OWSE-AF: Um *Framework* para Desenvolvimento de Aplicações Web

LEO SILVA, Federal University of Rio Grande do Norte (leo.moreira@me.com)
UIRÁ KULESZA, Federal University of Rio Grande do Norte (uirakulesza@gmail.com)
ULISSES TELEMACO, OWSE (ulisses.telemaco@gmail.com)

Este artigo apresenta um *framework* para desenvolvimento de aplicações web, chamado de OWSE-AF. Ele detalha a arquitetura e projeto do *framework* desenvolvido com o propósito de melhorar a produtividade de desenvolvimento de sistemas web implementados na linguagem Java. Padrões de projeto adotados no *framework* são apresentados para detalhar seu projeto. O artigo detalha, também, a experiência de uso do *framework* em projetos de desenvolvimento de *software*.

Categories and Subject Descriptors: **D.2.11 [Software Architectures]**: Patterns; **D.2.2 [Programming Languages]**: Language Constructs and Features – *Patterns*

ACM Reference Format:

Silva, L. Kulesza, U. and Telemaco, U. 2014. OWSE-AF: Um Framework para Desenvolvimento de Aplicações Web. Proceedings of the 10th Latin America Conference on Pattern Languages of Programs (SugarLoafPLoP'2014). November 2014, 15 pages.

1. INTRODUÇÃO

Ter um sistema para a web tornou-se imprescindível para uma empresa aumentar a qualidade do serviço prestado e, também, mantê-la competitiva. Com a finalidade de aumentar a produtividade no desenvolvimento de um sistema para a web, os *frameworks* web ganharam destaque nos últimos 10 anos [Mürk and Kabanov 2006]. Os autores, ainda, afirmam que, atualmente, existem centenas desenvolvidos para a linguagem Java, deixando para trás produtos comerciais ou outras plataformas como o .NET. Tais *frameworks* possuem filosofias de projeto diferentes, e, até mesmo dentro de uma abordagem ou funcionalidade, há *frameworks* que implementam de maneira bem diferente tais aspectos comparado com outros, gerando uma certa incompatibilidade entre eles [Mürk and Kabanov 2006].

A OWSE – Objects, Web and Software Engineering, uma empresa de desenvolvimento de *software* situada no Rio de Janeiro – RJ, optou por desenvolver o seu *framework* próprio, o OWSE *Application Framework*, tendo como motivações padronizar o desenvolvimento das aplicações para que cada uma delas seguisse um mesmo padrão arquitetural e de codificação, assim como reduzir a quantidade de código repetido elaborado em vários sistemas distintos para resolver problemas comuns entre eles.

Várias das aplicações desenvolvidas pela empresa OWSE são desenvolvidas com base no OWSE *framework*. A principal estratégia desse *framework* é oferecer um conjunto de funcionalidades normalmente exigidas numa plataforma web. Essas funcionalidades estão parcialmente implementadas e requerem menos esforço computacional do que implementá-las a partir do zero. Dessa forma, esse framework permite a criação de aplicações que demandam alta disponibilidade e segurança, além de baixa complexidade resultante da adoção de vários padrões de projeto no desenvolvimento do *framework*.

Os *frameworks*, em geral, podem ser classificados quanto à extensão em duas categorias: caixa preta e caixa branca [Johnson and Woolf 1998]. Os do tipo caixa preta são utilizados através de composição, ou seja, sua extensibilidade é garantida definindo-se interfaces para os componentes através de composição de objetos. Já nos do tipo caixa branca, o reuso acontece por herança. Nesta categoria, o usuário deve conhecer em detalhes o *framework*, pois terá que criar subclasses para as suas classes abstratas [Fayad et al. 1999]. O *framework* apresentado neste artigo é do tipo caixa branca, como será exposto adiante.

Este artigo apresenta o *framework* OWSE-AF e os padrões utilizados para implementá-lo. A seção 2 apresenta uma visão geral da arquitetura. Em seguida, a seção 3 apresenta a organização de camadas da arquitetura do *framework*. A seção 4 apresenta um resumo dos padrões de projeto utilizados no *framework*. A seção 5 comenta, brevemente, sistemas que têm a sua arquitetura baseada no *framework*. A seção 6 relata as experiências com a utilização do OWSE-AF. A seção 7 exibe as principais características que os *softwares* devem possuir para que o *framework* seja utilizado no seu desenvolvimento. Por fim, a seção 8 apresenta as conclusões do trabalho e os trabalhos futuros.

2. OWSE APPLICATION FRAMEWORK

O OWSE *Application Framework* (OWSE-AF) é uma infraestrutura de desenvolvimento para aplicações JEE [Oracle 2006] para a web, oferecendo, dentre outras coisas, facilidades para o desenvolvimento de aplicações CRUD (*Create*, *Retrieve*, *Update*, *Delete*). Nesta seção serão mostrados as principais funcionalidades do *framework*.

2.1 VISÃO GERAL

O OWSE-AF teve a sua arquitetura planejada de tal maneira a fornecer, aos desenvolvedores e clientes:

- *Segurança*: A segurança é realizada através de autorização e autenticação. A autenticação é baseada em usuário e senha. Já a autorização é feita através de papéis definidos de acordo com o nível de acesso de determinado usuário, para um determinado sistema.
- *Baixa complexidade*: A arquitetura do *framework* permite abstrair aspectos críticos/complexos do *software*, de forma a tornar o desenvolvimento das aplicações e seus respectivos casos de uso mais simples.
- *Baixa curva de aprendizado*: O *framework* é relativamente simples, de fácil entendimento. Quando um novo programador é contratado, em apenas um dia de treinamento ele se torna apto a desenvolver ou manter aplicações que utilizam o OWSE-AF.

Em 2006, quando o *framework* foi concebido, haviam poucas opções de *frameworks* para a plataforma Java que davam suporte ao desenvolvimento de uma aplicação web completa. Em geral, esses *frameworks* eram destinados para um aspecto muito específico da aplicação. Por exemplo, o JBoss Seam e Spring [Spring 2004] eram excelentes opções para injeção de dependência, o Hibernate era especializado em mapeamento objeto-relacional, e o *Java Server Faces* para implementar visões e controladores. No entanto, eram escassas as opções de *frameworks* que oferecessem suporte a todas as camadas de uma aplicação. Uma dessas opções era o jCompany [PowerLogic 2003], um *framework full stack* para desenvolvimento de aplicações web. No entanto, esse *framework*, na época, não era *open-source*, dificultando customizações por parte da OWSE com a finalidade de adaptar o *framework* às suas necessidades. Adicionalmente, com um *framework* próprio, havia a possibilidade de desenvolvedores estarem aptos a trabalhar em diferentes projetos, já que ele seria padrão na empresa e todos os projetos seriam desenvolvidos em uma mesma plataforma.

O *framework* OWSE-AF se propõe a fornecer um conjunto de funcionalidades e facilidades comuns para o desenvolvimento de aplicações CRUD para a web. O uso da ferramenta torna o desenvolvimento do *software* mais rápido pelo fato de minimizar o trabalho da equipe de desenvolvimento, poupando-os de escrever códigos repetitivos e "*boiler-plate*". Também, permite que a equipe se concentre em implementar as especificidades de cada aplicação, como regras de negócio e interface com o usuário.

O principal objetivo do desenvolvimento do OWSE-AF foi fornecer um conjunto de bibliotecas, infraestrutura e modelo de desenvolvimento que oferecesse qualidade e agilidade no desenvolvimento de aplicações. Grande parte das aplicações de sistemas de informação contemplam gerência e manutenção de dados em um SGBD, acrescida da execução de regras de negócio específicas. O *framework* presume que a manutenção de dados em uma aplicação trata-se de um algoritmo estático e repetitivo. Assim sendo, um dos benefícios da sua utilização é generalizar a execução de tais tarefas, deixando a cargo do desenvolvedor apenas as regras de negócio e especificidades de cada aplicação.

2.2 ARQUITETURA

O *framework* utiliza um conjunto de bibliotecas e tecnologias de mercado para o JEE. A Figura 1 apresenta um esboço da organização e distribuição dessas bibliotecas. Pode-se destacar, nesta arquitetura, a utilização do JBoss Seam [Red Hat 2009] como base da execução do *framework*. As aplicações são desenvolvidas utilizando a mesma arquitetura: são todas aplicações web que, na camada de apresentação utilizam JSF [Oracle 2006] , Richfaces [Red Hat 2009], JBoss Seam, jQuery [Foundation 2014] e Facelets [Facelets 2009]. Na camada de persistência, há o JPA [Oracle 2006] e o Hibernate [Inc. 2004] além do JBoss Seam. O *framework* em questão fornece suporte para as operações CRUD, estando, portanto, entre as camadas de apresentação e persistência das aplicações.

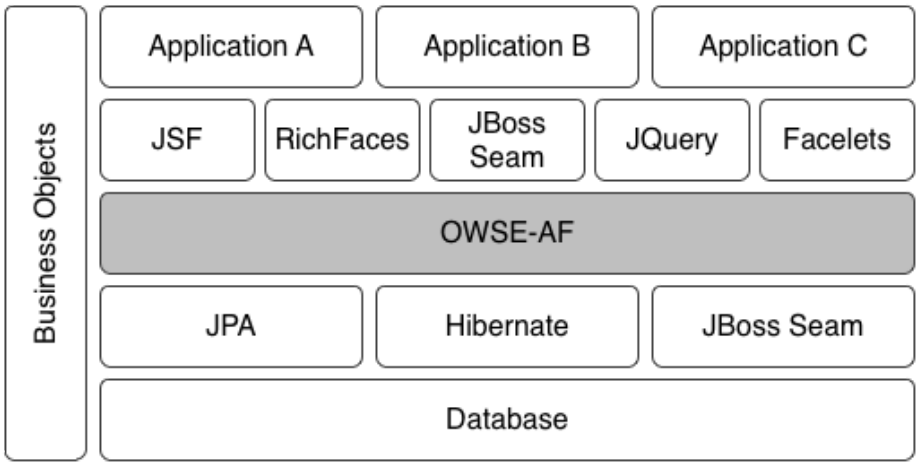

Fig. 1. Bibliotecas utilizadas no OWSE Application Framework.

2.3 MACRO ARQUITETURA MVC

Como mostrado anteriormente, a arquitetura do OWSE-AF é fortemente baseada no *framework* JBoss Seam e trabalha, basicamente, com três tipos de classes: *Entity, Bean e Service*. Essas classes desempenham funções específicas no modelo, fazendo alusão ao modelo de arquitetura *Model-View-Controller* [Buschmann 1996, Fowler 2002]. A Figura 2 ilustra o relacionamento entre essas classes.

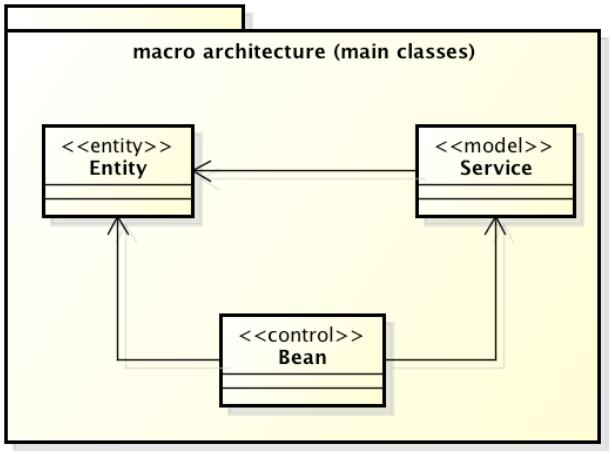

Fig. 2. Macro arquitetura do OWSE Application Framework.

No modelo MVC, as classes *Bean* e *Service* desempenham, respectivamente, os papéis de *Controller* e *Model* do padrão arquitetural *Model-View-Controller*. Já a classe *Entity* funciona como um DTO (*Data Transfer Object*) [Fowler 2002] que navega por todas as camadas do sistema. Como regra geral, para cada entidade identificada na modelagem conceitual, existirão três classes no modelo de implementação: *Entity, Service e Bean*.

3. SEPARAÇÃO EM CAMADAS

O *framework* utiliza a abordagem de separação em camadas. O intuito de tal separação é não misturar as responsabilidades dos componentes da aplicação. Dessa forma, cada componente deve ter suas responsabilidades bem definidas. Dessa forma, o primeiro padrão que o *framework* utiliza é o MVC. Esse padrão consiste em separar os componentes do sistema em três papéis com responsabilidades diferentes: o *model*, o *view* e o *controller*.

3.1 CAMADA DE PERSISTÊNCIA

As classes de entidade refletem o modelo conceitual da aplicação, bem como os modelos que atuarão também como DTO. Todas as classes desta camada devem herdar, obrigatoriamente, da classe `com.owse.fwk.entity.Entity`, para que todas as operações CRUD genéricas possam ser executadas. O quadro adiante apresenta um exemplo de entidade que herda da classe `Entity`. O padrão *Layer Supertype* [Fowler 2002] é aplicado nesta camada. Dessa forma, a classe `Entity` é a superclasse de todas as classes da camada de persistência.

Quadro 1. Trecho de código-fonte de entidade utilizando o OWSE-AF.

```
@Entity
@NamedQuery(name = "TipoCargaConteiner.findAll", query = "select tpcc from
TipoCargaConteiner tpcc")
public class TipoCargaConteiner extends
com.owse.fwk.entity.Entity<TipoCargaConteiner> {

    private static final long serialVersionUID = -157087528049939153 3L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NotEmpty
    @Length(min = 3, max = 80)
    private String nome;

    public Long getId() {
            return id;
    }
...
```

Por convenção, os nomes das classes *Entity* não devem ter nenhum prefixo ou sufixo. Será simplesmente o nome da entidade. Por exemplo: Acordo, Estacao, Cidade, UnidadeFederativa. O *framework* ainda provê o mapeamento Objeto/Relacional das classes *Entity*. Esse mecanismo é feito através das anotações do JPA. Por exemplo, para fazer o mapeamento de uma entidade é necessário definir, minimamente, as anotações `@Entity` e `@Id`. Não faz parte do escopo deste documento conceituar as tecnologias envolvidas no *framework*.

O OWSE-AF faz extenso uso de Named Queries, que são consultas predefinidas, criadas através do JPA. Essas consultas são construções que auxiliam na definição de consultas de uma entidade. Todas as consultas referentes a uma determinada entidade devem ser declaradas na classe *Entity* correspondente. Por convenção do *framework*, elas devem seguir a seguinte convenção: "NomeDaEntidade.findBy" + nome dos parâmetros separados pela palavra chave "And". O quadro adiante exibe um trecho de código com a definição de Named Queries.

Quadro 2. Trecho de código-fonte com definição de Named Queries.

```
1. @Entity
2. @NamedQueries({
3.    @NamedQuery(name = "Motorista.findAll", query = "SELECT m FROM Motorista
      m WHERE UPPER(m.statusActiveInactive) LIKE UPPER('A')"),
4.    @NamedQuery(name = "Motorista.findByFilters", query = "SELECT m FROM
      Motorista m "),
5.    @NamedQuery(name = "Motorista.findByCPF", query = "SELECT m FROM
      Motorista m WHERE m.cpf = :cpf"),
6.    @NamedQuery(name = "Motorista.findByCNH", query = "SELECT m FROM
      Motorista m WHERE m.cnh = :cnh"),
```

```
7.    @NamedQuery(name = "Motorista.checkBusinessKeyConstraint.cnh", query =
      "SELECT m FROM Motorista m WHERE m.cnh = :cnh"),
8.    @NamedQuery(name = "Motorista.checkBusinessKeyConstraint.cpf", query =
      "SELECT m FROM Motorista m WHERE m.cpf = :cpf"),
9.    @NamedQuery(name = "Motorista.checkBusinessDependency.programacoes",
      query = "SELECT p FROM Programacao p LEFT JOIN p.motoristasPermissionados
      m WHERE m.cnh = :cnh "),
10.   @NamedQuery(name = "Motorista.checkBusinessDependency.registroEntrada",
      query = "SELECT rcm FROM RegistroCaminhaoMotorista rcm WHERE
      rcm.motoristaEntrada.cnh = :cnh "),
11.   @NamedQuery(name = "Motorista.checkBusinessDependency.registroSaida",
      query = "SELECT rcm FROM RegistroCaminhaoMotorista rcm WHERE
      rcm.motoristaSaida.cnh = :cnh ")})
12.   public class Motorista extends com.owse.fwk.entity.Entity<Motorista> {
...
```

No quadro exibido, as consultas das linhas 5 e 6 seguem a convenção exposta anteriormente. Entretanto, existem ainda as consultas sem filtros, que são definidas como Named Queries chamadas de "findAll", como pode ser visto na linha 3. As Named Queries das linhas 7 e 8 obedecem a outro padrão estabelecido pelo *framework*: NomeDaEntidade + "checkBusinessKeyConstraint + nome do atributo. Esse padrão é utilizado no momento da inserção de um novo registro da entidade no banco de dados utilizado pela aplicação. No exemplo do quadro acima, ao se inserir um novo registro da entidade `Motorista`, o *framework*, automaticamente, verificará se já existe um motorista cadastrado com a CNH e com o CPF fornecidos.

Já as consultas das linhas 9, 10 e 11 são executadas no ato na remoção de um registro do banco de dados. Mais uma vez, há um padrão do OWSE-AF: NomeDaEntidade + "checkBusinessDependency" + nome do atributo do tipo *Collection*. Named Queries nesse formato, no *framework*, são executadas antes de remoções de registros, verificando se há dependência de outros registros relacionados com o registro a ser excluído.

A camada de persistência possibilita, ainda, o uso do padrão de projeto *Open Session in View* [Inc. 2009], cujo objetivo é evitar problemas de *lazy loading* [Fowler 2002] do Hibernate. O Hibernate possibilita que, no mapeamento das entidades, os relacionamentos possam ser configurados como *eager* (ansioso) ou *lazy* (preguiçoso). No tipo *eager*, quando uma entidade é carregada, todos os seus relacionamentos são, automaticamente, carregados. Já em relacionamentos do tipo *lazy*, os dados só são buscados no banco de dados se eles forem realmente utilizados. Para possibilitar o tipo de carregamento *lazy*, o Hibernate utiliza o padrão de projeto Proxy [Gamma et al. 1994] para controlar o acesso ao método da entidade que dá acesso ao relacionamento. Entretanto, para que os dados sejam buscados no banco de dados utilizando-se o tipo *lazy*, é necessário que haja uma sessão do Hibernate aberta e, normalmente, essa sessão é fechada logo após uma busca para evitar que conexões fiquem abertas com o banco de dados. O *Open Session in View* trata esse problema mantendo uma sessão do Hibernate aberta a partir do momento que ela é chamada até o momento em que toda a apresentação é exibida ao usuário para, só então, fechar a sessão.

3.2 CAMADA SERVICE

As classes da camada de serviço são usadas para implementar as regras de negócio relacionadas a uma determinada entidade. Essas classes devem herdar da classe `com.owse.fwk.web.jsf.AbstractCrudService`. Essa classe do *framework* concentra várias funcionalidades necessárias à implementação de um CRUD. Mais uma vez, o padrão de projeto *Layer Supertype* pode ser visto, onde a classe `AbstractCrudService` é a superclasse de todas as classes desta camada. O diagrama de classes a seguir mostra a relação entre as classes *Entity* e *Service*.

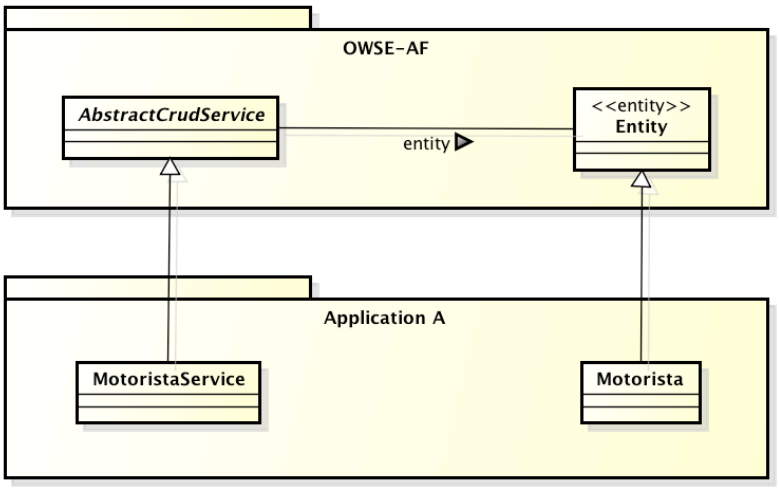

Fig. 3. Relação entre as classes da camada Entity e Service no *framework* e em uma aplicação.

Quadro 3. Trecho de código-fonte de uma classe de serviço.

```
@Name("motoristaService")
@Scope(ScopeType.CONVERSATION)
public class MotoristaService extends AbstractCrudService<Motorista> {
...
```

O Quadro 3 exibe um trecho de código da classe `MotoristaService`, que é a classe de serviço relacionada a entidade `Motorista`. Cada instância de uma classe de serviço tem a ela associada uma classe Entity. Tal entidade pode ser recuperada ou definida através dos métodos `getEntity()`/`setEntity()`. O tipo do objeto retornado em `getEntity()` depende da definição da classe *Service*. No quadro acima, a classe de serviço está herdando de `AbstractCrudService` passando como tipo a entidade Motorista. Nesse caso, o tipo da entidade de retorno do método `getEntity()` será Motorista. Ainda, o método `newEntity()` cria uma nova entidade do tipo especificado.

3.2.1 *Métodos CRUD*

A classe `AbstractCrudService` oferece os principais métodos para tratamento de persistência. Alguns deles são:

- `saveEntity()`
- `updateEntity()`
- `removeEntity()`
- `refreshEntity()`
- `findByPrimaryKey()`
- `rollback()`
- `clear()`

Esses métodos são os blocos principais de instrução de cada operação de CRUD. Via de regra, eles não precisam ser invocados pelo desenvolvedor, já que o *framework* se encarrega de orquestrar a execução do ciclo de vida das entidades.

3.2.2 *Uso do Template Method*

O OWSE-AF implementa o padrão de projeto *Template Method* [Gamma et al. 1994]. Esse padrão se encaixa na categoria de padrões de comportamento e provê uma técnica de programação conhecida como *hook*. Os principais métodos do CRUD são implementados com esse padrão, e permitem uma fácil customização. Como

regra geral, para cada um dos métodos CRUD da classe `AbstractCrudService`, existem dois métodos *hook* (*before* e *after*) que são executados pelo *framework* antes e depois do método principal. Esse *template method* permite que o método seja customizado através da sobrescrita dos *hooks*. O diagrama de sequência adiante mostra o funcionamento dos métodos *hook* na operação de salvar uma nova entidade.

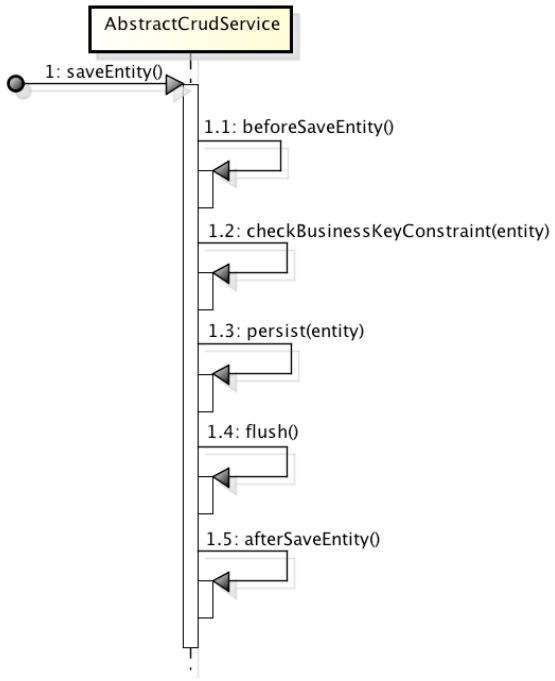

Fig. 4. Diagrama de sequência da operação de salvar uma nova entidade.

Os métodos *hook* podem ser usados para implementar uma regra específica de uma determinada entidade. O programador pode sobrescrever os métodos *before* e *after*, e, dentro destes métodos, invocar os métodos específicos de negócio. Vale salientar que os métodos *hook* existem para cada operação CRUD. A Figura 5 exibe o diagrama de sequência com a sequência de métodos executados no contexto de um sistema de Eventos. Apenas os métodos destacados em preto contém código escrito pelo desenvolvedor. Um ponto importante a se considerar é que os métodos *hook* não devem conter lógica de negócio, mas sim chamadas aos métodos que contém essas regras.

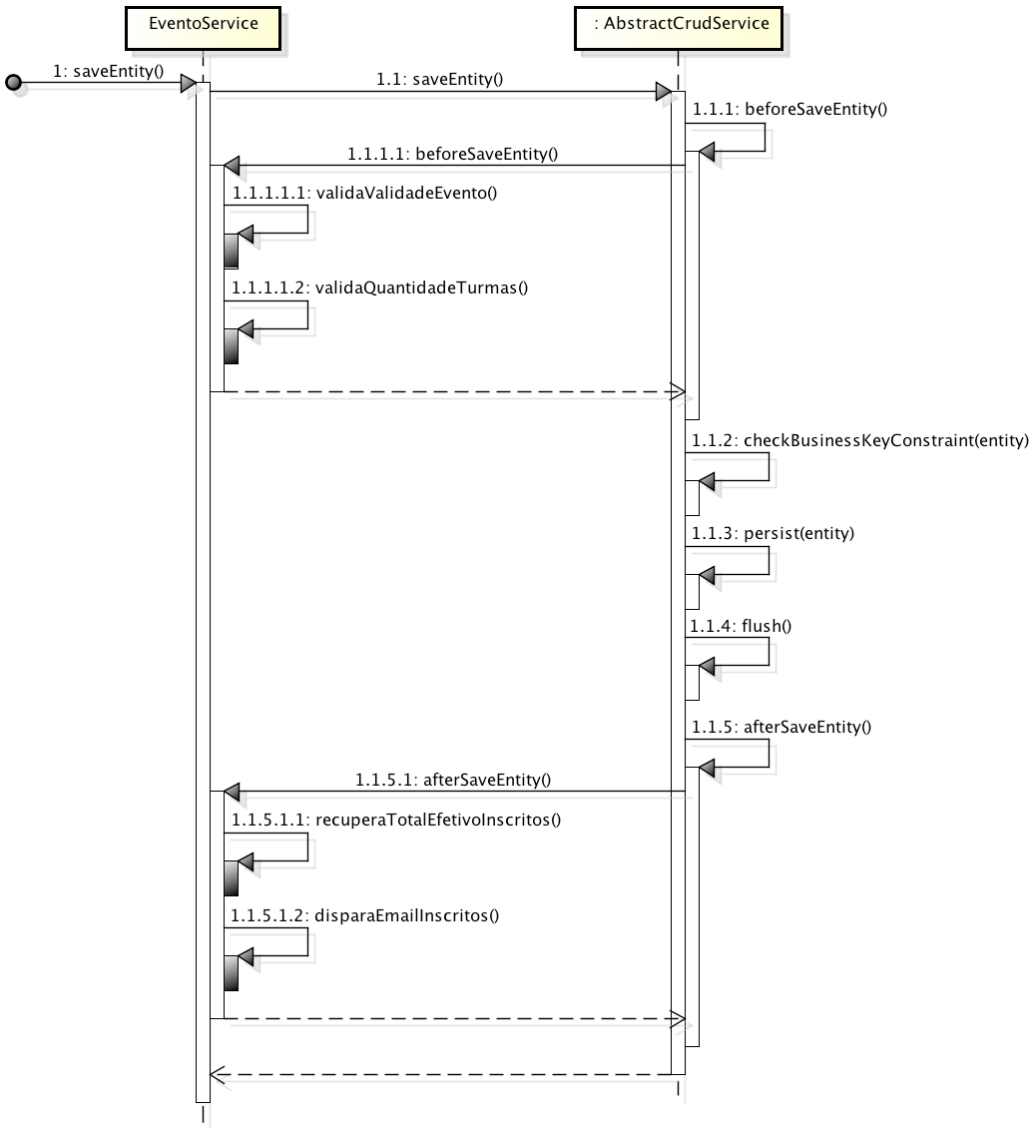

Fig. 5. Diagrama de sequência da operação de salvar uma nova entidade, executando-se validações de regras de negócio.

3.2.3 *Métodos de consulta*

A camada *Service* oferece ao desenvolvedor métodos de consulta que são suficiente em vários casos onde há a necessidade de consulta ao banco de dados da aplicação. Por exemplo, o *framework* oferece um método para buscar pela chave primária do objeto, utilizando Named Queries ou a técnica de Criteria, do Hibernate. Todos os métodos utilizam *Generic Types*, tecnologia presente na linguagem Java [Java 2011], fazendo com que os parâmetros e o retorno do método sejam contextualizados em relação à classe Service que está sendo executada em questão. O Quadro 4 exibe um exemplo de consulta por chave primária realizada pelo *framework*. É executado o método `findTerminalById()`, que recebe um objeto da classe `Terminal` como parâmetro. Esse objeto contém apenas o atributo id preenchido, que será usado pelo *framework* para localizar o respectivo registro no banco de dados. A única linha de código executada por esse método é uma chamada ao método da superclasse (`AbstractCrudService`) que, de fato, realiza a operação.

Quadro 4. Trecho de código-fonte de uma consulta por id realizada pelo *framework*.

```
public Terminal findTerminalById(Terminal terminal) throws DAOException{
      return findByPrimaryKey(terminal);
}
```

No Quadro 5, o método `findByPrimaryKey()` da classe `AbstractCrudService` é apresentado. O método realiza a busca pela chave primária do objeto recebido como parâmetro. Pode-se notar que o tipo de retorno do método é o tipo da entidade informada no momento em que a herança da classe `AbstractCrudService` é definida, como pode ser visto no Quadro 6.

Quadro 5. Trecho de código-fonte do método findByPrimaryKey, da classe AbstractCrudService.

```
public E findByPrimaryKey(Object pk) throws DAOException {
      return getEntityManager().find(getEntityClass(), pk);
}
```

Quadro 6. Trecho de código-fonte da declaração da classe, da classe TerminalService.

```
@Name("terminalService")
@Scope(ScopeType.CONVERSATION)
public class TerminalService extends AbstractCrudService<Terminal> {
...
```

3.3 CAMADA BEAN

As classes desta camada são correspondentes as da camada *Controller* do modelo MVC. São responsáveis por receber as interações do usuário junto à interface, processá-las e invocar, quando necessário, a camada *Service* para processamento de regras de negócio ou acesso ao banco de dados.

3.3.1 *Métodos CRUD*

Analogamente à camada *Service*, a camada *Bean*, representada no *framework* principalmente pela classe `AbstractCrudBean`, também dispõe de métodos CRUD generalizados. Esses métodos recebem os dados da camada de visão (*View*, do MVC) e disparam os métodos equivalentes na camada Service. O padrão de projeto *Layer Supertype*, novamente, é utilizado nesta camada. Assim sendo, a classe `AbstractCrudBean` é a superclasse de todas as classes filhas da camada *Bean*. Adiante segue um trecho de código do método `save()`, da classe `AbstractCrudBean`. No código, a classe de serviço correspondente é recuperada através do método `getEntityService()` para, então, serem chamados os métodos *hook*.

Quadro 7. Trecho de código-fonte do método save() da classe AbstractCrudBean

```
public String save() {
    try {
      if (getEntity().getVersion() == null) {
            beforeSave();
            getEntityService().saveEntity();
            getEntityService().flush();
            facesMessages.addFromResourceBundle(Severity.INFO,
      "msg.crud.save.sucess");
      afterSave();
...
```

3.3.2 *Uso do Template Method*

Os métodos CRUD da camada *Bean* também utilizam o padrão *Template Method*. Entretanto, nesta camada, a utilização dos métodos *hook* é voltada para o preparo dos dados para envio à camada *Service*, tratamento dos dados retornados da camada *Service* e preparação de dados para a camada de visão.

Como exemplo, o método `beforeSearch()` da classe `AbstractCrudBean`. No fluxo de consulta de uma entidade, todo o mecanismo de consulta e retorno dos dados é orquestrado pelo OWSE-AF, o desenvolvedor tem a obrigação, somente, de informar ao *framework* qual consulta deve ser executada. Para tal, o desenvolvedor pode sobrescrever o método `beforeSearch()` e, nele, definir a variável *namedQuery*. Essa variável é utilizada pelo *framework* e é ela quem informa qual consulta é a padrão do caso de uso em questão. O trecho de código do Quadro 8 mostra o método `beforeSearch()` sobrescrito em uma subclasse de `AbstractCrudBean`.

Quadro 8. Trecho de código-fonte do método beforeSearch() em uma subclasse de AbstractCrudBean.

```
protected void beforeSearch() throws
com.owse.fwk.exception.ApplicationException {
    super.beforeSearch();
    this.namedQuery = "Evento.findByTitulo";
}
```

Outro uso para os métodos *hook* da camada *Bean* é o preparo e complemento dos dados para serem enviados à camada *Service*. Por exemplo: dados sobre o ambiente de execução da aplicação, usuário corrente e idioma atual podem ser recuperados e utilizados para compor as entidades de negócio.

3.3.3 *Tratamento de Exceções*

Nos sistemas de informação, em geral, é uma boa prática que quando alguma regra de negócio, validação ou erros de qualquer natureza aconteçam, os usuários sejam comunicados amigavelmente. O *framework* tem essa preocupação e oferece tratamento para dois tipos de mensagens: mensagens globais, que dizem respeito a um caso de uso ou interação do usuário com o *software*; e mensagens específicas de cada campo da tela.

A classe `com.owse.fwk.exception.ApplicationException` é utilizada para o tratamento dessas exceções. Essa classe possui assinaturas que permitem que sejam criadas tanto mensagens globais quando específicas. No trecho de código do quadro adiante, é exibido um construtor onde é possível atrelar a mensagem a um determinado componente visual, através do parâmetro `componentId`.

Quadro 9. Um dos construtores da classe ApplicationException.

```
public ApplicationException(String componentId, String messageId, Object...
parameters) {
    this.componentId = componentId;
    this.mensagemId = messageId;
    this.argumentos = parameters;
}
```

Ao utilizar esse construtor, o desenvolvedor deve informar o id do componente visual cuja mensagem será exibida, o id da mensagem (o *framework* utiliza o mecanismo de internacionalização do JSF), e os parâmetros da mensagem, se houverem.

3.4 CAMADA VIEW

A camada *View*, ou camada de apresentação, é responsável pela interação com o usuário, sendo ela quem exibe as saídas da aplicação e recebe os dados fornecidos pelo usuário. O OWSE-AF utiliza os *frameworks* Facelets, JavaServer Faces e, eventualmente, o jQuery nesta camada. O Facelets, para o desenvolvedor que utiliza o JSF em suas aplicações, é o *framework* indicado para a criação de *templates* de páginas, sendo flexível e poderoso para a criação de *layouts*.

3.4.1 *Templates Facelets*

No intuito de agilizar o desenvolvimento de aplicações baseadas no *framework*, o OWSE-AF traz consigo um conjunto de *templates* Facelets para a confecção das páginas do CRUD, componentes visuais como janelas modais, janelas de diálogo e paginação de consultas. Quando uma aplicação é criada utilizando o OWSE-AF, os

templates padrões ficam disponíveis para utilização pelo desenvolvedor. A Figura 6 exibe alguns desses esses *templates*.

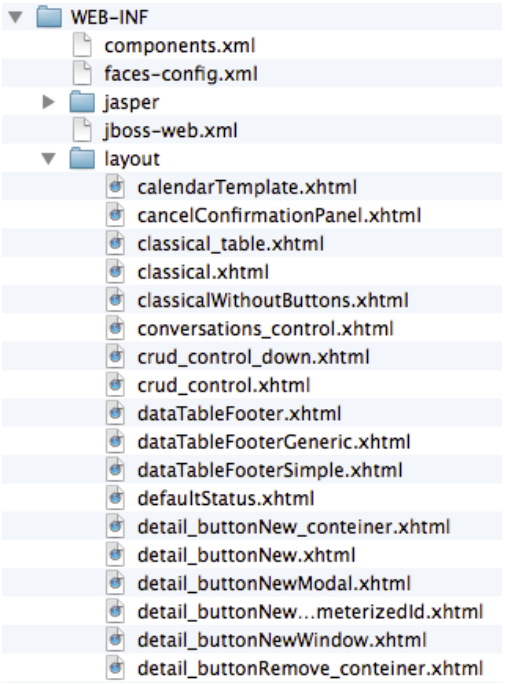

Fig. 6. Lista de *templates* disponibilizados pelo OWSE-AF.

Os *templates* listados acima se encaixam em duas categorias: a primeira, são os *templates* de página, que formam um *layout* geral e, geralmente, são atrelados ao fluxo do CRUD. O uso desses *templates* se dá através da definição da *tag template* em uma página Facelets XHTML. O quadro 10 apresenta um exemplo de uma página utilizando o *template* `templateSearchAndEdit.xhtml`.

Quadro 10. Página XHTML Facelets utilizando o template templateSearchAndEdit.xhtml.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "
http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<ui:composition xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:a4j="http://richfaces.org/a4j"
      xmlns:rich="http://richfaces.org/rich"
      xmlns:c="http://java.sun.com/jstl/core"
      xmlns:s="http://jboss.com/products/seam/taglib"
      template="/WEB-INF/layout/templateSearchAndEdit.xhtml">

      <!-- Bean gerenciado pela pagina -->
      <ui:param name="defaultBean" value="#{motoristaBean}" />
...
```

A segunda categoria dos *templates* são as que se assemelham a *snippets* de código-fonte ou com componentes visuais, que são inseridos sob demanda em uma página. Esses *templates* evitam repetição de código na camada *View*, trazendo benefícios para o desenvolvedor e para o projeto. O quadro 11 apresenta um exemplo de utilização de um *template* para rodapé de tabelas de resultados de consulta com paginação.

Quadro 11. Inclusão de *template* de rodapé de tabela.

```
<f:facet name="footer">
      <ui:include src="/WEB-INF/layout/dataTableFooterSimple.xhtml">
            <ui:param name="searchBean" value="#{acordoBean}" />
      </ui:include>
</f:facet>
```

Do modo como foi implementado, o *template* necessita de apenas um parâmetro para funcionar corretamente. No quadro acima, o parâmetro é o `searchBean`. Uma vez recebido o parâmetro, o *template* extrai informações da última consulta realizada através do framework OWSE-AF, e exibe informações como quantidade de linhas por página, total de páginas, total de registros da consulta e ainda permite a personalização do número de registros a ser exibido por página.

4. SUMÁRIO DOS PADRÕES DE PROJETO UTILIZADOS

A Tabela 1 mostra um resumo dos padrões de projeto utilizados na implementação do *framework*.

Tabela 1 Padrões de Projeto utilizados

Nome do Padrão	**Descrição**
Template Method	Define o esqueleto de um algoritmo e permite que alguns de seus passos sejam sobrescritos em classes filhas. Na camada Service e Bean existem os métodos *hook*, que permitem a customização.
Data Transfer Object	Esses tipos de objetos têm como função transportar dados entre camadas. No *framework*, os DTOs são utilizados para montar objetos provenientes de consultas específicas à base de dados.
Dependency Injection	Esse padrão consiste em um contêiner de injeção de dependências ficar responsável por instanciar objetos de classes e resolver suas dependências. No OWSE-AF, o responsável é o JBoss Seam.
Layer Supertype	Consiste em uma classe que funciona como superclasse de todas as classes de uma camada. No *framework*, o AbstractCrudService e o AbstractCrudBean são exemplos dessas classes. Elas são utilizadas como superclasse de todas as classes das camadas Service e Bean, respectivamente, fornecendo atributos e métodos comuns às subclasses.
Lazy Loading	Evita que objetos sejam buscados de forma desnecessária, fornecendo, entretanto, uma maneira de busca-los caso seja necessário. Esse padrão é utilizado pelo *framework* Hibernate.
Proxy	Tem como função controlar o acesso aos métodos de um objeto. Utilizado, por exemplo, para possibilitar o *lazy loading* de objetos carregados pelo Hibernate.
MVC	Utilizado em todo o sistema para separar os componentes em três papéis com responsabilidades distintas: o *model*, o *view* e o *controller*.
Open Session in View	Esse padrão é utilizado para garantir que a sessão do Hibernate estará aberta durante a renderização da interface gráfica, de forma que não ocorram problemas com o *lazy loading* de objetos.

5. USOS CONHECIDOS

Dentre as aplicações desenvolvidas pela OWSE, merecem destaque aplicações nas áreas de Logística de Transporte de Cargas e Gestão Portuária. Dois exemplos são o Portal de Clientes e o Sistema Integrado de Terminais. Tais aplicações têm a sua arquitetura baseada no OWSE-AF.

O Portal de Clientes é uma plataforma web para interação entre os clientes externos e os Terminais Portuários. Através dessa plataforma, os clientes dos Terminais Portuários podem executar tarefas como: cotação de preço de serviço de armazenagem, consulta de contrato, solicitação e agendamento de entrega ou coleta de contêiner, entre outros.

O Sistema Integrado de Terminais foi criado com a finalidade de gerenciar todos os aspectos logísticos da armazenagem de cargas (em contêineres ou não) em um depósito. O sistema controla, por exemplo, a entrada e saída, através de caminhão ou trem, de contêineres no depósito e gerencia motoristas que têm acesso ao terminal. Ambos os sistema são multiterminais e estão implantados em vários localidades.

6. EXPERIÊNCIA DE UTILIZAÇÃO DO FRAMEWORK

A OWSE é um empresa que trabalha, desde a sua fundação, com o desenvolvimento de aplicações para a plataforma web. Durante os quase 20 anos de existência, já desenvolveu uma grande variedade de aplicações web. Com base nessa experiência, a empresa decidiu investir na elaboração do *framework* OWSE-AF. A estratégia do OWSE-AF foi oferecer suporte para o desenvolvimento de funcionalidades normalmente requeridas em aplicações para a plataforma web. A ideia era que o esforço para desenvolver determinado aspecto da aplicação fosse menor com a utilização do OWSE-AF. Além disso, o *framework*, também, iria garantir um ganho direto com a padronização dos artefatos de *software* desenvolvidos.

O *framework* teve sua primeira versão lançada em 2006 e já passou por diversas evoluções durante esse período. O ciclo de desenvolvimento/evolução do framework segue normalmente o seguinte roteiro: (1) identifica-se uma funcionalidade que normalmente está presente em um sistema web ou que está ganhando popularidade e decide-se agregá-la ao *framework* (por exemplo integração com redes sociais); (2) implementa-se essa funcionalidade nas camadas do *framework*; e (3) garante que a inclusão dessa funcionalidade não quebra a compatibilidade e o funcionamento dos serviços existentes no *framework*.

A principal dificuldade no desenvolvimento/evolução do OWSE-AF é implementar uma funcionalidade de forma genérica e personalizável e que não prejudique a compatibilidade dos serviços existentes. O OWSE-AF já foi utilizado no desenvolvimento de cerca de 10 sistemas desenvolvidos pela OWSE. Em especial, sistemas web de pequeno e médio porte para suporte à gestão da logística de transporte de cargas e à gestão portuária.

A experiência da utilização do *framework* foi bastante positiva. A empresa relata que o tempo e o esforço de desenvolvimento foram menores com a utilização do *framework* em comparação a sistemas que foram desenvolvidos utilizando outras arquiteturas. Com relação aos artefatos gerados, a quantidade de linhas de código escrita foi menor e mais padronizada. Um ganho dessa padronização de código foi a possibilidade de intercambiar desenvolvedores entre sistemas de forma mais natural.

É importante entender que o *framework* oferece suporte apenas para funcionalidades comuns a aplicações web (autenticação e autorização, CRUD, exportação de consultas e relatórios para documentos PDF ou XLS, etc). Os aspectos específicos de cada aplicação precisam ser implementados individualmente. Por exemplo, num fluxo de exclusão de um registro que deve obedecer a uma regra específica, o *framework* trata todos os detalhes genéricos associados a essa exclusão (*layout* de tela, disposição de resultados, botões para disparar ações, mensagens de erro, criação e invocação dos comandos SQL, etc), mas cabe ao desenvolvedor implementar a regra específica que deverá ser executada antes de realizar a exclusão.

7. CARACTERÍSTICA DE USABILIDADE DO OWSE-AF

Como já dito anteriormente, o OWSE-AF é um *framework* projetado para suportar o desenvolvimento de aplicações *web*. A estratégia é reutilizar componentes com o objetivo de melhorar a qualidade do *software* e diminuir o esforço de desenvolvimento. Esta seção apresenta as características de usabilidade dessa ferramenta. Ou seja, são listadas as características que indicam a viabilidade de se utilizar, ou não, o *framework* para o desenvolvimento de determinado *software*.

A primeira etapa da utilização do OWSE-AF é fazer uma modelagem do domínio do projeto e, depois, verificar se ele atende aos seguintes requisitos:

- *Aplicações Java*: o *framework* foi escrito na linguagem Java (versão 6) e oferece suporte exclusivo para o desenvolvimento de aplicações nessa linguagem que utilizem a versão 6 ou superior. Não existe nenhum suporte para o desenvolvimento de aplicações em outra linguagem de programação;
- *Aplicações para a plataforma web*: o OWSE-AF foi projetado para dar suporte ao desenvolvimento de aplicações *web*. Apesar de ser possível, não existe nenhuma vantagem em utilizar o *framework* para o desenvolvimento de sistemas em outra arquitetura (aplicações *desktop*, por exemplo). Dessa forma, é recomendado utilizá-lo apenas para o desenvolvimento de aplicações *web*;
- *Tecnologias Core*: outro ponto importante está relacionado às tecnologias que o *framework* incorpora. Ao utilizá-lo, agrega-se um arcabouço de tecnologias que não podem ser alteradas ou cuja alteração traria um alto custo, de modo que inviabilizaria a sua utilização. A camada de *view*, por exemplo, deve ser implementada utilizando a tecnologia JSF 1.2. Até o presente momento, o *framework* não dispõe de opções para alterar várias dessas tecnologias por outra semelhante. Se houver alguma restrição (de negócio, arquitetural ou outra natureza) quanto à utilização de alguma dessas tecnologias, o *framework* não poderá ser utilizado. A seguir, seguem as tecnologias que fazem parte do *core* do *framework* e cuja alteração não é suportada:

    - Camada *View*: JSF 1.2 e Facelets 1.1.x
    - Camada *Bean* e *Service*: JBoss Seam 2.2.x
    - Camada de Persistência: Hibernate e JPA
    - Biblioteca para geração de arquivos PDF: JasperReport
    - Biblioteca para geração de arquivos XLS: POI
- *Transaction-per-request*: o OWSE-AF foi projetado para trabalhar com sistemas *web* transacionais onde cada requisição do usuário é tratada com uma transação. Não existe a possibilidade de alterar esse comportamento e criar uma transação para atender a várias requisições *web*. Portanto, se esse cenário é exigido, o *framework* não poderá ser utilizado;
- *Contextos*: o *framework* trabalha com os seguintes contextos: Business Process, Application, Session, Conversation, Request e Page. Com exceção dos contextos Business Process e Conversation, que são contextos específicos do *framework* JBoss Seam, os demais fazem parte da especificação Servlet. Esses são os únicos contextos disponíveis ao utilizar o *framework*. Caso haja necessidade de utilizar outro contexto, a recomendação é não utilizar o OWSE-AF;
- *Operações CRUD*: o *framework* oferece suporte ao desenvolvimento de funcionalidades do tipo CRUD. A quantidade de código gerada para uma funcionalidade como essa, com poucas customizações, é bastante reduzida. Portanto, sugere-se a utilização do OWSE-AF para o desenvovimento de sistemas de informação que tenham uma quantidade significativa de objetos que precisam ser manipulados através de funcionalidades do tipo CRUD. Para sistemas que atendem um segmento específico de mercado, onde a ocorrência de CRUDs seja pequena, o *framework* pode não trazer tantos benefícios, e, portanto, a sua utilização é desencorajada;
- *Operações para Consulta e Exportação PDF e XLS*: o *framework* oferece funcionalidades relacionadas a consultas e exportação de resultados. Atualmente, o OWSE-AF oferece suporte para a exportação de documentos nos formatos PDF e XLS. Caso haja a necessidade de exportar para outro tipo de documento, o *framework* poderá user utilizado, mas deverá passar por customizações para suportar o novo requisito.

8. CONCLUSÃO E TRABALHOS FUTUROS

Baseado na vasta experiência em desenvolvimento de sistemas para a web e na experiência de outros desenvolvedores na criação de outros *frameworks* web para a linguagem Java, a OWSE desenvolveu o seu próprio *framework*, o OWSE-AF, fazendo com que as aplicações fossem desenvolvidas seguindo o mesmo padrão arquitetural e de codificação, dentre outras vantagens expostas ao longo deste artigo. Tal *framework* tem se mostrado adequado para o desenvolvimento das aplicações dos clientes da empresa. Vários problemas corriqueiros enfrentados por desenvolvedores foram sanados ou diminuídos, por exemplo, poupando-os de escrever códigos repetitivos, aumentando a produtividade e qualidade dos produtos desenvolvidos.

Os padrões de projeto tiveram papel fundamental na implementação do OWSE-AF, pois vários deles foram utilizados para implementar as soluções fornecidas pelo *framework*, como, por exemplo, o *Template Method*, que foi utilizado para oferecer ao desenvolvedor um modo fácil de customização das operações CRUD. A utilização dos padrões de projeto foi importante para reduzir a curva de aprendizado do *framework*, quando novos desenvolvedores começam a trabalhar na arquitetura.

A equipe de arquitetos da empresa tem se preocupado sempre em melhorar e modernizar a arquitetura, trazendo novas tecnologias e acrescentando novas funcionalidades, objetivando aumentar a produtividade e facilitar o desenvolvimento. Entretanto, esse acréscimo de novas funcionalidades e tecnologias têm sido feito paulatinamente, com pesquisa e planejamento, para que os desenvolvedores e aplicações atuais não sejam prejudicados com novos *releases*.

Como trabalhos futuros, existe um planejamento que o *framework* implemente diversas melhorias e novas funcionalidades. Dentre elas, pode-se destacar:

- *Auditoria*: fornecer aos desenvolvedores a possibilidade de auditar as entidades (classes filhas da classe Entity) da aplicação. Essa funcionalidade, a princípio, será implementada utilizando anotações.
- *Serviços RESTful*: os serviços RESTful [Richardson and Ruby 2007] fornecerão ao OWSE-AF a possibilidade de integrar-se mais facilmente com outros sistemas, em outras plataformas (*tablets* e *smartphones*), utilizando, para isso, o estilo arquitetural REST [Fielding 2000].

- *Suporte a Bigdata*: uma das funcionalidades planejadas é o suporte ao Bigdata. Essa tecnologia se refere a uma quantidade enorme de dados desestruturados produzidos por aplicações de alta performance [Cuzzocrea et al. 2011].
- *Configuração via anotação*: fornecer aos programadores a possibilidade de configurar diversos aspectos do *frameworks* através de anotações.

REFERENCES

2011. Java. http://www.java.com/

Buschmann, F., et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley Sons, 1996.

Cuzzocrea, A., Song, I.-Y., and Davis, K.C. *Analytics over Large-Scale Multidimensional Data: The Big Data Revolution!* Proc. of ACM DOLAP, 2011.

Facelets. 2009. *JavaServer Facelets*. Disponível em: https://facelets.java.net/. Acesso em: 10 de agosto de 2014.

Fayad, M. E., Schmidt, D. C., and Johnson, R. 1999. *Building Application Frameworks: Object-Oriented Foundation of Frameworks Design*. John Wiley & Sons, Nova Iorque, pp. 3-27, 1999.

Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation. University of California, Irvine, 2000.

Foudation, jQuery. 2014. *jQuery*. Disponível em: http://jquery.com/. Acesso em: 10 de agosto de 2014.

Fowler, M. 2002. *Patterns of Enterprise Application Architecture* 1st Ed. Addison-Wesley.

Gamma, E., Johnson, R., Helm, R., and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* 1st Ed. Addison-Wesley.

Inc., J. 2004. *Hibernate*. Disponível em: http://www.hibernate.org/. Acesso em: 11 de agosto de 2014.

Johnson, Ralph E. and Woolf, B. 1998. *Type Object*. Em: "Martin, R. C.; Riehle, D.; Buschmann, F. *Pattern Languages of Program Design 3*. Reading-MA, Addison-Wesley, 1998", p. 47-65.

PowerLogic, 2003. *jCompany Developer Suite*. Disponível em: http://www.powerlogic.com.br/powerlogic/ecp/comunidade.do?evento=portlet&pIdPlc=ecpTaxonomiaMenuPortal&app=portal&tax=1020&lang=pt_BR&pg=540&taxn=1003&view=interna&taxp=0&/. Acesso em: 11 de agosto de 2014.

Oracle. 2006. *JavaServer Faces*. Disponível em: http://javaserverfaces.java.net/. Acesso em: 10 de agosto de 2014.

Oracle. 2006. *Java EE 5 Technologies*. Disponível em: http://www.oracle.com/technetwork/java/javaee/tech/javaee5-jsp-135162.html. Acesso em: 10 de agosto de 2014. http://www.hibernate.org/. Acesso em: 10 de agosto de 2014.

Oracle. 2006. *Java Persistence API*. Disponível em: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html/. Acesso em: 10 de agosto de 2014.

Mürk, O. and Kabanov, J. 2006. Aranea – Web Framework Construction and Integration Kit. In *PPPJ 2006.* Mannheim, Germany.

Red Hat. 2009. *Seam Framework*. Disponível em: http://www.seamframework.org/. Acesso em: 10 de agosto de 2014.

Red Hat. 2009. *Richfaces*. Disponível em: http://richfaces.jboss.org/. Acesso em: 10 de agosto de 2014.

Richardson, L. and Ruby, S. 2007. *RESTful web servisse* 1st Ed. O`Reilly Media.

Spring, 2004. *Spring Framework*. Disponível em: http://projects.spring.io/spring-framework/. Acesso em: 11 de agosto de 2014.