Framework em Java para Desenvolvimento de Aplicações contendo Janelas Gráficas

Leandro Salvatti Piscke[1], Adilson Vahldick[1]

[1]Departamento de Sistemas e Computação
Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brasil

piscke@gmail.com, adilsonv77@gmail.com

Resumo. *Este artigo descreve a arquitetura de um framework desenvolvido para agilizar e simplificar o desenvolvimento de aplicações desktop utilizando objetos remotos com a tecnologia EJB3. Ao final do artigo é apresentado um exemplo de como utilizar o framework. Esse trabalho também serve como referência na exemplificação de uso dos padrões de projeto e é resultado de um projeto de conclusão de curso de graduação em Ciências da Computação.*

1. Introdução

O desenvolvimento de aplicações exige cada vez mais conhecimento técnico dos programadores, se comparado com as tecnologias usadas nas décadas passadas. Essa tendência vem em função da multiplicidade de ambientes que as aplicações precisam suportar, além de considerável mudança de requisitos por parte dos clientes, seja por ajuste ou alteração do foco nos negócios, seja pelas alterações de legislação.

Um bom ferramental na produção de software precisa considerar esse ciclo de vida de constantes alterações que os sistemas vivenciam. As ferramentas geradoras de código são uma alternativa, contudo elas impõem limitações quando uma aplicação possui muitas particularidades. O ideal é uma solução que ao mesmo tempo agilize o trabalho do desenvolvedor e permita que ele ajuste o grau de automatização, fazendo com que situações comuns utilizem soluções comuns, e problemas diferenciados possam ser tratados pelos programadores.

Os padrões ajudam na produção de software diminuindo o tempo de desenvolvimento, e principalmente no tempo de manutenção, que é considerada a fase mais custosa do ciclo de vida do software. Entretanto, a utilização de padrões exige um alto nível técnico da equipe, além de uma minuciosa documentação e catalogação das classes disponíveis.

Conclui-se que existe um paradoxo no uso de padrões: de um lado é inegável a relevância na sua utilização e do outro, o custo em manter uma equipe com elevado nível técnico e pela disseminação das informações sobre as bibliotecas de classes mantidas por ela.

Na busca de uma solução é proposto neste trabalho um *framework* que permite o desenvolvimento rápido de janelas nas aplicações, e que estes recursos exijam o conhecimento de poucos padrões permitindo que programadores com nível técnico não tão elevado possam fazer parte da equipe, e que ao mesmo tempo programadores mais experientes possam customizar as classes na arquitetura proposta.

Nesse trabalho foram utilizados vários componentes e *frameworks* de terceiros, o que tornou motivador e norteador o desenvolvimento do próprio trabalho, pois um bom *framework* é aquele que pode ser utilizado e reutilizado várias vezes por pessoas e projetos diferentes, e que não tiveram influência nenhuma na produção dele.

2. Arquitetura do Framework

Uma das características desse *framework* é a disponibilização pela Internet das interfaces com o usuário. Por essa razão, ele foi dividido em duas versões: uma para execução no servidor e outra para execução nas máquinas clientes. A figura 1 apresenta as duas versões e a divisão em camadas dentro de cada uma delas.

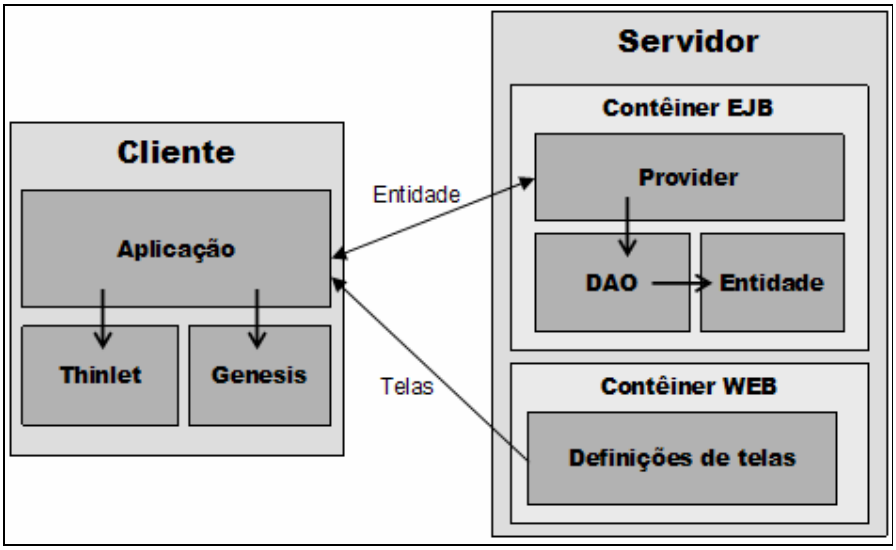

Figura 1. Arquitetura do framework

2.1. Aplicação Servidor

A versão do servidor usa os dois tipos de contêiner da especificação JEE. No contêiner EJB é onde se concentram as classes que o desenvolvedor utilizará e estenderá do *framework*. A figura 2 apresenta o diagrama com essas classes. Esse conjunto de classes foi estruturado seguindo uma arquitetura em camadas, conforme será descrito nos parágrafos abaixo.

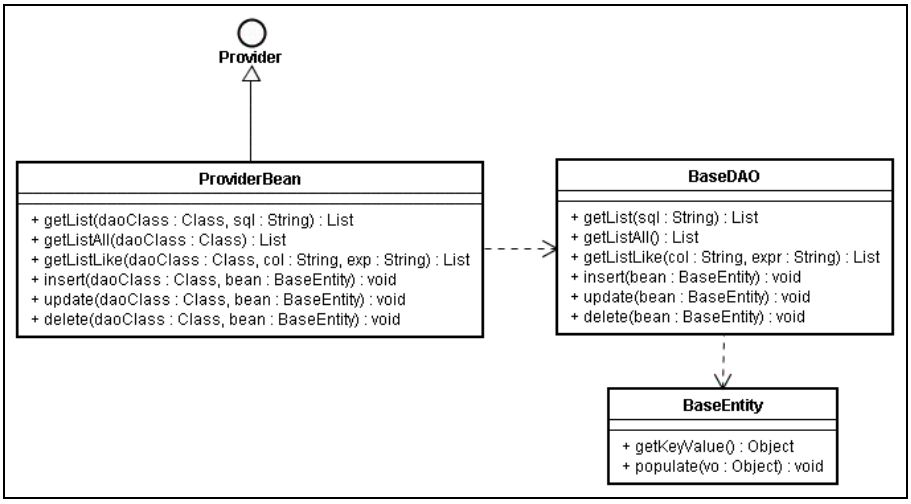

Figura 2. Classes da versão servidor

Toda classe que representa uma entidade (ou classe de modelo) precisa estender da classe `BaseEntity` e utilizar as anotações de entidades do EJB3. Os objetos dessa classe são transferidos entre a aplicação servidora e a aplicação cliente. Essa classe segue o padrão *Transfer Object* (ALUR, CRUPI E MALKS, 2004).

A persistência dos objetos entidade é gerenciada pelos objetos das classes que estendem de `BaseDAO`. A implementação de classes descendentes só se faz necessário quando os métodos oferecidos por essa classe base não atenderem. Por exemplo, a classe `BaseDAO` possui uma série de métodos para retornar uma lista de objetos (`getList`, `getListAll` e `getListLike`). Se o programador precisar de uma lista com um critério diferente, então ele precisa implementar esse método na classe descendente de `BaseDAO`. Essa classe segue o padrão *Data Access Object* (ALUR, CRUPI E MALKS, 2004).

A classe `ProviderBean` e a interface `Provider` foram definidas para a responsabilidade de desempenhar a tarefa de disponibilizar os serviços do *framework*. A aplicação cliente se comunicará com objetos de classes descendentes de `ProviderBean`, implementadas pelo programador usuário do *framework.* As classes descendentes precisam utilizar as anotações de objetos sem estado (*stateless*) do EJB3. A implementação dessas classes segue o padrão *Session Façade* (ALUR, CRUPI E MALKS, 2004; FOWLER, 2006) e *Business Object* (ALUR, CRUPI E MALKS, 2004).

No contêiner web foi desenvolvida uma aplicação para gerenciar as definições das telas da aplicação, armazenadas como arquivos XML. A utilização desses arquivos é feita pela aplicação cliente conforme descrita na próxima seção. Através do navegador o administrador adiciona, altera e remove os arquivos XML das telas da aplicação. A figura 3 apresenta a página principal dessa aplicação web.

Figura 3. Página principal da aplicação Web de Gerenciamento de Telas

Existem partes das janelas que se repetem, e poderiam ser agrupadas e consideradas como um único componente. Um exemplo é uma barra de botões para representar ações de navegação pelos registros de uma tabela, além de iniciar qualquer atividade de inclusão, alteração ou exclusão também conhecidas como padrão CRUD (YODER, JOHNSON E WILSON, 1998). Não só partes se repetem, mas depois que o desenvolvedor estabeleceu um padrão de interfaces com o usuário, ele acaba copiando o arquivo XML inteiro para criar uma nova janela.

Como uma alternativa para utilizar de componentização foram implementadas algumas *Custom Tags* (SUN..., 2007) de exemplo para inspirar o desenvolvedor na produção padronizada das janelas. Nesse caso, em vez de dispor de um arquivo XML, o programador monta uma página JSP que produzirá o XML ao cliente. Cada *Custom Tag* representa um conjunto de componentes do *Thinlet*. Foram implementadas as seguintes *Custom Tags*:

- `EditSearch`: concentra um painel com um rótulo, um campo de edição e um botão;
- `LabelLookup`: contém um rótulo seguido de um painel com um campo de edição e um botão;
- `LabelCombobox` e `LabelEditTag`: ambos correspondem a um rótulo seguido de uma caixa de seleção e campo de edição respectivamente.

2.2. Aplicação Cliente

A aplicação cliente foi desenvolvida para prover telas com interface gráfica. Para isso foi utilizado o *Thinlet*, que é um *toolkit* GUI leve para Java (BAJZAT, 2006). Em uma única classe Java processa a hierarquia e propriedades da interface gráfica, trata interação com usuário e permite invocar métodos de objetos específicos da aplicação.

Para desenvolver uma interface é necessário editar um arquivo XML que descreva os atributos dos componentes suportados pelo *Thinlet*. Essas telas também podem ser projetadas utilizando a ferramenta *ThinG*, que foi desenvolvida com o próprio *Thinlet* (MÖBIUS, 2007). A figura 4 apresenta a interface principal dessa ferramenta.

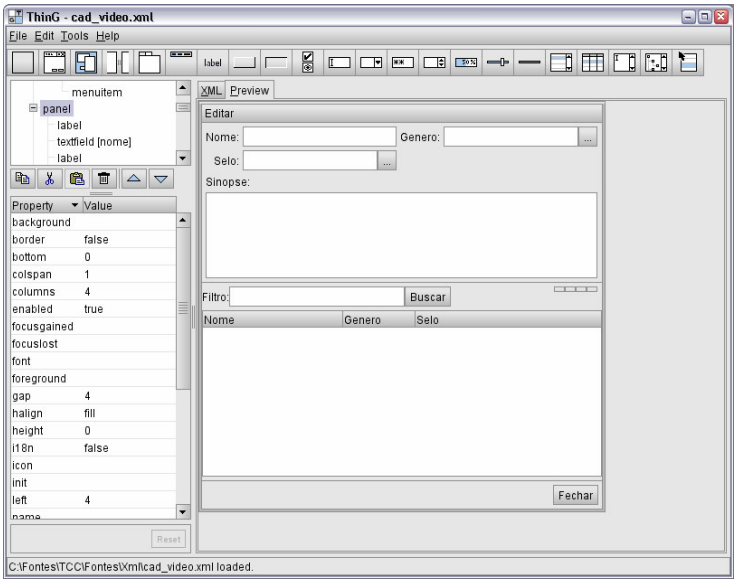

Figura 4. Tela principal do ThinG

As classes do *framework* para a aplicação cliente estão representadas na figura 5.

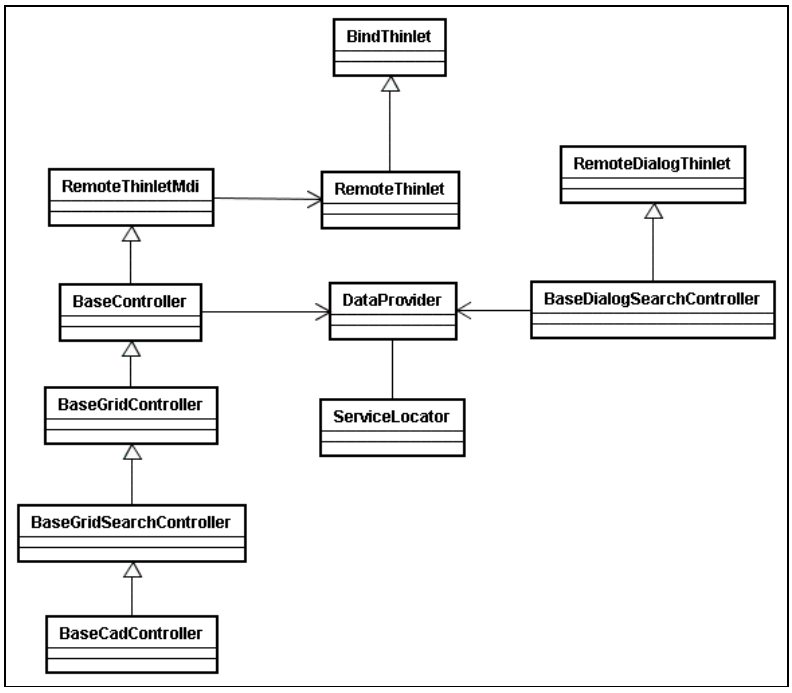

Figura 5. Classes da aplicação cliente

A classe `RemoteThinlet` é responsável por solicitar as definições das telas ao servidor e utilizar o *Thinlet* para a montagem das telas. A solicitação dos documentos XML foi simplificado com o uso do componente `HTTPClient` do projeto Jakarta (APACHE..., 2007b). Essa montagem está programada na classe `BindThinlet` que é ancestral de `RemoteThinlet`. A `BindThinlet` é descendente de uma classe de outro *framework* que é o *Genesis* (SUMMA..., 2006) que implementa o processo de ligação entre as telas *Thinlet* com objetos no padrão *JavaBeans*. A classe `BindThinlet` implementa o padrão *Adapter* (GAMMA et al, 2000), pois recebe um objeto de entidade e mapeia para o formato exigido da classe `BaseThinlet` do *Genesis*. Nesse mapeamento foi utilizado o componente `BeanUtils` do projeto Jakarta (APACHE..., 2007a).

Os objetos de controle da aplicação cliente são instâncias da classe `BaseController` ou descendentes. Essa classe estabelece uma ponte com os serviços implementados na aplicação servidora, delegando essas tarefas a objetos do tipo `DataProvider` e `ServiceLocator`. A classe `DataProvider` implementa o padrão *Business Delegate* (ALUR, CRUPI E MALKS, 2004) e é responsável em comunicar-se com os serviços remotos do *framework*. A classe `ServiceLocator` é que fornece os objetos remotos ao `DataProvider` comunicando-se com o servidor.

As classes descendentes de `BaseController`, como `BaseGridController`, `BaseGridSearchController` e `BaseCadController`, são padrões de tela com recursos como uma grade para apresentar a lista de registros, uma barra com configurações de pesquisa e funções básicas de inclusão, alteração e exclusão, também conhecido como modelo CRUD (YODER, JOHNSON E WILSON, 1998). O desenvolvedor pode simplesmente utilizar essas classes ou estendê-las quando a janela possuir algum recurso adicional.

3. Estudo de Caso

Para demonstrar a utilização do protótipo foi implementado um sistema de vídeo-locadora atendendo os casos de uso apresentados na figura 6. O modelo conceitual desse sistema está representado na figura 7.

Foram definidas cinco etapas para utilizar o *framework*, quatro delas na aplicação servidora que são (i) implementar as entidades, (ii) classes DAO, (iii) regras de negócios e (iv) criar definições de telas, e mais outra na aplicação cliente que é a implementação das classes de controle.

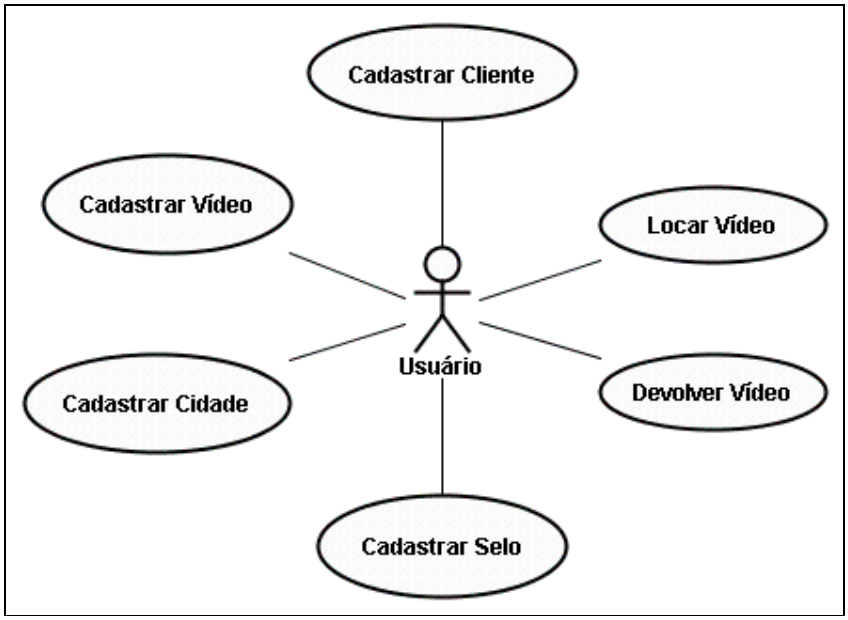

Figura 6. Diagrama de casos de uso do sistema de vídeo-locadora

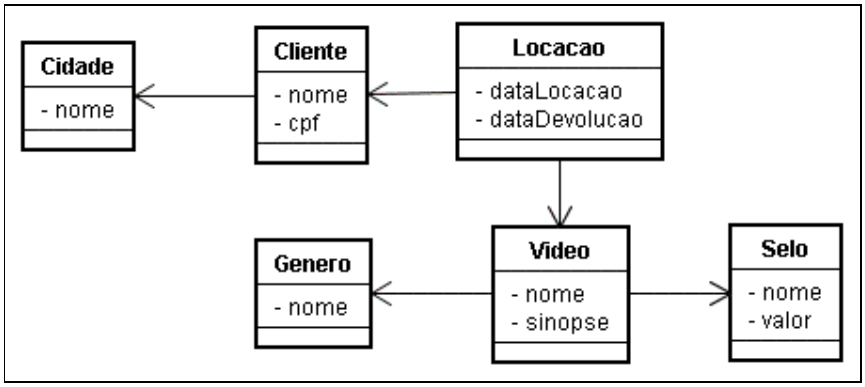

Figura 7. Modelo conceitual do sistema de vídeo-locadora

Na primeira etapa são implementadas as classes de entidades. Uma delas identificada no protótipo é a classe `Locacao`. A figura 8 apresenta parte do código dessa classe. Essa classe segue o padrão *JavaBeans* e são adicionadas anotações EJB3 de entidade para desempenhar o mapeamento objeto-relacional.

```
// imports...
@Entity
@Table(name = "locacao")
public class Locacao extends BaseEntity implements Serializable{

   private int codigo;
   private Cliente cliente;
   // demais atributos

   @Id
   @GeneratedValue
   public int getCodigo() { return codigo; }
   public void setCodigo(int codigo) { this.codigo = codigo; }

   @ManyToOne(optional = false)
```

```
   public Cliente getCliente() { return cliente; }
   public void setCliente(Cliente cliente) { this.cliente = cliente; }

   // ...
}
```

Figura 8. Entidade locação

A segunda etapa envolve a implementação de classes DAO. Basicamente basta estender da classe `BaseDAO`. No caso da locação, para retornar os vídeos locados e ainda não devolvidos de um cliente, foi necessário adicionar um novo método que é o `getVideos` (figura 9).

```
// imports
public class LocacaoDAO extends BaseDAO<Locacao> {

   public LocacaoDAO(EntityManager entityMnger) { super(entityMnger); }
   public LocacaoDAO() { super(); }

   public List<Locacao> getVideos(Cliente cli){
       Query q = getEM().createQuery("from Locacao where " +
                 "cliente.codigo=:codigo and dtDevolucao is null");
       q.setParameter("codigo", cli.getCodigo());
       return q.getResultList();
   }
}
```

Figura 9. Classe DAO para a entidade locação

Na terceira etapa acontece o desenvolvimento das regras de negócio, onde essas classes estendem da classe `ProviderBean` e implementam um EJB de sessão. Também é necessário definir uma interface com os serviços a serem disponibilizados à aplicação cliente, pois a referência será pela interface e não pela classe. Cada caso de uso poderia corresponder a uma classe de serviços. A figura 10 apresenta a classe de serviços de locação.

```
// imports
public @Stateless
class LocadoraProvider extends ProviderBean implements Provider,
       LocadoraInterface {

       public void devolucao(Locacao loc) {
          LocacaoDAO dao = new LocacaoDAO(getEM());
          loc.setDtDevolucao(new Date());
          dao.update(loc);
       }

       public List<Locacao> getVideos(Cliente cli) {
          LocacaoDAO dao = new LocacaoDAO(getEM());
          return dao.getVideos(cli);
       }
// outros métodos
}
```

Figura 10. Classe de serviços do cadastro de locação

A etapa seguinte é a definição das telas. Inicialmente as telas foram desenvolvidas com o *ThinG* e depois migradas para arquivos JSP, demonstrando a utilização das *Custom Tags*. A figura 11 apresenta o código da janela de locação e a figura 12 a representação gráfica dessa janela. Foi adicionado inclusive a *Custom Tag*

`labellookup` nessa janela. Os conteúdos dos atributos `action` correspondem aos métodos que a janela invocará dos objetos controladores.

```
<%@ page language="java" contentType="text/xml; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://piscke.server.org/tags" prefix="fk"%>
<dialog columns="1" gap="4" left="4" top="4" right="4" height="300"
                                resizable="true" scrollable="true" width="400">
    <panel columns="2" gap="4" left="4" right="4">
       <fk:labellookup name="cliente" buttonText="..." text="Cliente:"
                                      action="showCliente"></fk:labellookup>
    </panel>
    <separator/>
    <panel columns="2">
       <panel halign="left" right="4" weightx="1">
         <button action="locacao" text="Locação" tooltip="Locação" />
         <button action="devolucao" text="Devolução" tooltip="Devolução" />
       </panel>
       <panel halign="right" right="4" weightx="1">
         <button action="delete" alignment="right" halign="right"
                                   icon="img/excluir.gif" tooltip="Excluir" />
       </panel>
    </panel>
    <table name="grid" weighty="1" weightx="1">
        <header>
            <column name="video" text="Video"/>
            <column name="video.selo.valor" text="Valor"/>
        </header>
    </table>
    <separator/>
    <panel columns="2">
      <panel halign="right" right="4" weightx="1">
        <label alignment="right" text="Total:"/>
        <textfield editable="false" name="total"/>
      </panel>
    </panel>
    <jsp:include page="inc/fechar.jsp"></jsp:include>
</dialog>
```

Figura 11. Definição da tela de locação e devolução de vídeos

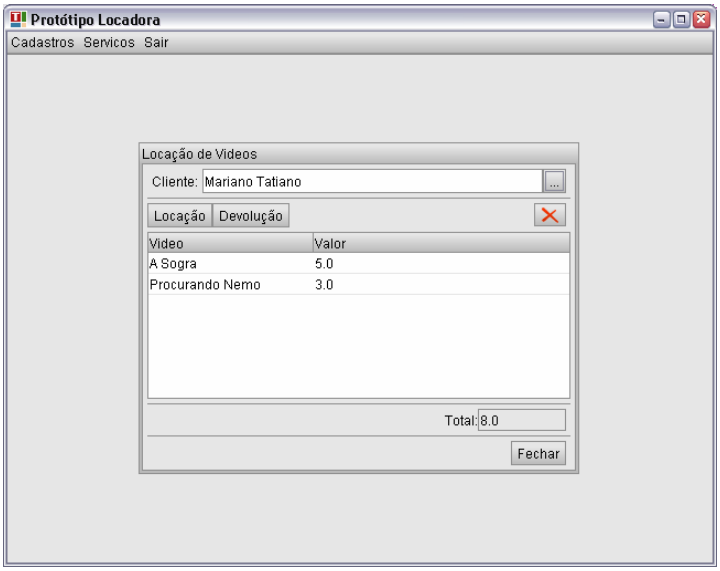

Figura 12. Representação gráfica da tela principal e da tela de locação e devolução de vídeos

A última etapa é a implementação das classes controladoras. Elas devem estender das classes descritas na seção 2.2. No caso da locação foi criada uma classe descendente de `BaseGridController`. A figura 13 apresenta a classe `LocacaoController` ilustrando dois métodos para os requisitos de locação e devolução. Deve-se observar que esses métodos foram relacionados na definição da tela citada na figura 11.

```
// imports
public class LocacaoController extends BaseGridController<Locacao> {

   public void devolucao() throws ScreenNotFoundException{
      Locacao obj = getSelected();
      if (obj != null) {
         locaServ.devolucao(obj);
         populateGrid(locaServ.getVideos(cliente));
      } else
         MessageDialog.show(getThinlet(), "Informação",
                                                "Não há item selecionado!");
   }

   public void locacao() throws Exception{
      if (cliente == null){
         MessageDialog.show(getThinlet(),"Informação","Selecione um cliente!");
      }else{
         BaseDialogSearchController<Video> vid =
             new BaseDialogSearchController<Video>(getThinlet(), "lookup.jsp",
                          new DataProvider(VideoDAO.class), "grid", "nome");
         if (vid.showSearch()) {
            Locacao loc = new Locacao();
            loc.setCliente(cliente);
            loc.setVideo(vid.getObj());
            setText(find("video"), vid.getObj().getNome());
            setText(find("cliente"), cliente.getNome());
            locaServ.locacao(loc);
            populateGrid(locaServ.getVideos(cliente));
         }
         atualizaSoma();
      }
   }
   // outros métodos
}
```

Figura 13. Classe de controle do cadastro de locação

Como janela principal do sistema foi implementado um descendente de `RemoteThinlet` e também definido um documento XML contendo as opções do menu.

4. Conclusões

Na implementação do estudo de caso a preocupação foi voltada em atender o modelo conceitual e os casos de uso. Isso demonstrou que o foco do desenvolvedor foi quanto à lógica de negócio, deixando a infraestrutura por conta do *framework*. Esse sistema está pronto para execução em um ambiente multiusuário e ao mesmo tempo independente de plataforma.

Outro fato que se pode comprovar é que a quantidade de conhecimento para a utilização do *framework* é consideravelmente baixa, permitindo que programadores iniciantes possam fazer parte da equipe. Exige-se do programador o entendimento das

anotações de EJB de entidade e que classes do *framework* devem ser estendidas para implementar a aplicação. Ao mesmo tempo, programadores mais experientes podem customizar suas janelas estendendo das classes de controle e implementando novas *Custom Tags*.

Várias tecnologias foram imprescindíveis no sucesso desse *framework*. Pode-se destacar o *Thinlet*, o *Genesis* e o EJB3, que permitiram transparência quanto a disponibilização de serviços remotos e quanto ao mapeamento objeto-relacional. As classes `BaseEntity` e `ProviderBean` foram os EJBs implementados: o primeiro de entidade e o segundo de sessão. Nesse projeto também foi desenvolvida a classe `BindThinlet` é descendente de uma classe do *Genesis*. O *Thinlet* é utilizado indiretamente, visto que o `BindThinlet` faz a comunicação entre o documento XML que contém as definições de tela (no formato *Thinlet*) e o próprio *Thinlet* que monta as telas.

Comparando esse trabalho em relação ao *Genesis*, o *framework* aqui desenvolvido destaca-se por utilizar as especificações EJB3 e facilitar o desenvolvimento de telas no modelo CRUD. O *Genesis* é mais genérico e não implementa as facilidades para o desenvolvimento de telas no modelo CRUD. Porém, destaca-se por possuir suporte a diferentes bibliotecas gráficas como: *Thinlet*, SWT e *Swing*.

Esse trabalho possui algumas limitações como formatação de campos, onde não é possível definir máscaras para os campos, e notificação das telas *Thinlet* quando o modelo é alterado para que as telas possam ser recarregadas. Essa limitação pode ser resolvida aplicando o padrão *Observer* (GAMMA et al, 2000).

Referências

Alur, D., Crupi, J. and Malks, D. Core J2EE patterns: as melhores práticas e estratégias de design. Tradução Altair Dias Caldas de Moraes. Rio de Janeiro: Campus, 2004.

Apache Software Foundation. Commons BeanUtils. [S.l.], 2007a. Disponível em: <http://jakarta.apache.org/commons/beanutils/>. Acesso em: 14 set. 2006.

_____. HttpClient. [S.l.], 2007b. Disponível em: <http://jakarta.apache.org/commons/httpclient/>. Acesso em: 11 maio. 2007.

Bajzat, R. Thinlet. [S.l.], 2006. Disponível em: <http://thinlet.sourceforge.net/home.htmlThinlet>. Acesso em: 18 maio. 2007.

Fowler, M. Padrões de arquitetura de aplicações corporativas. Tradução Acauan Fernandes. Porto Alegre: Artmed, 2006.

Gamma, E. et al. Padrões de projeto: soluções reutilizáveis de software orientado a objetos. Tradução Luiz A. Meirelles Salgado. Porto Alegre: Bookman, 2000.

Möbius, D. ThinG - a GUI Editor for Thinlet. [S.l.], 2007. Disponível em: <http://thing.sourceforge.net/ >. Acesso em: 15 maio. 2007.

Summa Tecnologies do Brasil. Genesis. [S.l.], 2006. Disponível em: <http://genesis.dev.java.net/nonav/3.0-EA3/maven-site/pt-BR/index.html>. Acesso em: 27 ago. 2006.

Sun Developer Network. Custom Tags in JSP Pages. [S.l.], 2007. Disponível em: <http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags.html>. Acesso em: 12 maio. 2007.

Yoder, J., Johnson, R. and Wilson, Q. Connecting business objects to relational database. In: CONFERENCE ON PATTERNS LANGUAGES OF PROGRAMS, 5th, 1998, Monticello. Proceedings**...** Illinois: Dept. of Computer Science, 1998. p. 9-11. Disponível em: <http://www.joeyoder.com/papers/patterns/PersistentObject/Persista.pdf>. Acesso em: 27 out. 2006.