FORMBUILDER: FRAMEWORK PARA GERAÇÃO AUTOMÁTICA DE TELAS PRIMEFACES COM BASE EM ANOTAÇÕES

Silvano Lohn[1], Adilson Vahldick[1, 2]

[1]Universidade Regional de Blumenau (FURB) – Blumenau/SC - Brasil
[2]Departamento de Sistemas de Informação – UDESC Ibirama/SC – Brasil

silvano.lohn@gmail.com, adilson.vahldick@udesc.br

Resumo
Este artigo apresenta o *framework* FormBuilder, desenvolvido para automatizar a geração de telas PrimeFaces e agilizar o desenvolvimento de aplicações Web. O *framework* gera três tipos de componentes: botão, caixa de edição e caixa de marcação, que serão gerados com base em anotações declaradas nas classes de controle desenvolvidas pelo usuário, a ligação entre a classe de controle e a página web será realizada através de uma *tag* JSF customizada. No final do artigo é apresentada uma aplicação exemplo que utiliza o *framework* desenvolvido.

Palavras-chave: *Framework*. Gerador de Tela. Java. JSF. PrimeFaces.

Abstract
This article presents the framework FormBuilder, developed to automate the generation of screens PrimeFaces and expedite the development of Web applications. The framework generates three types of components: button, edit box and check box, that will be generated based on annotations declared in control classes developed by the user, the connection between the control class and the web page will be accomplished through a custom tag JSF. In the end of the article is displayed a example application that uses the framework developed.

Keywords: *Framework. Screen Generator. Java. JSF. PrimeFaces.*

1. Introdução

Em constante crescimento, a Internet está cada vez mais presente na vida das pessoas ao redor do mundo, diversificando seu uso e afetando diversos aspectos da vida cotidiana dos usuários domésticos, indústria, comércio e educação. Hoje diversos setores que realizam seus negócios e operações na Internet estão migrando e criando novos sistemas para o ambiente web, suas funcionalidades e conteúdo passam por mudanças constantes exigindo um grande esforço das equipes de desenvolvimento.

Segundo Gimenes et al. (2005), um dos principais objetivos da Engenharia de Software é o reuso de código. Através da reutilização de software obtém-se o aumento da qualidade da aplicação e a redução do esforço de desenvolvimento, para isso, utilizam-se componentes de software, que são unidades reutilizáveis que oferecem serviços através de interfaces bem definidas.

Horstmann (2007) define um *framework* sendo um conjunto de classes cooperativas que implementam os mecanismos que são essenciais para um domínio de problemas específicos, provendo funcionalidades comuns e adicionais para o desenvolvimento dos módulos da aplicação.

Os principais benefícios da utilização de *frameworks*, segundo Sommerville (2007), advêm da modularidade, reusabilidade e extensibilidade que os *frameworks* proporcionam. *Frameworks* encapsulam detalhes de implementação voláteis através de seus pontos de extensão, interfaces estáveis e bem definidas, aumentando a modularidade da aplicação. Os locais de mudanças de

projeto e implementação da aplicação construída usando o *framework* são localizados, diminuindo o esforço para entender e manter a aplicação.

Pensando no reuso de código e o aumento da produtividade do desenvolvimento é proposto neste trabalho o desenvolvimento de um *framework* chamado FormBuilder. O *framework* proposto é dividido em três partes principais: uma *tag* JSF customizada, que realiza a ligação da página web do usuário com a classe de controle, as anotações que serão declaradas pelo usuário em suas classes de controle e um motor que fará a leitura das anotações definidas, gerando os componentes na página web. O *framework* realiza a geração de três tipos de componentes de interface: botão, caixa de edição e caixa de marcação, estes foram os escolhidos por estarem presentes na maioria dos formulários. Abaixo será demonstrado detalhadamente cada componente do *framework* e uma aplicação exemplo demonstrando a utilização da *tag* customizada, as anotações e a página gerada.

A seguir será apresentada a arquitetura do *framework* FormBuilder, os componentes desenvolvidos e uma aplicação exemplo, que demonstra a utilização do *framework* desenvolvido.

2. JSF e PrimeFaces

Segundo Geary (2010), JavaServer Faces (JSF) é uma especificação e uma implementação de referência de um *framework* para o desenvolvimento de aplicações web. O JSF foi criado para facilitar de maneira significativa, a escrita e manutenção de aplicações que rodam em um Java Application Server e apresentam suas interfaces de usuário de volta para um cliente especifico.

Caliskan (2013) menciona que o PrimeFaces é um *framework* de componentes utilizado para auxiliar no desenvolvimento e elaboração de interfaces de sistemas web que utilizam tecnologia JSF. O *framework* PrimeFaces oferece cerca de 100 componentes de interface, sendo todos personalizados e de código fonte aberto. A utilização deste *framework* permite uma infinita gama de possibilidades na utilização de *layouts* e mais de 30 temas para personalização da interface, estes de fácil inclusão e alteração durante o desenvolvimento e o uso da aplicação.

3. Arquitetura do Framework

No desenvolvimento deste trabalho foram utilizados os frameworks, citados no item 2, a união na utilização destes *frameworks* trouxe uma maior velocidade no desenvolvimento do *framework* FormBuilder. Abaixo é apresentado os componentes desenvolvidos.

3.1. Anotações

Chisty (2009) informa que anotações são *meta-tags* que podem ser adicionadas ao código-fonte Java, não alterando de nenhuma forma a execução do programa, mas possibilitando que em tempo de execução suas propriedades sejam acessadas pela aplicação. Anotações podem ser utilizadas em diversas situações: registro de logs, testes unitários, documentação, geração de código e suporte para os componentes de uma classe.

No *framework* foram desenvolvidas as anotações *Component*, *Model* e *Button*, que serão declaradas pelo usuário em suas classes de controle, definindo assim os componentes que serão gerados na tela. Abaixo é apresentada cada uma das anotações.

3.1.1. Anotação Button

A anotação *Button* deve ser definida nos métodos das classes de controle do usuário. Na tela gerada será apresentado ao usuário um botão. Se o mesmo for pressionado o método que foi o alvo da anotação será executado. Essa anotação exige que o usuário defina no parâmetro *caption*

o texto que será apresentado no botão gerado. A Figura 1 apresenta a utilização da anotação *Button*.

```
@Button(caption = "Texto do Botão")
public void metodoDaClasseDeControle() {

}
```

Figura 1 – Utilização da anotação Button

3.1.2. Anotação Component

A anotação *Component* deve ser definida nos atributos da classe de controle. Dois parâmetros devem ser configurados pelo usuário na utilização desta anotação: o parâmetro *label* irá definir o texto que será apresentado junto ao componente gerado, e o parâmetro *type* define qual o tipo do componente será gerado: *INPUT* (caixa de edição) ou *CHECKBOX* (caixa de marcação). A Figura 2 apresenta a utilização da anotação *Component*.

```
@Component(label = "Label do Campo: ", type = ComponentType.INPUT)
private String campoDaClasseDeControle1;

@Component(label = "Label do Campo", type = ComponentType.CHECKBOX)
private String campoDaClasseDeControle2;
```

Figura 2 – Utilização da anotação Component

3.1.3. Anotação Model

A anotação *Model* deve ser utilizada quando o usuário deseja gerar componentes para uma classe de modelo, para isso a classe de modelo deve estar declarada na classe de controle do usuário. Essa anotação possui um único parâmetro chamado *value*, esse parâmetro é um *array* de anotações do tipo *Component*, dessa forma o usuário poderá declarar na anotação *Model* vários atributos da classe de modelo. Quando a anotação *Component* é utilizada em conjunto com a anotação *Model* o parâmetro *attribute* deve ser declarado na anotação *Component*, neste parâmetro deve ser informado o nome do atributo da classe de modelo que será utilizado para a geração do componente. A Figura 3 apresenta a utilização da anotação *Model*.

```
@Model({
    @Component(label = "Label do Campo: ", type = ComponentType.INPUT, attribute = "atributoDaClasseDeModelo"),
    @Component(label = "Label do Campo: ", type = ComponentType.CHECKBOX, attribute = "atributoDaClasseDeModelo")
})
private ClasseDeModelo classeDeModelo;
```

Figura 3 – Utilização da anotação Model

3.2. Tag formBuilder

Foi criada uma *tag* customizada chamada *formBuilder* que será utilizada para apresentar os componentes gerados na página web do usuário. A *tag* é desenvolvida utilizando *Facelets* que, segundo Oracle (2012), é uma linguagem leve e poderosa de declaração utilizada para construir *templates* de páginas e componentes customizados, e dessa forma sendo possível reduzir o tempo e o esforço dispensado para realização do desenvolvimento de interfaces web. A linguagem *Facelets* está inclusa no *framework* JSF a partir da versão 2.0.

A declaração de uma *tag* customizada *Facelets* é realizada através de um arquivo XML. A Figura 4 apresenta a declaração da *tag formBuilder*.

```
1  <facelet-taglib xmlns="http://java.sun.com/xml/ns/javaee"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java
4      version="2.0">
5
6      <namespace>http://www.silvano.com.br/formbuilder</namespace>
7      <tag>
8          <tag-name>formBuilder</tag-name>
9          <handler-class>br.com.silvano.FormBuilder</handler-class>
10         <attribute>
11             <name>managedBean</name>
12             <required>true</required>
13         </attribute>
14     </tag>
15 </facelet-taglib>
```

Figura 4 – Declaração da tag formBuilder

Na linha 6 da Figura 4 é declarado o *namespace* da *tag* customizada. É através do *namespace* que uma *tag* é diferenciada de outras *tags*, e dessa forma não acontece conflitos entre bibliotecas de componentes diferentes que podem ser utilizadas no desenvolvimento de um mesmo projeto JSF. Na linha 8 é definido o nome da *tag*, e neste caso *formBuilder*. Na linha 9 é definida a classe que fará a manipulação deste componente. Na linha 11, é declarado o atributo *managedBean* que será utilizado pelo usuário para informar qual a classe de controle foi utilizada para a declaração das anotações.

3.3. Motor Gerador de Telas

A classe *FormBuilder* que realiza a manipulação da *tag* desenvolvida também foi utilizada para servir como motor gerador de telas: é ela que fará a leitura da classe de controle do usuário, procurando as anotações declaradas e gerando os respectivos componentes de interface.

Durante a execução da aplicação, para a *tag formBuilder* declarada em uma página web, será criada uma instância da classe *FormBuilder*. A Figura 5 apresenta o construtor da classe *FormBuilder*.

```
34 public FormBuilder(TagConfig config) {
35     super(config);
36     TagAttribute tag = getAttribute("managedBean");
37     if (tag != null) {
38         managedBeanName = tag.getValue();
39         panel = createPanel();
40         panel.setId(managedBeanName + "FormBuilder" + tagId);
41     }
42 }
```

Figura 5 – Construtor da classe FormBuilder

Na linha 36 é realizada a leitura da propriedade *managedBean* que foi declarada na *tag*. Na linha 39 é criado um painel onde serão inseridos todos os componentes de interface gerados pelo motor. Toda vez que a página web onde a *tag* está declarada sofre alguma atualização, o método *apply* da classe *FormBuilder*, apresentado na Figura 6, é executado.

```
44  public void apply(FaceletContext context, UIComponent parent) throws IOException {
45      this.parent = parent;
46      UIComponent oldPanel = parent.findComponent(panel.getId());
47      if (oldPanel != null) {
48          parent.getChildren().remove(oldPanel);
49      }
50      Object managedBean = getManagedBean();
51      if (managedBean != null) {
52          PanelGrid panelGrid = createPanelGrid(2);
53          parserFields(panelGrid, managedBean.getClass());
54          Collection<UIComponent> methodsCollection = new ArrayList<UIComponent>();
55          parserMethods(methodsCollection, managedBean);
56          panel.getChildren().add(panelGrid);
57          panel.getChildren().addAll(methodsCollection);
58          parent.getChildren().add(panel);
59      }
60  }
```

Figura 6 – Método apply da classe FormBuilder

O método *getManagedBean*, executado na linha 50, retorna um objeto referente à classe de controle que foi declarada na *tag formBuilder*. A função *parserFields*, executada na linha 53, realiza o *parser* dos campos da classe de controle, procurando as anotações *Component* e *Model*. Na linha 55 é executado o método *parserMethods* que realiza o *parser* dos métodos da classe de controle procurando por anotações *Button*. Todos os componentes gerados serão inseridos em um painel, inicialmente acontece a geração dos componentes de caixa de edição e caixa de marcação, o componente será gerado seguindo a ordem em que as anotações foram definidas na classe de controle, em seguida acontece a geração dos botões, que serão inseridos no final do painel. A Figura 7 apresenta o método *parserFields*.

```
62  private void parserFields(PanelGrid panelGrid, Class<?> clazz) {
63      Field[] fields = clazz.getDeclaredFields();
64      for (Field field : fields) {
65          Annotation[] annotations = field.getAnnotations();
66          for (Annotation annotation : annotations) {
67              if (annotation instanceof Component) {
68                  parserComponent(panelGrid, (Component) annotation, field.getName());
69              } else if (annotation instanceof Model) {
70                  for (Component component : ((Model) annotation).value()) {
71                      parserComponent(panelGrid, component, field.getName() + "." + component.attribute());
72                  }
73              }
74          }
75      }
76  }
```

Figura 7 – Método parserFields

O método *parserFields* busca todos os campos declarados na classe de controle. Em seguida, cada campo da classe é verificado se o mesmo foi alvo das anotações *Component* ou *Model*. Para cada anotação *Componet* encontrada o método *parserComponent* será executado. A Figura 8 apresenta o método *parserComponent*.

```
78  private void parserComponent(PanelGrid panelGrid, Component component, String fieldName) {
79      panelGrid.getChildren().add(createOutputLabel(component.label(), fieldName));
80      switch (component.type()) {
81      case INPUT:
82          panelGrid.getChildren().add(createInputText(fieldName));
83          break;
84      case CHECKBOX:
85          panelGrid.getChildren().add(createCheckBox(fieldName));
86          break;
87      }
88  }
```

Figura 8 – Método parserComponent

Na linha 79 do método *parserComponent* é criado um *label* referente ao componente que será gerado, na linha 80 é verificado qual o tipo de componente o motor deverá gerar. A Figura 9 apresenta o método *parserMethods*, esse método realiza a busca das anotações *Button* nos métodos da classe de controle. Um botão será gerado se o método foi alvo da anotação *Button*.

```
90  private void parserMethods(Collection<UIComponent> methodsCollection, Object object) {
91      Method[] methods = object.getClass().getMethods();
92      for (Method method : methods) {
93          Annotation[] annotations = method.getAnnotations();
94          for (Annotation annotation : annotations) {
95              if (annotation instanceof Button) {
96                  Button button = (Button) annotation;
97                  methodsCollection.add(createCommandButton(button.caption(), method.getName()));
98              }
99          }
100     }
101 }
```

Figura 9 – Método parserMethods

A Figura 10 apresenta o método que realiza a geração da caixa de edição. Na linha 130 é criada a expressão que faz a ligação do componente com o atributo que foi o alvo da anotação na classe de controle. Dessa forma, toda vez que a página sofrer uma atualização, o valor do atributo será apresentado no componente gerado. Todo componente gerado deve possuir um identificador único. Na linha 132 é atribuído o identificador ao componente, que é gerado realizando a concatenação do nome informado no *managedBean* e o nome do atributo da classe referente ao componente gerado.

```
128 private InputText createInputText(String fieldName) {
129     InputText inputText = (InputText) createComponent(InputText.COMPONENT_TYPE);
130     ValueExpression ve = createValueExpression("#{" + managedBeanName + "." + fieldName + "}");
131     inputText.setValueExpression("value", ve);
132     inputText.setId(getComponentId(fieldName));
133     return inputText;
134 }
```

Figura 10 – Geração da caixa de edição

A Figura 11 apresenta o método que realiza a geração da caixa de marcação. Da mesma forma que o campo de edição, é criada a expressão que realiza a ligação do componente com o atributo da classe que foi o alvo da anotação e a geração do seu identificador.

```
103 private SelectBooleanCheckbox createCheckBox(String fieldName) {
104     SelectBooleanCheckbox checkBox = (SelectBooleanCheckbox) createComponent(SelectBooleanCheckbox.COMPONENT_TYPE);
105     ValueExpression ve = createValueExpression("#{" + managedBeanName + "." + fieldName + "}");
106     checkBox.setValueExpression("value", ve);
107     checkBox.setId(getComponentId(fieldName));
108     return checkBox;
109 }
```

Figura 11 – Geração da caixa de marcação

A Figura 12 apresenta o método que realiza a geração do botão. Na linha 139 é atribuído ao botão o nome que foi definido na propriedade *caption* da anotação *Button*. Na linha 141 é atribuída uma ação ao botão que realiza a execução do método que foi alvo da anotação toda vez que o botão gerado for pressionado.

```
136 private CommandButton createCommandButton(String value, String action) {
137     CommandButton commandButton = (CommandButton) createComponent(CommandButton.COMPONENT_TYPE);
138     commandButton.setAjax(false);
139     commandButton.setValue(value);
140     MethodExpression me = createMethodExpression("#{" + managedBeanName + "." + action + "}");
141     commandButton.setActionExpression(me);
142     commandButton.setId(managedBeanName + action);
143     commandButton.setUpdate(parent.getId());
144     return commandButton;
145 }
```

Figura 12 – Geração do botão

A Figura 12 apresenta o método que realiza a geração do botão. Na linha 139 é atribuído ao botão o nome que foi definido na propriedade *caption* da anotação *Button*. Na linha 141 é atribuída uma ação ao botão que realiza a execução do método que foi alvo da anotação toda vez que o botão gerado for pressionado.

4. Trabalhos Correlatos

Um dos principais *frameworks* para geração de interface com base nos objetos de controle ou modelos da aplicação é o *framework* Apache Isis. Segundo Apache (2013), ele é um *framework* Java utilizado para o desenvolvimento rápido de aplicações orientadas a domínio, os desenvolvedores definem objetos de domínio seguindo convenções e anotações, o *framework* interpreta estes dados e faz a geração da apresentação e a persistência dos dados da aplicação.

O Quadro 1 apresenta a comparação entre as principais funcionalidades dos *frameworks* FormBuilder e Apache Isis.

Principais Funcionalidades	*FormBuilder*	*Apache Isis*
Configuração de validações e componentes via anotações		X
Geração de componentes de interface via anotações	X	X
Framework desenvolvido em Java e de código aberto	X	X
Possui camada de persistência e segurança		X
Possui integração com jUnit		X
Reutilização de páginas para geração automatizada de componentes	X	

Quadro 1 – Comparação de funcionalidades

5. Aplicação Exemplo

Para demonstrar a utilização do *framework*, foi desenvolvida uma aplicação exemplo dividida em três partes: a página XHTML, onde estará declarada a *tag formBuilder*; a classe de controle, onde será declarada as anotações; e a classe de modelo. Nas próximas seções serão apresentados cada um desses artefatos.

5.1. Página XHTML

A Figura 13 apresenta a página XHTML que fará a exibição dos componentes gerados. Na linha 9 é declarado o *namespace* referente a *tag* desenvolvida. Na linha 16 está declarada a *tag formBuilder.*

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4      "http://www.w3.org/TR/xhtml1/DTD/xhtml1- transitional.dtd">
5  <html xmlns="http://www.w3.org/1999/xhtml"
6      xmlns:h="http://java.sun.com/jsf/html"
7      xmlns:p="http://primefaces.org/ui"
8      xmlns:f="http://java.sun.com/jsf/core"
9      xmlns:d="http://www.silvano.com.br/formbuilder">
10
11 <h:head>
12     <title>Cadastro de Carros</title>
13 </h:head>
14 <h:body>
15     <h:form>
16         <d:formBuilder managedBean="carroController" />
17     </h:form>
18 </h:body>
19 </html>
```

Figura 13 – Declaração da página XHTML

A ligação entre a *tag formbuilder* e a classe de controle é realizada através do parâmetro *managedBean* da *tag*. Na classe de controle deve-se declarar a anotação *ManagedBean*, o valor informado no parâmetro *name* da anotação deverá ser o mesmo utilizado no parâmetro *managedBean* da *tag formBuilder*. A Figura 14 apresenta a declaração da anotação *ManagedBean* da classe de controle.

```
11 @SessionScoped
12 @ManagedBean(name = "carroController")
13 public class CarroController {
14
```

Figura 14 – Declaração da anotação ManagedBean na classe de controle

5.2. Classe de Controle

A figura 15 apresenta a classe de controle *CarroController*. Nesta classe foi realizada a declaração da anotação *ManagedBean* conforme citado no item 5.1. As anotações declaradas na classe de controle serão utilizadas para geração dos componentes. Na linha 15 é declarada a anotação *Model*, onde a mesma possui a declaração das anotações *Component* que foram utilizadas para referenciar os campos da classe de modelo *CarroModelo*. Na linha 23 é realizada a declaração da anotação *Component* referenciando o campo quantidade que faz parte da própria classe de controle. Como citado no item 3.1.2 este tipo de declaração não utilizada o parâmetro *attribute* da anotação *Component*.

```
@SessionScoped
@ManagedBean(name = "carroController")
public class CarroController {


    @Model({
        @Component(label = "Marca: ", type = ComponentType.INPUT, attribute = "marca"),
        @Component(label = "Modelo: ", type = ComponentType.INPUT, attribute = "modelo"),
        @Component(label = "Ano: ", type = ComponentType.INPUT, attribute = "ano"),
        @Component(label = "Usado: ", type = ComponentType.CHECKBOX, attribute = "usado")
    })
    private CarroModelo carroModelo;

    @Component(label = "Quantidade: ", type = ComponentType.INPUT)
    private int quantidade;

    public CarroController() {
        carroModelo = new CarroModelo();
        carroModelo.setMarca("Chevrolet");
        carroModelo.setModelo("Opala");
        carroModelo.setAno(1980);
        carroModelo.setUsado(true);
        quantidade = 2;
    }
```

Figura 15 – Classe de controle

A Figura 16 apresenta as declarações da anotação *Button* que tiveram como alvo os métodos *excluirCarro*, *editarCarro* e *cadastrarCarro* declarados na classe de controle.

```
    @Button(caption = "Excluir")
    public void excluirCarro() {

    }
    @Button(caption = "Editar")
    public void editarCarro() {

    }
    @Button(caption = "Cadastrar")
    public void cadastrarCarro() {

    }
```

Figura 16 – Declaração das anotações Button

5.3. Classe de Modelo

A Figura 17 apresenta a classe *CarroModelo*, os atributos marca, modelo, ano e usado desta classe foram referenciados pela anotação *Model* declarada na classe de controle, dessa forma, cada atributo referenciado na anotação terá um componente próprio gerado na página XHTML.

```
public class CarroModelo {

    private String marca;
    private String modelo;
    private int ano;
    private boolean usado;

```

Figura 17 – Classe de modelo

5.4. Página Gerada

A Figura 18 apresenta a página gerada com base nas anotações declaradas na classe de controle *CarroController*. Pode-se verificar que os valores dos atributos que foram inicializados no construtor da classe foram apresentados em seus respectivos componentes na página gerada.

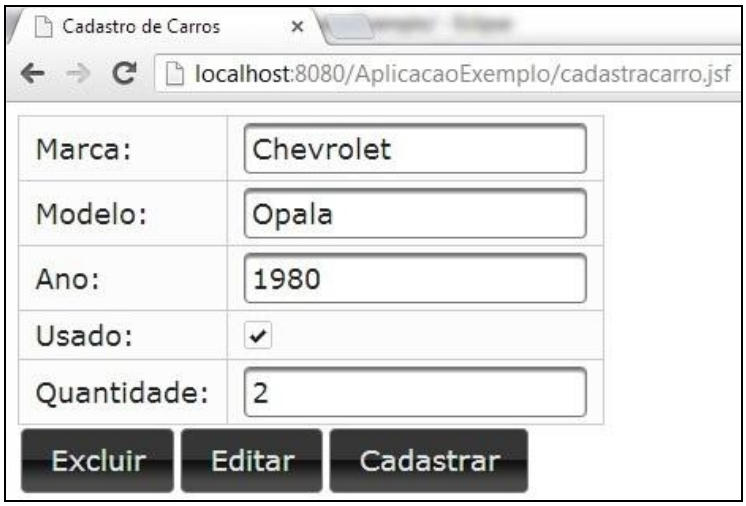

Figura 18 – Página gerada

6. Considerações Finais

Um bom *framework* deve fornecer interfaces bem definidas para que o seu código seja reutilizado por outras aplicações. Sem a utilização do *framework* PrimeFaces para a geração dos componentes de interface não seria possível desenvolver o *framework* FormBuilder no tempo disponível.

O desenvolvimento deste trabalho mostrou que é possível criar um *framework* capaz de gerar páginas web com tecnologias JSF e PrimeFaces. Mesmo realizando a geração de apenas três tipos de componentes, a arquitetura do *framework* possibilita o desenvolvimento de novas anotações e a geração de outros componentes PrimeFaces realizando poucas modificações no *framework*.

Os principais objetivos para adoção dos *frameworks* nas empresas é o ganho da produtividade e a qualidade final das aplicações desenvolvidas. É possível notar no desenvolvimento da aplicação exemplo como é fácil e rápido construir uma página de apresentação, um desenvolvedor sem grandes conhecimentos em JSF e PrimeFaces consegue declarar as anotações em suas classes de controle gerando uma página web sem dificuldades.

Optou-se em não gerar páginas XHTML completas e realizar somente o processamento da *tag* formBuilder, dessa forma o desenvolvedor consegue definir outros componentes PrimeFaces ná página XHTML que não existem referências nas classes de controle, o desenvolvedor também poderia reutilizar páginas já existentes, incluíndo somente a *tag* formBuilder para processamento. A forma utilizada não traz problemas de performance, tendo em vista que todos os componentes JSF são processados da mesma forma, a *tag* formBuilder é tratada como um simples painel onde é adicionado os componentes gerados.

Abaixo segue alguns recursos não presentes no *framework* desenvolvido que podem ser implementados:

- Desenvolver novas anotações e a geração de outros componentes de interface como tabelas, grades, listas, caixa de mensagens e diálogos;
- Desenvolver anotações que definem formatos de *layout* para geração das telas;

- Criar novos parâmetros para as anotações definindo propriedades dos componentes como tamanho e cor;
- Desenvolver recursos para validação de campos e apresentação de mensagens;
- Desenvolver mecanismos para geração de componentes de interface de outros *frameworks* como o RichFaces e IceFaces.

No desenvolvimento do *framework* foi utilizada a IDE Eclipse Indigo e o servidor de aplicação GlassFish 3.1.2. Os códigos fontes utilizados no desenvolvimento do *framework* e da aplicação exemplo estão disponíveis no endereço: https://github.com/silvano-lohn/FormBuilder.

Referências

APACHE. **Apache Isis.** 2013. Disponível em: <http://isis.apache.org>. Acesso em 25 julho 2013.

CALISKAN, Mert; VARAKSIN, Oleg. **PrimeFaces Cookbook.** Birmingham: Packt, 2013.

CHISTY, M. M. I. **An Introduction to Java Annotations.** 2009. Disponível em: <http://www.developer.com/java/other/article.php/3556176/An-Introduction-to-Java-Annotations.htm>. Acesso em: 28 maio 2013.

GEARY, David; HORSTMANN, Cay S. **Core JavaServer Faces.** Saddle River: Prentice Hall, 2010.

GIMENES, I. M. S, HUZITA, E. H. M. **Desenvolvimento baseado em componentes: conceitos e técnicas.** São Paulo: Ciência Moderna, 2005.

HORSTMANN, Cay S. **Padrões e projeto orientados a objetos.** 2. ed. São Paulo: Bookman, 2007.

JACOBI, Jonas; FALLOWS, John. Pro **JSF e Ajax: construindo componentes ricos para a internet.** Rio de Janeiro: Ciência Moderna, 2007.

ORACLE. **The Java EE 6 Tutorial.** Redwood City, 2013. Disponível em: <http://docs.oracle.com/javaee/6/tutorial/doc/gijtu.html>. Acesso em 28 maio 2013.

SOMMERVILLE, Ian. **Engenharia de software.** 8. ed. São Paulo: Pearson, Prentice Hall, 2007.