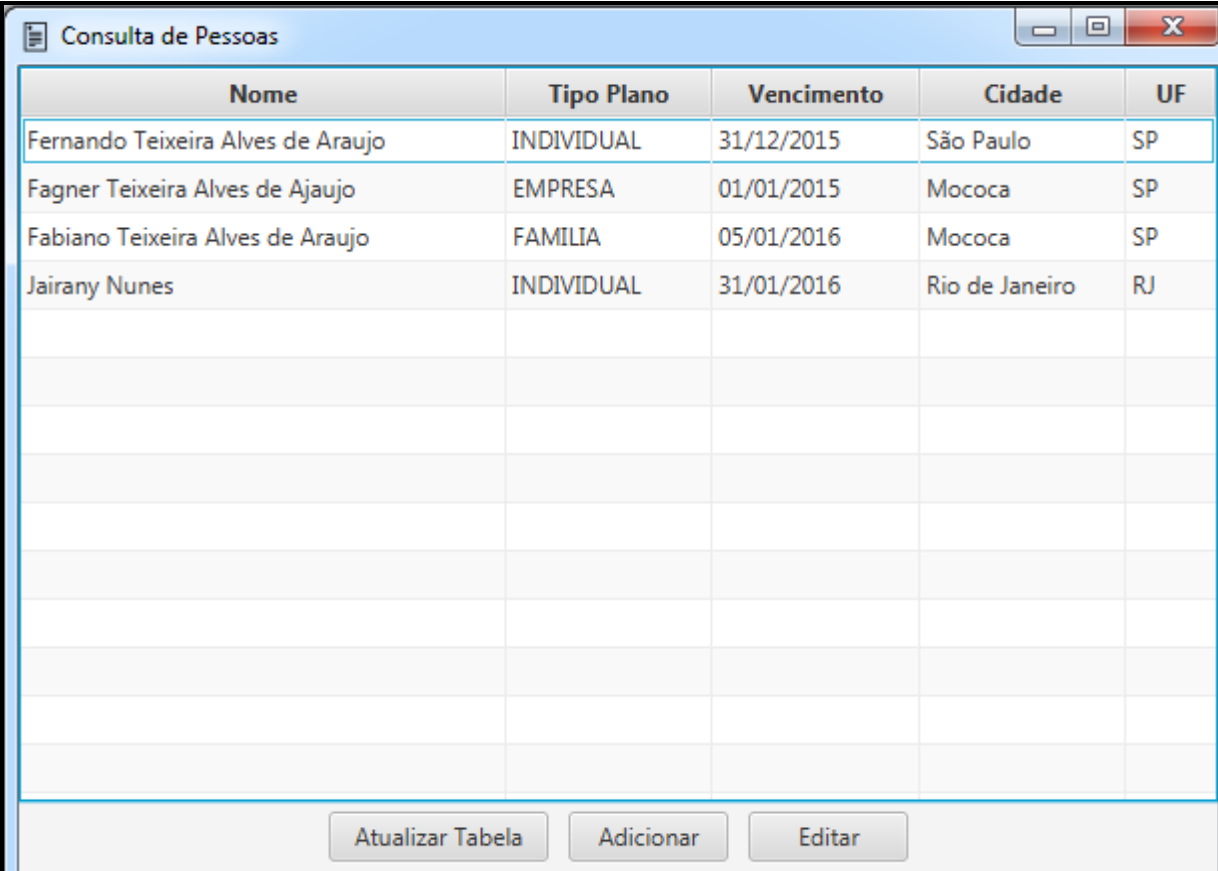
Nesse DER, a entidade PESSOA é a entidade principal e a se concentra todas as rotinas do sistema.

4.2.2 Interfaces

A primeira tela tem a funcionalidade de consultar as pessoas conveniadas cadastradas trazendo todas, e dando opções de começar um no processo de inclusão de uma pessoa nova, quando clicado no botão adicionar, e editar uma pessoa existente, após ser selecionado.

Na Figura 18, há a imagem da tela de consulta do sistema local.

Figura 18 – Tela de Consulta do “Plano de Saúde” – Plataforma *Desktop*.



Nome	Tipo Plano	Vencimento	Cidade	UF
Fernando Teixeira Alves de Araujo	INDIVIDUAL	31/12/2015	São Paulo	SP
Fagner Teixeira Alves de Ajaujo	EMPRESA	01/01/2015	Mococa	SP
Fabiano Teixeira Alves de Araujo	FAMILIA	05/01/2016	Mococa	SP
Jairany Nunes	INDIVIDUAL	31/01/2016	Rio de Janeiro	RJ

Atualizar Tabela Adicionar Editar

Fonte: Autoria própria, 2015.

Como falado, essa tela faz parte do caminho para adição e edição de uma nova pessoa e que faz o usuário, que está preenchendo, sair da tela de consulta e ir para a tela de cadastro.

A Figura 19 mostra a tela de cadastro de uma nova pessoa ou de uma pessoa já incluída ou que foi selecionada para edição.

Figura 19 – Tela de Cadastro/Edição do “Plano de Saúde” – Plataforma *Desktop*.

Cadastro de Pessoas

Código: 2

Dados Gerais | Endereço | Telefone

Nome: Fernando Teixeira Alves de Araujo Nascimento: 03/05/1988

Tipo Plano: INDIVIDUAL Vencimento: 31/12/2015

Dependentes

Dependente	Data de Nascimento
Fagner Teixeira Alves de Araujo	25/01/1984
Fabiano Teixeira Alves de Araujo	14/01/1982

Adicionar Editar Remover

Salvar Cancelar

Fonte: Autoria própria, 2015.

Na tela de cadastro há o campo código desabilitado, devido a ter a geração automática. Há três abas referentes às informações gerais, endereço e telefone da pessoa conveniada. Além disso, há uma grid para o cadastro de dependentes, com opções de adicionar um novo, e editar e remover o dependente selecionado.

Ao salvar os dados na tela de cadastro, os dados são persistidos no banco de dados, ficando disponível para sistema *batch* sincroniza-los para o sistema *web*.

4.3 Sistema “Plano de Saúde” – Integração

O sistema “Plano de Saúde” responsável pela integração do sistema *web* e *desktop*. É um sistema com uma rotina única que é executada de tempos automaticamente sem intervenção do usuário.

Por executar uma rotina de processamento, o sistema de integração é feita a partir do conceito de *batch*, onde cada execução tem início e fim e pode ser executada uma ou mais vezes ao dia.

Para realizar a sincronização, o sistema *batch* tem acesso ao banco de dados local, então, ele lê todas as pessoas que estão inseridas no banco de dados e transmite para via *Web Service* SOAP para o serviço exposto no sistema *web*. Como todos os dados são enviados, o serviço que estão no sistema verifica se essa pessoa já está cadastrada, e se não estiver, o sistema insere, senão, o sistema altera os dados para os novos dados que estão preenchidos no objeto vieram do *Web Service*.

Não há problemas de o sistema *batch* rodar diversas vezes em um dia, pois ele sempre vai incluir ou alterar os dados, não existindo fluxo alternativo a esses.

5 RESULTADOS E DISCUSSÃO

Os resultados ocorreram e se mostraram satisfatórios no desenvolvimento do projeto “Plano de saúde”.

No projeto, foi criado um cenário simples, porém mais próximo dos sistemas reais. Para tornar o cenário mais realista, foram criados dois sistemas independentes e um sistema responsável pela integração, que é onde está o foco de nosso trabalho.

No sistema de integração, temos um cenário de comunicação com *Web Service* SOAP com a utilização de JAX-WS. Nessa integração utilizamos dois *frameworks*, um para conectar ao banco de dados que é o *Spring Framework* JDBC e o outro que é um *framework* interno do JAX-WS que são os *Handlers*.

Nesses cenários, o principal objetivo era ganhar produtividade de desenvolvimento de manutenção em caso ocorra alteração de leiaute dos artefatos e mudanças das estruturas do serviço do cliente do *Web Service*, não necessitando retrabalho de reconfiguração de componentes já alterados ao gerar novamente o cliente pelo comando WSIMPORT.

A maneira utilizada para que em uma possível nova geração do cliente do *Web Service* nada se perca, foi desacoplar o cliente das alteração e configurações do *Web Service*, e para desacoplar foi criado *proxy* herdado das classes do cliente.

Um resultado alternativo que foi alcançada com o desacoplamento do cliente do *Web Service*, foi a possibilidade de extrair o cliente do projeto, possibilitando não haver código desse cliente, e em vez de ter códigos Java do cliente em nosso

projeto, é possível gerar um JAR pelo WSIMPORT e referenciar no projeto que utilizará.

Essa característica é tão interessante em questão de produtividade, que em mudança do leiaute dos artefatos do cliente do *Web Service* (se a alteração não afetar de forma direto o código do sistema que o utiliza), nem seria necessário alteração no projeto, e apenas o fazer uma nova geração do cliente com o WSIMPORT.

Produtividade por não gerar retrabalhos e dar integridade das alterações realizadas

A partir de que não há alteração do código do cliente gerado pelo WSIMPORT, não é mais necessário gerar as classes e copiar para o projeto, e possível apenas gerar o JAR do cliente utilizando o comando parâmetro `clientjar` do WSIMPORT.

6 CONSIDERAÇÕES FINAIS

Para finalizar o trabalho, é interessante fazer algumas considerações referente ao que foi estudado, desenvolvido, os resultados alcançados e possíveis continuações desse trabalho devem ser levantadas.

Com os estudos realizado nesse trabalho, as ideias amadureceram, possibilitando resultados satisfatórios referente ao uso de *proxy* em clientes de *Web Services*, e possibilitando inclusive desacoplar todo o cliente em um componente externo ao projeto.

Além de conseguir esse resultado, também alcançou seu objeto principal que era eliminar retrabalho em caso de nova geração do cliente do *Web Service* através do uso do WSIMPORT por possíveis mudanças de leiaute dos artefatos e estrutura dos serviços.

O trabalho exigiu o desenvolvimento de um projeto de teste integrando dois sistemas a partir de um terceiro sistema que é responsável apenas pela integração.

Esse projeto pode servir de base para um implantação em um sistema em produção através de um estudo de caso.

Além de esse trabalho poder continuar com um estudo de caso aplicando o conceito de unir *frameworks* com *Web Service* com o uso de *Design Patterns* em um sistema de produção.

REFERÊNCIAS

BUTEK, Russell; GALLARDO, Nicholas. **JAX-RPC versus JAX-WS**: Web services hints and tips. Disponível em: <<http://www.ibm.com/developerworks/library/ws-tip-jaxwsrpc/>>. Acesso em: 15 out. 2015

CHAPPELL, David; JEWELL, Tyler. **Java Web Services**. Sebastopol, California: O'Reilly Media, 2002.

CORREIOS (EMPRESA BRASILEIRA DE CORREIOS E TELÉGRAFOS). **Calculador de preços e prazos de encomendas**. Disponível em: <<http://www.correios.com.br/para-voce/correios-de-a-a-z/pdf/calculador-remoto-de-precos-e-prazos/manual-de-implementacao-do-calculo-remoto-de-precos-e-prazos>>. Acesso em: 12 out. 2015.

DEITEL, M. Harvey. **Java Como Programar**. Traduzido por Edson Furmankiewicz. 8ª Edição. São Paulo: Pearson Prentice Hall, 2010.

ELMASRI, Ramez; NAVATHE, Shamkant. **Sistemas de bancos de dados**. Traduzido por Daniel Vieira. 6ª Edição. São Paulo: Pearson Addison Wesley, 2011.

GAMMA, Erick et al. **Design Patterns**: Elements of Reusable Object-Oriented Software. Westford, Massachusetts: Addison Wesley, 1995.

GLASSFISH COMMUNITY. **A little bit about Handlers in JAX-WS**. Disponível em: <https://jax-ws.java.net/articles/handlers_introduction.html >. Acesso em: 25 out. 2015.

GLASSFISH COMMUNITY. **JAX-WS Reference Implementation**. Disponível em: <<https://jax-ws.java.net/>>. Acesso em: 15 out. 2015.

GLASSFISH COMMUNITY. **JAX-WS Release Documentation**. Disponível em: <<https://jax-ws.java.net/2.2.10/docs/>>. Acesso em: 15 out. 2015.

GUERRA, Eduardo. **Design Patterns com Java**: Projeto orientado a objetos guiado por padrões. São Paulo: Casa do Código, 2013.

ORACLE CORPORATION. **Oracle Buys Sun**. Disponível em: <<http://www.oracle.com/us/corporate/press/018363>>. Acesso em: 11 out. 2015.

SAUDATE, Alexandre. **SOA Aplicado**: Integrando com Web Services e além. São Paulo: Casa do Código, 2012.

SIERRA, Kathy; BATES, Bert. **Head First Java**. 2nd Edition. Sebastopol, California: O'Reilly Media, 2005.

SPRING. **Data access with JDBC**. Disponível em: <<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/jdbc.html>>. Acesso em: 17 nov. 2015.

SPRING. **GETTING STARTED - Accessing Relational Data using JDBC with Spring**. Disponível em: <<https://spring.io/guides/gs/relational-data-access/>>. Acesso em: 17 nov. 2015.

SPRING. **Spring Framework**. Disponível em: <<http://projects.spring.io/spring-framework/>>. Acesso em: 17 nov. 2015.

THE APACHE SOFTWARE FOUNDATION. **JAX-WS Guide**. Disponível em: <<https://axis.apache.org/axis2/java/core/docs/jaxws-guide.html>>. Acesso em: 15 out. 2015.

TIOBE SOFTWARE. **Tiobe Index**. Disponível em: <<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>>. Acessado em: 11 out. 2015.

WEISSMANN, Henrique. **Vire o jogo com Spring Framework**. São Paulo: Casa do Código, 2012.