

**UNIVERSIDADE REGIONAL DE BLUMENAU**  
**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**  
**CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO**

**FRAMEWORK EM JAVA PARA GERAÇÃO DE TELAS NO**  
**MODELO CRUD BASEADO EM XML E OBJETOS**  
**REMOTOS UTILIZANDO A ARQUITETURA MVC E**  
**PADRÕES**

**LEANDRO SALVATTI PISCHE**

**BLUMENAU**  
**2007**

**2007/1-24**

**LEANDRO SALVATTI PISCHE**

**FRAMEWORK EM JAVA PARA GERAÇÃO DE TELAS NO  
MODELO CRUD BASEADO EM XML E OBJETOS  
REMOTOS UTILIZANDO A ARQUITETURA MVC E  
PADRÕES**

Trabalho de Conclusão de Curso submetido à  
Universidade Regional de Blumenau para a  
obtenção dos créditos na disciplina Trabalho  
de Conclusão de Curso II do curso de Ciências  
da Computação — Bacharelado.

Prof. Adilson Vahldick, Especialista - Orientador

**BLUMENAU  
2007**

**2007/1-24**

**FRAMEWORK EM JAVA PARA GERAÇÃO DE TELAS NO  
MODELO CRUD BASEADO EM XML E OBJETOS  
REMOTOS UTILIZANDO A ARQUITETURA MVC E  
PADRÕES**

Por

**LEANDRO SALVATTI PISCHE**

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente: \_\_\_\_\_  
Prof. Adilson Vahldick, Especialista – Orientador, FURB

Membro: \_\_\_\_\_  
Prof. Marcel Hugo, Mestre – FURB

Membro: \_\_\_\_\_  
Prof. Jomi Fred Hübner, Doutor – FURB

Blumenau, 10 de julho de 2007

Dedico este trabalho a todos aqueles que me ajudaram diretamente na realização deste, em especial a minha família e amigos.

## **AGRADECIMENTOS**

A Deus, por me amar incondicionalmente e ter me mostrado o sentido da vida.

À minha família, que sempre me apoiou e esteve ao meu lado.

Aos meus amigos, que me incentivaram para a conclusão do curso e foram companheiros durante esse período.

Ao meu orientador, Adilson Vahldick, pelo conhecimento compartilhado e dedicação na orientação desse trabalho.

Pois a loucura de Deus é mais sábia que a sabedoria humana, e a fraqueza de Deus é mais forte que a força do homem.

I Coríntios 1:25

## RESUMO

Este trabalho apresenta um *framework* para geração de telas no modelo CRUD usando uma arquitetura no padrão MVC. Os parâmetros de entrada para o *framework* são arquivos XML, contendo as definições de telas no modelo CRUD, e objetos no padrão DTO, possuindo os dados que são mostrados na tela gerada. O *framework* desenvolvido tem como objetivo economizar tempo e facilitar o desenvolvimento de sistemas de informação. No trabalho são abordados padrões de projetos como: *Session Facade*, *Business Object*, *Data Access Object*, *Transfer Object*, *Composite View*, *Service Locator* e *Business Delegate*, detalhando como foram aplicados no desenvolvimento do *framework*. No projeto são utilizados outros *frameworks* e componentes que são: *Thinlet*, *Genesis*, *HttpClient*, *BeansUtil* e *Custom Tag*. Também é apresentada a especificação EJB3 e como foi utilizada na implementação do projeto.

Palavras-chave: *Framework*. EJB3. MVC. *Thinlet*. Padrões de projeto.

## ABSTRACT

This work presents a framework to generate screens in the CRUD model using a MVC pattern architecture. The framework input parameters are XML files containing the screens definitions in the CRUD model, and objects in the DTO pattern, containing the data showed in the generated screen. The developed framework has as objective to economize time and to facilitate the development of information systems. This work presents patterns like: Session Facade, Business Object, Data Access Object, Transfer Object, Composite View, Service Locator e Business Delegate, detailing how they were applied in the development framework. The project used others frameworks and components that are: Thinlet, Genesis, HttpClient, BeansUtil e *Custom Tag*. It also presents the EJB3 specification and how it was used in the project implementation.

Key-words: *Framework*. EJB3. MVC. Thinlet. Patterns.



## LISTA DE ILUSTRAÇÕES

Figura 1 – Fluxo de eventos em informações em uma arquitetura MVC.....	16
Figura 2 – Diagrama de classes da estratégia <i>Session Facade</i> .....	18
Figura 3 – Diagrama de classes representando os relacionamentos do padrão DAO .....	19
Figura 4 – Diagrama de classes representando o padrão <i>Composite View</i> .....	20
Figura 5 – Diagrama de classes representando o padrão <i>Service Locator</i> .....	21
Figura 6 – Diagrama de classes representando o padrão <i>Bussiness Delegate</i> .....	21
Quadro 1 – Código de uma interface POJO .....	23
Quadro 2 – Código de uma POJO sem estado .....	23
Quadro 3 – Código de uma EJB de entidade.....	24
Quadro 4 – Código <code>calculator.xml</code> de uma tela Thinlet.....	26
Figura 7 – Interface de uma calculadora utilizando Thinlet.....	26
Quadro 5 – <code>Calculator.java</code> .....	26
Quadro 6 – Classe simples.....	27
Quadro 7 – Utilizando <code>BeanUtils</code> .....	27
Quadro 8 – Exemplo de utilização do <code>HttpClient</code> .....	28
Figura 8 – Padrões utilizados no <i>framework</i> .....	32
Figura 9 – Diagrama de classes da camada servidor do <i>framework</i> .....	33
Figura 10 – Diagrama de seqüência da camada servidora do <i>framework</i> .....	35
Figura 11 – Diagrama web da aplicação <i>FrameworkWeb</i> .....	36
Figura 12 – Diagrama de classes de <i>Custom Tags</i> do <i>framework</i> .....	37
Figura 13 – Diagrama de classes da camada cliente do <i>framework</i> .....	38
Quadro 9 – Código da classe <code>BaseDAO</code> .....	41
Quadro 10 – Código da classe <code>ProviderBean</code> .....	43
Quadro 11 – Código da classe <code>LabelEdit</code> .....	44
Quadro 12 – Código da classe <code>RemoteThinlet</code> .....	45
Quadro 13 – Código da classe <code>BindThinlet</code> .....	46
Quadro 14 – Código da classe <code>BaseGridController</code> .....	47
Figura 14 – Diagrama de casos de uso do protótipo .....	48
Figura 15 – Diagrama de classes da camada servidora do protótipo.....	50
Figura 16 – Diagrama de classes da camada cliente do protótipo.....	51

Quadro 15 – Código da classe Cliente do protótipo.....	53
Quadro 16 – Código da classe LocacaoDAO do protótipo .....	54
Quadro 17 – Código da classe LocadoraProvider do protótipo.....	55
Quadro 18 – Código do JSP cad_cliente.jsp do protótipo.....	56
Quadro 19 – Código do XML do cadastro de clientes do protótipo.....	57
Figura 17 – Tela do cadastro de clientes do protótipo.....	58
Figura 18 – Tela do cadastro de vídeos .....	59
Figura 19 – Tela do cadastro de cidades.....	60
Quadro 20 – Arquivo server.properties .....	60
Quadro 21 – Código da classe ClienteController do protótipo.....	61

## LISTA DE SIGLAS

API – *Application Programming Interface*

CRUD – *Create Retrieve Update Delete*

DAO - *Data Access Object*

DTO - *Data Transfer Object*

DTD - *Document Type Definition*

EJB - *Enterprise JavaBeans*

GUI - *Graphical User Interface*

JAVAAEE - *Java Enterprise Edition*

JEMS - *Enterprise Middleware System*

JNDI - *Java Naming and Directory Interface*

JSP – *Java Server Pages*

MVC – *Model View Controller*

MDI - *Multiple Document Interface*

POJO - *Plain Old Java Objects*

SGBD - *Sistema de Gerenciamento de Banco de Dados*

XML - *eXtensible Markup Language*

## SUMÁRIO

<b>1 INTRODUÇÃO.....</b>	<b>13</b>
1.1 OBJETIVOS DO TRABALHO .....	14
1.2 ESTRUTURA DO TRABALHO .....	15
<b>2 FUNDAMENTAÇÃO TEÓRICA .....</b>	<b>16</b>
2.1 MVC.....	16
2.2 CRUD.....	16
2.3 PADRÕES DE PROJETO .....	17
2.3.1 Padrões e <i>frameworks</i> .....	17
2.3.2 Padrões no servidor .....	18
2.3.3 Padrões no cliente .....	20
2.4 EJB3 .....	22
2.4.1 Definindo um EJB de sessão.....	23
2.4.2 Definindo um EJB de entidade .....	23
2.5 <i>FRAMEWORKS</i> E COMPONENTES UTILIZADOS.....	25
2.5.1 Thinlet .....	25
2.5.2 BeanUtils.....	27
2.5.3 HttpClient.....	27
2.5.4 <i>Custom Tags</i> .....	28
2.6 TRABALHOS CORRELATOS .....	28
2.6.1 Genesis .....	29
2.6.2 PATI-MVC.....	29
<b>3 DESENVOLVIMENTO DO TRABALHO.....</b>	<b>31</b>
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO.....	31
3.2 ESPECIFICAÇÃO .....	32
3.2.1 Introdução ao <i>framework</i> .....	32
3.2.2 <i>Framework</i> para a camada servidora .....	32
3.2.3 Fornecimento do XML.....	36
3.2.4 <i>Framework</i> para a camada cliente.....	37
3.3 IMPLEMENTAÇÃO .....	40
3.3.1 Ferramentas utilizadas.....	40
3.3.2 Implementação do <i>framework</i> .....	40

3.3.2.1 Camada servidora .....	41
3.3.2.2 Camada web.....	44
3.3.2.3 Camada cliente.....	44
3.4 VALIDAÇÃO DO <i>FRAMEWORK</i> .....	48
3.4.1 Introdução ao protótipo .....	48
3.4.2 Camada servidora do protótipo .....	49
3.4.3 Camada web do protótipo .....	51
3.4.4 Camada cliente do protótipo .....	51
3.4.5 Utilizando o <i>framework</i> .....	52
3.4.5.1 Implementar objetos relacionais .....	52
3.4.5.2 Implementar classes DAO .....	54
3.4.5.3 Implementar regras de negócio.....	54
3.4.5.4 Criar XML das telas do sistema.....	55
3.4.5.5 Implementar classes controladoras .....	60
3.5 RESULTADOS E DISCUSSÃO .....	62
<b>4 CONCLUSÕES .....</b>	<b>64</b>
4.1 EXTENSÕES .....	64
<b>REFERÊNCIAS BIBLIOGRÁFICAS .....</b>	<b>66</b>

## 1 INTRODUÇÃO

Com a evolução dos sistemas de informação, diariamente tem-se deparado com arquiteturas de software complexas, que são desafios para projetistas e desenvolvedores. Sistemas mal arquitetados e sem uso de padrões<sup>1</sup> no desenvolvimento podem acarretar vários problemas na solução final. Segundo Ahmed e Umrysh (2003, p. 5), sistemas desenvolvidos misturando camadas como regras de negócio, visualização e persistência trazem limites impostos na reutilização dessas partes, por essas estarem fortemente acopladas.

Sistemas de informação possuem em sua maior parte consulta, alteração e exclusão de dados. De acordo com Cockburn (2005, p. 145), essas funcionalidades seguem o modelo CRUD. Geralmente esse modelo repete-se em várias partes do sistema. A duplicação de funcionalidades pode consumir tempo no desenvolvimento de funções já conhecidas além de gerar outros problemas no projeto, como rotinas repetidas contendo a mesma solução e telas desenvolvidas sem uma padronização. No desenvolvimento de software, na fase de análise, Cockburn (2005, p. 145) descreve que ao invés de fazer vários casos de uso para cada modelo CRUD, pode-se escrever um caso de uso parametrizado, sendo possível assim representar vários casos de uso CRUD a partir de um modelo.

Tendo como base as idéias de Cockburn, esse trabalho tem como objetivo facilitar o desenvolvimento do modelo CRUD em sistemas de informações. O trabalho elabora um *framework* em Java, para a geração automática de telas no modelo CRUD usando uma arquitetura no padrão MVC<sup>2</sup>. Os parâmetros de entrada para o *framework* são arquivos XML, contendo as definições de telas no modelo CRUD, e objetos no padrão DTO<sup>3</sup>, possuindo os dados que são mostrados na tela gerada. Os arquivos XML contendo as definições das telas devem ser criados pelo próprio usuário do *framework*, assim como os DTO, que podem ser implementados como objetos relacionais que mapeiam a estrutura de uma tabela do banco de dados, ou como objetos que contenham valores extraídos de outra fonte de dados.

O *framework* implementa as três camadas da arquitetura MVC. A camada do modelo é

---

<sup>1</sup> De acordo com Alur, Crupi e Malks (2004, p. 10), “Um padrão é documentado quando a solução que ele oferece pode ser usada várias vezes para solucionar problemas similares em diferentes momentos e em diferentes projetos.”

<sup>2</sup> Conforme Gamma et al. (2000, p. 20), o MVC possui três camadas: Modelo, Visão e Controle. O Modelo é a camada de aplicação contendo os dados e a lógica de negócio. A Visão é a apresentação de tela ao usuário. O Controle define a maneira como a interface do usuário reage às entradas do mesmo.

<sup>3</sup> Segundo Alur, Crupi e Malks (2004, p. 374), um DTO serve para a transferência de dados entre camadas. Ele contém todos os elementos de dados em uma única estrutura tanto para solicitações quanto para respostas.

desenvolvida utilizando a plataforma JavaEE e alguns padrões de projetos, tais como *Data Access Object*, *Transfer Object*, *Session Facade* e *Business Object*. Essa camada provê a lógica de negócios do *framework*, sendo responsável por expor serviços que possibilitem o acesso remoto às definições da tela, gerenciamento da persistência e acesso aos DTO.

A camada da visão é responsável por gerar uma tela CRUD a partir do XML provido pela camada de modelo. Na camada de visão é usada uma biblioteca gráfica chamada Thinlet<sup>4</sup>, que implementa facilidades para o desenvolvimento de telas tendo como definição um arquivo XML.

A ligação entre a camada da visão e a camada do modelo é feita pela camada de controle, a qual requisita a definição da tela e o DTO ao modelo, relacionando a tela gerada com os dados do DTO. Essa camada implementa padrões de projetos como: *Service Locator* e *Business Delegate*.

Gamma et al. (2000, p. 42) afirma que um *framework* captura as decisões de projetos que são comuns ao domínio de aplicação. Assim, *frameworks* enfatizam reutilização de projetos em relação à reutilização de código. Com o uso de *frameworks* pode-se construir aplicações mais rapidamente, permitindo que desenvolvedores concentrem-se na parte pertinente do projeto, como a lógica de negócio.

Uma questão importante do *framework* proposto está associada à economia de tempo no desenvolvimento de um sistema de informação. Através do mesmo pretende-se disponibilizar uma infra-estrutura para facilitar a construção de telas no modelo CRUD, de forma que bastará definir uma tela através de um arquivo XML, programar um objeto no padrão DTO e utilizar o *framework* para fazer a ligação do DTO à tela gerada.

## 1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um *framework* em Java para a geração automática de telas no modelo CRUD usando uma arquitetura no padrão MVC.

Os objetivos específicos do trabalho são:

- a) implementar a parte essencial do *framework* na camada de modelo. Esta executará em um servidor de aplicação JavaEE e será responsável pela localização das

---

<sup>4</sup> Thinlet é um *framework* para construção de interfaces gráficas AWT representadas através de um XML (BAJZAT, 2005).

- definições das telas, instanciação dos DTO e gerenciamento da persistência;
- b) implementar a camada cliente do *framework*, composta pela camada de visão e de controle;
  - c) definir atributos no XML que serão referências para fazer a ligação com o DTO;
  - d) disponibilizar recursos para que o desenvolvedor possa acessar os componentes visuais da tela gerada;
  - e) disponibilizar recursos para o desenvolvedor acessar os dados do DTO;
  - f) desenvolver o protótipo de um sistema de locação de vídeos utilizando o *framework* proposto.

## 1.2 ESTRUTURA DO TRABALHO

No capítulo seguinte são apresentados alguns conceitos, tecnologias e ferramentas que são utilizadas nesse trabalho. O capítulo 3 descreve a especificação e desenvolvimento do trabalho, também são avaliados os resultados obtidos no desenvolvimento. Por último, no capítulo 4, são apresentadas as conclusões para o trabalho, bem como propostas de extensões para trabalhos futuros.

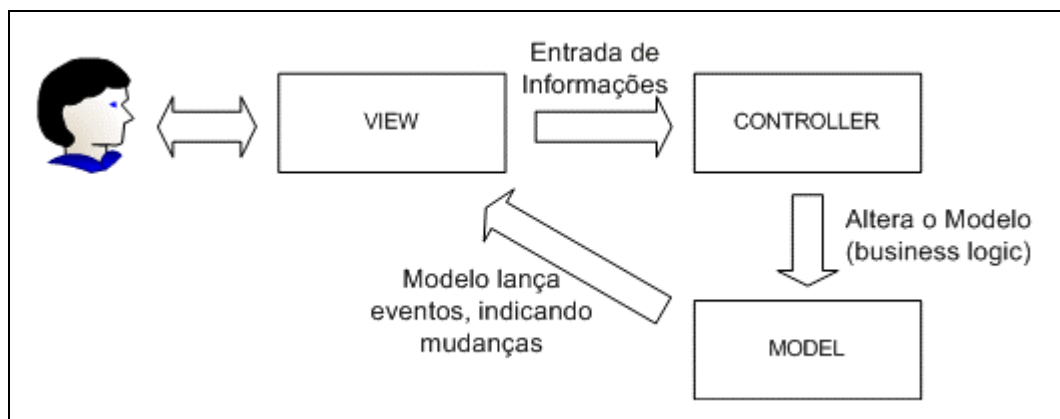


## 2 FUNDAMENTAÇÃO TEÓRICA

A seguir são explanados os principais assuntos relacionados ao desenvolvimento do *framework*, os quais são: MVC, CRUD, Padrões e *Framework*, Padrões de projeto, *Frameworks* e Componentes de Terceiros utilizados e Trabalhos Correlatos.

### 2.1 MVC

Segundo Gamma et al (2000, p. 20), antes do MVC, os projetos de interface para o usuário tendiam a agrupar os objetos de modelo, visão e controle. MVC separa esses objetos para aumentar a flexibilidade e a reutilização. A separação entre os objetos é estabelecida por um protocolo de inserção/notificação entre eles, na qual uma visão deve garantir que a sua aparência reflita o estado do modelo, de tal modo que quando os dados do modelo mudam, esse notifica as visões que dependem dele. Em resposta, cada visão tem a oportunidade de atualizar-se. Essa abordagem permite que uma aplicação possua várias visões para um modelo sem precisar reescrevê-lo. A figura 1 mostra o fluxo de eventos em uma arquitetura MVC.



Fonte: Almeida (2007).

Figura 1 – Fluxo de eventos em informações em uma arquitetura MVC

### 2.2 CRUD

Segundo Yoder, Johnson e Wilson (1998, p. 9), objetos persistentes precisam de

operações de leitura e escrita em um banco de dados. Às vezes, objetos precisam ser excluídos de seu armazenamento persistente. Conseqüentemente, quando um objeto necessita ser persistido ou excluído, é importante fornecer um mínimo de operações, para criar, ler, atualizar e excluir os dados. O modelo CRUD sugere as operações básicas para a persistência de objetos.

Um exemplo da utilização do modelo CRUD em objetos de persistência, de acordo com Yoder, Johnson e Wilson (1998, p. 9), pode ser descrito como objetos de um mesmo domínio que estendem de uma classe comum, a qual define as operações do modelo CRUD. Dessa maneira, as subclasses implementam seu comportamento quando as operações de criação, leitura, atualização e exclusão forem chamadas, padronizando assim as subclasses que possuem as operações básicas de persistência de objetos.

## 2.3 PADRÕES DE PROJETO

Nas seções seguintes são aprestados os padrões utilizados no trabalho desenvolvido, assim como a relação entre padrões de projetos e *frameworks*.

### 2.3.1 Padrões e *frameworks*

Segundo Gamma et al. (2000, p. 42), um *framework* implementado através do uso de padrões de projeto tem muito maior probabilidade de atingir altos níveis de reusabilidade de projeto e código, comparado com um que não usa padrões de projeto. Um benefício adicional de um *framework* que implementa padrões é que pessoas que conhecem os padrões obtêm rapidamente uma compreensão do *framework*.

De acordo com Gamma et al. (2000, p. 43), padrões e *frameworks* têm algumas similaridades, sendo que as pessoas frequentemente se perguntam em que os mesmos diferem. Eles são diferentes em três aspectos principais:

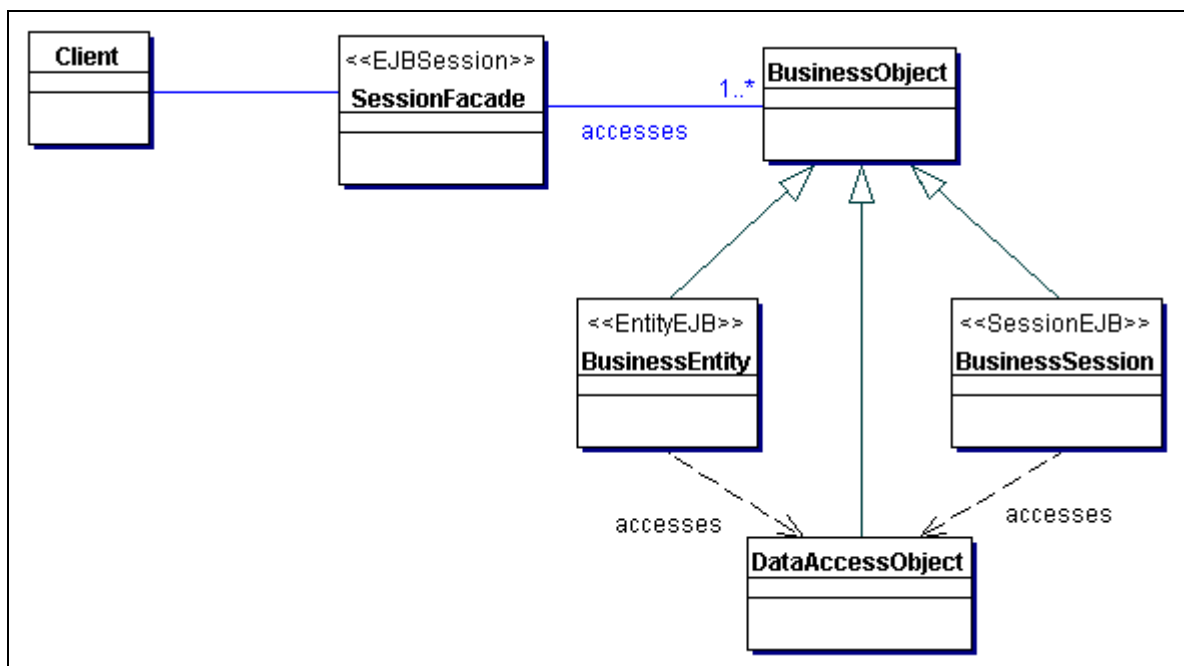
- a) padrões de projetos são mais abstratos que *frameworks*. Os *frameworks* podem ser escritos em uma linguagem de programação, sendo não apenas estudados, mas executados e reutilizados. Em contraposição, os padrões de projeto têm que ser implementados cada vez que eles são usados;

- b) padrões de projeto são elementos de arquitetura menores que *frameworks*. Um *framework* típico contém vários padrões de projeto, mas a recíproca nunca é verdadeira;
- c) padrões de projeto são menos especializados que *frameworks*. Os *frameworks* são voltados para a solução de um domínio de aplicação. Padrões podem ser usados em qualquer tipo de aplicação.

### 2.3.2 Padrões no servidor

Os padrões preliminarmente identificados na parte servidor do *framework* são: *Session Facade*, *Business Object*, *Data Access Object*, *Transfer Object* e *Composite View*.

O padrão *Session Facade*, segundo Alur, Crupi e Malks (2004, p. 303), serve para encapsular os componentes da camada de negócios e expor um serviço de granulação fina aos clientes remotos. Os clientes acessam um *Session Facade* em vez de acessar diretamente os componentes de negócio. Fowler (2006, p. 371) diz que o uso mais comum deste padrão é entre uma apresentação de interface para o usuário e um modelo de domínio, onde os dois podem rodar em processos diferentes. A figura 2 mostra o diagrama de classes representando o padrão *Session Facade*.



Fonte: Sun Developer Network (2007a).

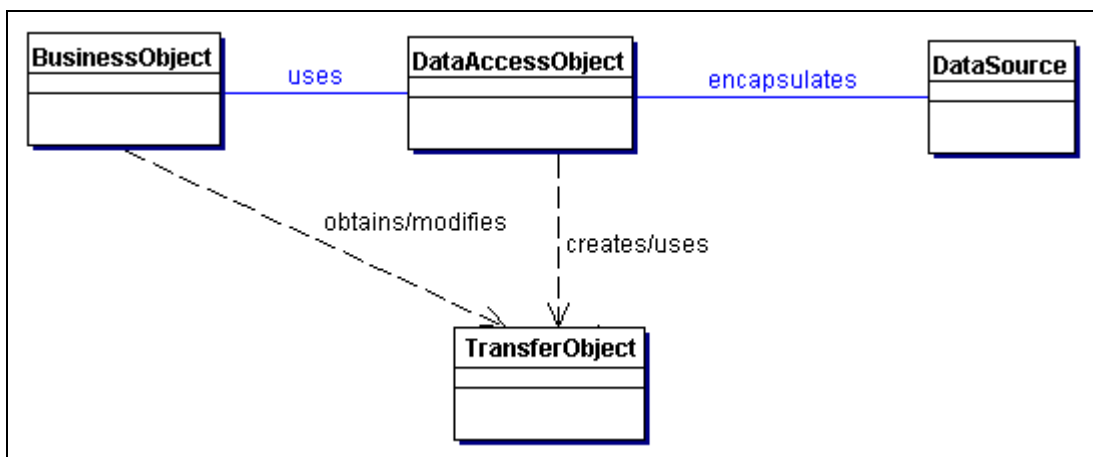
Figura 2 – Diagrama de classes da estratégia *Session Facade*

Os *Business Objects*, de acordo com Alur, Crupi e Malks (2004, p. 334), encapsulam e

gerenciam os dados de negócios, comportamento e persistência. Os *Business Objects* ajudam a separar a lógica de persistência da lógica de negócio. Os *Business Objects* mantêm os dados de negócios principais e implementam o comportamento que é comum a toda a aplicação. A figura 3 mostra um diagrama de classes utilizando o padrão *Business Objects*.

Alur, Crupi e Malks (2004, p. 417) definem *Data Access Object* um padrão que serve para abstrair e encapsular todo acesso ao armazenamento persistente. O padrão *Data Access Object* gerencia a conexão com a fonte de dados para obter e armazenar dados. A figura 3 mostra o diagrama de classes representando os relacionamentos do padrão *Data Access Object*.

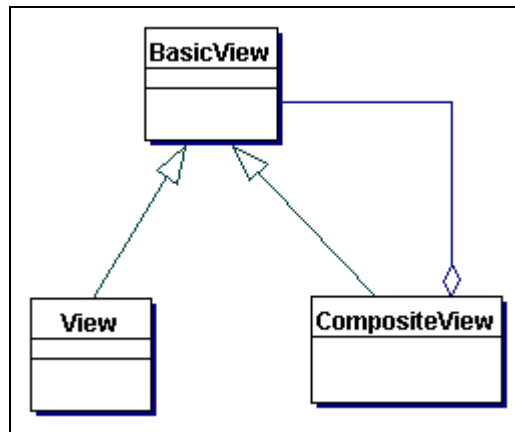
Segundo Alur, Crupi e Malks (2004, p. 374), o padrão *Transfer Object* é projetado para otimizar a transferência de dados entre camadas. Em vez de enviar ou receber elementos de dados individuais, um *Transfer Object* contém todos os elementos de dados em uma única estrutura requerida pela solicitação ou resposta. A figura 3 mostra o diagrama de classes utilizando o padrão *Transfer Object*.



Fonte: Sun Developer Network (2007a).

Figura 3 – Diagrama de classes representando os relacionamentos do padrão DAO

O padrão *Composite View* segundo Alur, Crupi e Malks (2004, p. 232), representa visões compostas de múltiplas sub-visões atômicas. Cada componente independente pode ser incluído dinamicamente e o leiaute da visão passa a ser mais facilmente gerenciável independentemente do conteúdo. Esta abordagem promove a reutilização de porções atômicas da visão e facilita em muito a alteração do leiaute. A figura 4 mostra o diagrama de classes do padrão *Composite View*.



Fonte: Sun Developer Network (2007a).

Figura 4 – Diagrama de classes representando o padrão *Composite View*

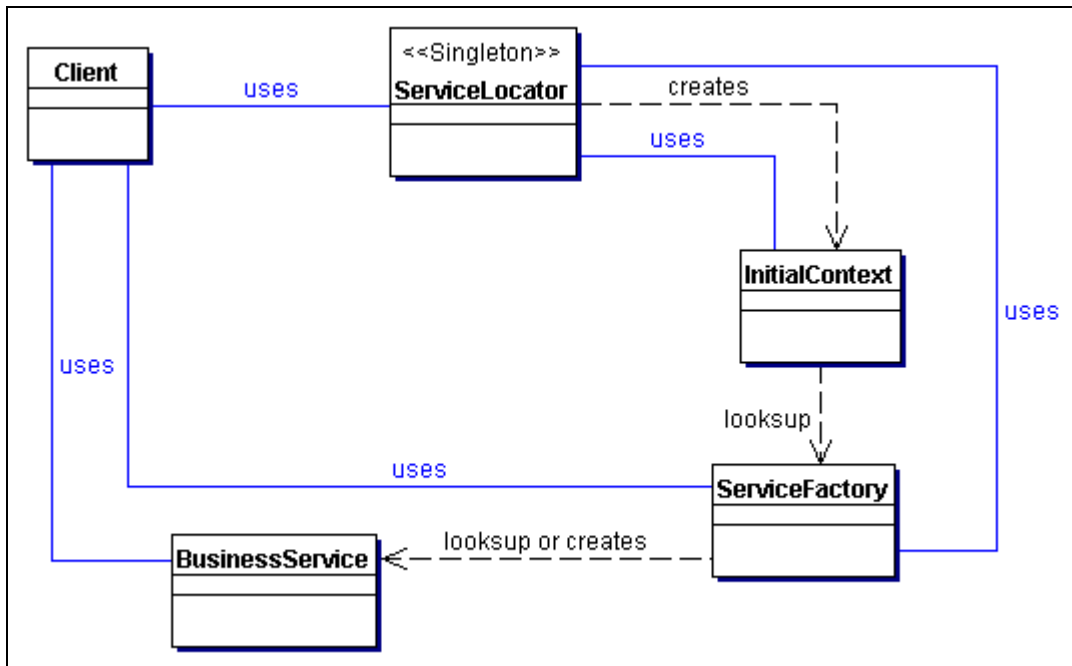
### 2.3.3 Padrões no cliente

Os padrões preliminarmente identificados na parte cliente do *framework* são: *Service Locator* e *Business Delegate*.

O padrão *Service Locator* abstrai as chamadas JNDI<sup>5</sup> e esconde as complexidades inerentes à criação de um contexto inicial e busca dos componentes. Este objeto pode ser utilizado em toda parte de uma aplicação que precisa acessar um recurso gerenciado por JNDI, reduzindo a complexidade do código, provendo um ponto único de controle. A figura 5 mostra o diagrama de classes do padrão *Service Locator*.

---

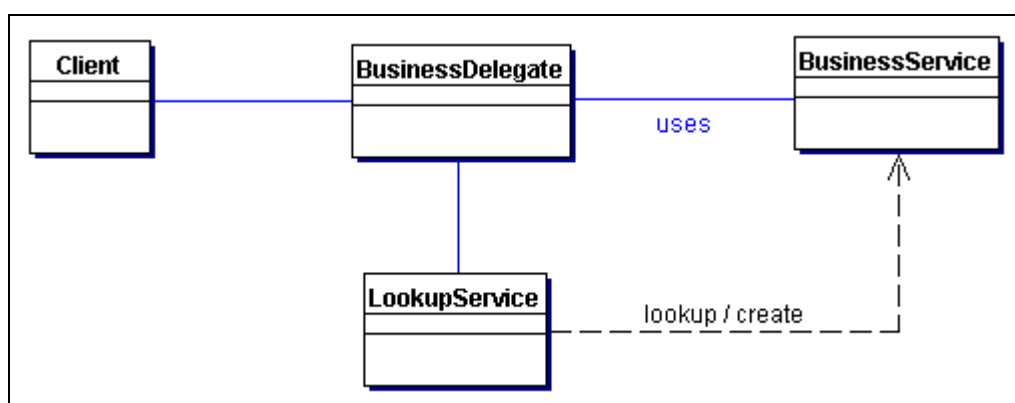
<sup>5</sup> JNDI é uma API para acesso a serviços de diretórios. Ela permite que aplicações cliente descubram e obtenham dados ou objetos através de um nome (SUN DEVELOPER NETWORK (2007d)).



Fonte: Sun Developer Network (2007a).

Figura 5 – Diagrama de classes representando o padrão *Service Locator*

De acordo com Alur, Crupi e Malks (2004, p. 268), o padrão *Business Delegate* serve para encapsular o acesso a um serviço de negócios. O *Business Delegate* oculta os detalhes de implementação do serviço de negócios, como mecanismos de pesquisa e acesso, tornando transparentes as chamadas de serviços orientados a negócio que estão em um servidor. A figura 6 mostra o diagrama de classes do padrão *Business Delegate*. O cliente faz uma requisição ao *Business Delegate* que provê acesso aos serviços de negócio. O *Business Delegate* usa um *LookupService* para localizar um componente de serviço de negócio.



Fonte: Sun Developer Network (2007a).

Figura 6 – Diagrama de classes representando o padrão *Business Delegate*

## 2.4 EJB3

EJB é uma arquitetura de componentes para o desenvolvimento de aplicações Java, que resolve muito dos problemas como escalabilidade, aplicações distribuídas e gerenciamento de ciclo de vida dos objetos. Com o objetivo de facilitar o trabalho do desenvolvedor para que não tenha que se preocupar com aspectos de infra-estrutura da aplicação. Essa infra-estrutura é resolvida pelos servidores de aplicação que implementam as especificações da arquitetura EJB (BOND et al, 2003, p. 111).

Segundo Bond et al. (2003, p. 113), existem três tipos diferentes de EJB, que são:

- a) EJB de sessão: são uma tecnologia importante dentro da plataforma JavaEE, pois permitem que a funcionalidade do negócio seja desenvolvida e depois implantada, independente da camada de interface com o usuário;
- b) EJB de entidade: assumem a responsabilidade de representar os dados do domínio. Ainda irá existir um depósito de dados persistentes para gerenciar os dados, quase certamente um SGBD, mas os EJB de entidade abstraem e ocultam os detalhes do mecanismo de persistência;
- c) EJB dirigido por mensagens: um EJB dirigido por mensagens é conceitualmente muito parecido com um EJB de sessão, mas é ativado apenas através mensagens assíncronas. Um exemplo de utilização de um EJB dirigido por mensagens seria um servidor de e-mails.

O EJB3 é uma revisão e uma simplificação profunda da especificação do EJB. Os objetivos do EJB3 são: simplificar e facilitar o desenvolvimento dirigido para teste, e focalizar mais no desenvolvimento de POJO<sup>6</sup>. EJB3 adota inteiramente as anotações<sup>7</sup> do Java introduzidas na versão 5.0 do JDK e simplifica também os EJB de entidade usando ferramentas para gerenciar a persistência (RED HAT, 2007a).

---

<sup>6</sup> De acordo com Alur, Crupi e Malks (2004, p. 374), POJO é um objeto Java normal que não implementa nenhuma interface nem estende nenhuma classe específica.

<sup>7</sup> Anotações são palavras-chaves colocadas na classe Java, iniciadas com o caractere @, que têm por objetivo simplificar o desenvolvimento, possibilitando que em tempo de compilação sejam carregados ou executados classes, permitindo acessar com facilidade propriedades e métodos das classes SUN DEVELOPER NETWORK (2007e).

### 2.4.1 Definindo um EJB de sessão

Para definir um EJB de sessão utilizando a especificação EJB3, primeiramente é necessário definir uma interface contendo todos os métodos de negócio. A interface do EJB de sessão é simplesmente uma interface POJO sem anotações. O cliente usa essa interface para acessar a referência remota do EJB de sessão que está em um *container* EJB3 (RED HAT, 2007b). O quadro 1 mostra como ficaria o código de uma interface POJO de uma calculadora.

```
public interface Calculator {
    public double calculate(int start, int end, double growthrate,
        double saving);
}
```

Fonte: Red Hat (2007b).

Quadro 1 – Código de uma interface POJO

Com a interface definida, pode-se agora implementar um simples POJO. O *container* EJB3 instancia e gerencia automaticamente as instâncias dos POJO. O quadro 2 mostra o código de uma classe que implementa a interface `Calculator` apresentada no quadro 1. A anotação `@Stateless` indica que o EJB de sessão é sem estado<sup>8</sup> (RED HAT, 2007b).

```
@Stateless
public class StatelessCalculator implements Calculator {
    public double calculate(int start, int end, double growthrate,
        double saving) {
        double tmp = Math.pow(1. + growthrate / 12., 12. * (end -
            start) + 1);
        return saving * 12. * (tmp - 1) / growthrate;
    }
}
```

Fonte: Red Hat (2007b).

Quadro 2 – Código de uma POJO sem estado

### 2.4.2 Definindo um EJB de entidade

Um EJB de entidade utilizando EJB3 é simplesmente um POJO com anotações que especificam como ele será armazenado em uma base de dados. O mapeamento entre um

<sup>8</sup> Um EJB de sessão sem estado não mantém os valores do objeto para um cliente particular. Seu estado é mantido apenas durante a chamada do método SUN DEVELOPER NETWORK (2007b).



objeto e uma tabela do banco de dados é feito automática e transparentemente pelo *container* EJB3. O desenvolvedor não precisa se preocupar com os detalhes do gerenciamento de conexão, nem com os acessos específicos as APIs do banco de dados utilizado. Colocando apenas uma anotação `@Entity` em uma classe POJO, a mesma se torna um EJB de entidade. O container EJB3 mapeia cada EJB de entidade para uma única tabela do banco de dados. A anotação `@Table` diz ao *container* qual o nome da tabela que ele deve fazer relação. Cada instância de um EJB de entidade representa uma linha de dados de uma tabela. E cada coluna da tabela, corresponde ao atributo de um EJB de entidade. No quadro 3 é mostrado uma EJB de entidade fazendo relação com uma tabela “fund” (RED HAT, 2007b).

```

@Entity
@Table(name = "fund")
public class Fund implements Serializable {
    private int id;

    private String name;

    private double growthrate;

    public Fund() {
    }

    public Fund(String name, double growthrate) {
        this.name = name;
        this.growthrate = growthrate;
    }

    @Id
    @GeneratedValue
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    // Outros métodos getter e setter...
}

```

Fonte: Red Hat (2007b).

Quadro 3 – Código de uma EJB de entidade

A anotação `@Id` especifica que o atributo é a chave primária da tabela. A anotação `@GeneratedValue` indica que a chave primária é gerada automaticamente pelo servidor. Nessa classe o nome dos atributos são os mesmos das colunas da tabela. Porém utilizando a

anotação `@Column(name="nome da coluna")` se faz o relacionamento com colunas da tabela que não tenham o mesmo nome dos atributos da classe EJB3 (RED HAT, 2007b).

Para distribuir um EJB de entidade, deve-se empacotar as classes em um arquivo JAR. O EJB3 também requer um arquivo `persistence.xml` no diretório META-INF do arquivo JAR. O arquivo `persistence.xml` define qual banco de dados é usado com os EJB de entidade (RED HAT, 2007b).

## 2.5 FRAMEWORKS E COMPONENTES UTILIZADOS

Nas seções seguintes são apresentados *frameworks* e componentes que foram utilizados na implementação do *framework*.

### 2.5.1 Thinlet

Thinlet é um *toolkit* GUI leve para Java. Em uma única classe Java, ele processa a hierarquia e propriedades da interface gráfica, trata interação com usuário e permite invocar métodos de objetos específicos da aplicação. Thinlet separa apresentação gráfica (descrita em um arquivo XML) dos métodos de aplicação (escritos em código Java) (BAJZAT, 2005).

Para desenvolver uma interface utilizando Thinlet é necessário editar um arquivo XML que descreva os atributos dos componentes suportados pelo Thinlet. O exemplo do quadro 4 mostra o XML de uma interface para uma calculadora (quadro 5). O componente raiz é um `panel` que contém três `textfields`, um `label (+)` e um `button (=)`. Para tratamento dos eventos do componente `button` foi adicionado o atributo `action` na `tag`, que tem a informação do método que será chamado para tratar o evento dentro de uma classe Java. O resultado do XML interpretado pelo Thinlet é mostrado na figura 7 (BAJZAT, 2005).

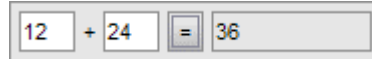
```

<panel gap="4" top="4" left="4">
  <textfield name="number1" columns="4" />
  <label text="+" />
  <textfield name="number2" columns="4" />
  <button text="=" action="calculate(number1.text,
    number2.text, result)" />
  <textfield name="result" editable="false" />
</panel>

```

Fonte: Bajzat (2005).

Quadro 4 – Código calculator.xml de uma tela Thinlet



Fonte: Bajzat (2005).

Figura 7 – Interface de uma calculadora utilizando Thinlet

A classe Java desenvolvida para gerar a interface da figura 7 é mostrada no quadro 5, a qual estende a classe `Thinlet` e implementa o método `calculate()` para tratar o evento do componente `button` da interface. No construtor da classe é carregado o arquivo `calculator.xml` e adicionado ao `Thinlet`. No método `main()` da classe é criado um `frame` e incluído o componente `Calculator`. O método `calculate()` soma os dois primeiros campos e atualiza o terceiro campo com o valor calculado.

```

package thinlet.demo;

import thinlet.*;

public class Calculator extends Thinlet {

    public Calculator() throws Exception {
        add(parse("calculator.xml"));
    }

    public static void main(String[] args) throws Exception {
        new FrameLauncher("Calculator", new Calculator(), 320, 240);
    }

    public void calculate(String number1, String number2, Object
        result) {
        try {
            int i1 = Integer.parseInt(number1);
            int i2 = Integer.parseInt(number2);
            setString(result, "text", String.valueOf(i1 + i2));
        } catch (NumberFormatException nfe) {
            getToolkit().beep();
        }
    }
}

```

Fonte: Bajzat (2005).

Quadro 5 – Calculator.java

### 2.5.2 BeanUtils

A maioria dos desenvolvedores Java cria suas classes nomeando as propriedades no padrão *getters* e *setters*. O componente BeanUtils do projeto Jakarta tem como objetivo facilitar o acesso dinâmico de objetos que implementem esse padrão. O componente BeanUtils usa a reflexão do Java para acessar as propriedades das classes dinamicamente. (APACHE SOFTWARE FOUNDATION, 2007a). No quadro 6 é mostrada uma classe Java simples, usada pelo Objeto `PropertyUtils` para atribuir e resgatar atributos da mesma, conforme mostrado no quadro 7.

```
public class Employee {
    private String firstName;
    private String lastName;

    public String getFirstName();

    public void setFirstName(String firstName);

    public String getLastName();

    public void setLastName(String lastName);
}
```

Quadro 6 – Classe simples

```
Employee employee = new Employee();
employee.setFirstName("Leandro");
employee.setLastName("Piscke");

// Resgatar atributos da classe
String fName = (String)
    PropertyUtils.getProperty(employee, "firstName");
String lName = (String)
    PropertyUtils.getProperty(employee, "lastName");

// Atribuir atributos da classe
PropertyUtils.setProperty(employee, "firstName", fName);
PropertyUtils.setProperty(employee, "lastName", lName);
```

Quadro 7 – Utilizando BeanUtils

### 2.5.3 HttpClient

Embora o pacote `java.net` provê funcionalidade básica para acessar recursos por HTTP, ele não provê a flexibilidade completa ou funcionalidade que muitas aplicações precisam. O componente `HttpClient` do projeto Jakarta provê um pacote eficiente, com características que implementa o lado cliente dos mais recentes padrões de HTTP e

recomendações. O componente de HttpClient pode ser de interesse a qualquer um construindo aplicações de cliente HTTP como *browsers*, cliente *web service* ou sistemas que influenciam ou estendem o protocolo HTTP para comunicação distribuída. (APACHE SOFTWARE FOUNDATION, 2007b). O quadro 8 mostra um código da utilização do componente HttpClient.

```

HttpClient httpClient = new HttpClient();
GetMethod httpget = new GetMethod("http://www.myhost.com/");
try {
    httpClient.executeMethod(httpget);
    Reader reader = new InputStreamReader(
        httpget.getResponseBodyAsStream(),
        httpget.getResponseCharSet());

    // ... consumir a resposta da página
} finally {
    httpget.releaseConnection();
}

```

Quadro 8 – Exemplo de utilização do HttpClient

#### 2.5.4 Custom Tags

A tecnologia JSP provê um mecanismo para encapsular outros tipos de funcionalidade dinâmica às *Custom Tags*, que são extensões da linguagem JSP. Uma *Custom Tag* é uma *tag* definida pelo usuário. Quando uma JSP contém uma *Custom Tag* a mesma é traduzida em um *servlet* e convertida para um objeto o qual é a implementação da *Custom Tag*. O *container* Web invoca as operações desse objeto quando o *servlet* da JSP é executado (SUN DEVELOPER NETWORK, 2007c).

## 2.6 TRABALHOS CORRELATOS

A seguir são apresentados dois trabalhos relacionados ao proposto: projeto Genesis e os padrões PATI-MVC.

### 2.6.1 Genesis

O Genesis é um *framework* desenvolvido em Java que tem como objetivo principal simplificar e tornar produtivo o desenvolvimento de aplicações corporativas. Ele simplifica o desenvolvimento de componentes de negócio e construção de interfaces gráficas complexas com o mínimo esforço do desenvolvedor da aplicação (SUMMA TECNOLOGIES DO BRASIL, 2006). Segundo Summa Technologies do Brasil (2006), as principais características do *framework* são:

- a) *binding swing* e Thinlet: através do *binding* é possível exibir e manter JavaBeans sincronizados com o estado da tela. Além disso, pode-se ligar métodos a botões apenas com uma anotação;
- b) modelo simples para carregar componentes visuais: escrevendo um método que retorne uma lista ou matriz, o *framework* responsabiliza-se em carregar componentes visuais como listas e tabelas;
- c) remotabilidade transparente: permite implementar chamadas remotas de classes que não implementam nem estendem nenhuma classe POJO com o simples uso de anotações. No lado do cliente, seus objetos podem ser instanciados e usados normalmente.

### 2.6.2 PATI-MVC

Trata-se de padrões no desenvolvimento de aplicações de sistema de informação baseados na arquitetura MVC. De acordo com Souza, Pires e Barros (2003), os padrões CRUD-MVC participam de uma família de padrões dedicada a construção de sistemas de informação, apresentando relacionamentos e interações entre classes do modelo, visão e controle para realização de operações CRUD e suporte para outras operações de negócio. Dentre os principais padrões, estão:

- a) manutenção em grade: define características de interação com o usuário a partir de objetos genéricos de interface para manipulação de uma coleção de entidades de negócio em grade, onde a grade é semelhante a uma tabela e os atributos das entidades são mostrados em colunas, permitindo comandos de edição, criação, atualização e exclusão de registros, sincronizando a grade com a coleção de

entidades;

- b) seleção secundária: utilizado para entidades principais ou secundárias de média ou alta complexidade com vários atributos e relacionamentos em situações que requerem a seleção de entidades sob determinado critério, para somente depois realizar edição ou inclusão em objetos de interface, e não diretamente sobre a grade.

### 3 DESENVOLVIMENTO DO TRABALHO

O presente capítulo descreve a especificação e implementação do projeto, detalhando aspectos importantes acerca da forma como os padrões de projetos, *frameworks*, componentes e ferramentas foram utilizadas no desenvolvimento do projeto.

#### 3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O *framework* deverá:

- a) gerar uma tela a partir de um XML e um DTO seguindo o modelo CRUD (Requisito Funcional - RF);
- b) implementar funcionalidades do modelo CRUD na camada de controle (RF);
- c) implementar a camada de controle para ser extensível pelo usuário (RF);
- d) disponibilizar métodos e propriedade na classe controle que forneçam para o desenvolvedor acesso aos componentes da tela gerada (RF);
- e) disponibilizar métodos e propriedades na classe de controle para que o desenvolvedor acesse os dados do DTO (RF);
- f) implementar um objeto que oculte a interação com o servidor JavaEE (RF);
- g) instanciar e gerenciar a persistência dos DTO na camada de modelo (RF);
- h) implementar um mecanismo de busca do XML no servidor (RF);
- i) ser implementado utilizando a especificação JavaEE (Requisito Não-Funcional - RNF);
- j) utilizar a arquitetura de sistema MVC (RNF);
- k) utilizar padrões *Session Facade*, *Business Object*, *Data Access Object*, *Transfer Object*, *Composite View*, *Service Locator* e *Business Delegate* no desenvolvimento (RNF).



## 3.2 ESPECIFICAÇÃO

Nas subseções seguintes são abordadas as especificações do *framework* desenvolvido, que se divide em: introdução ao *framework*, *framework* para a camada servidora, fornecimento do XML e *framework* para a camada cliente. Para a diagramação do projeto foi utilizada a ferramenta Enterprise Architect por suportar recursos da versão 5.0 do JDK.

### 3.2.1 Introdução ao *framework*

Para melhor entendimento da arquitetura do *framework*, a figura 8 mostra os padrões que são utilizados na camada cliente e servidora do *framework*. Na figura 8 também está representada a organização de como os padrões foram utilizados nas duas camadas.

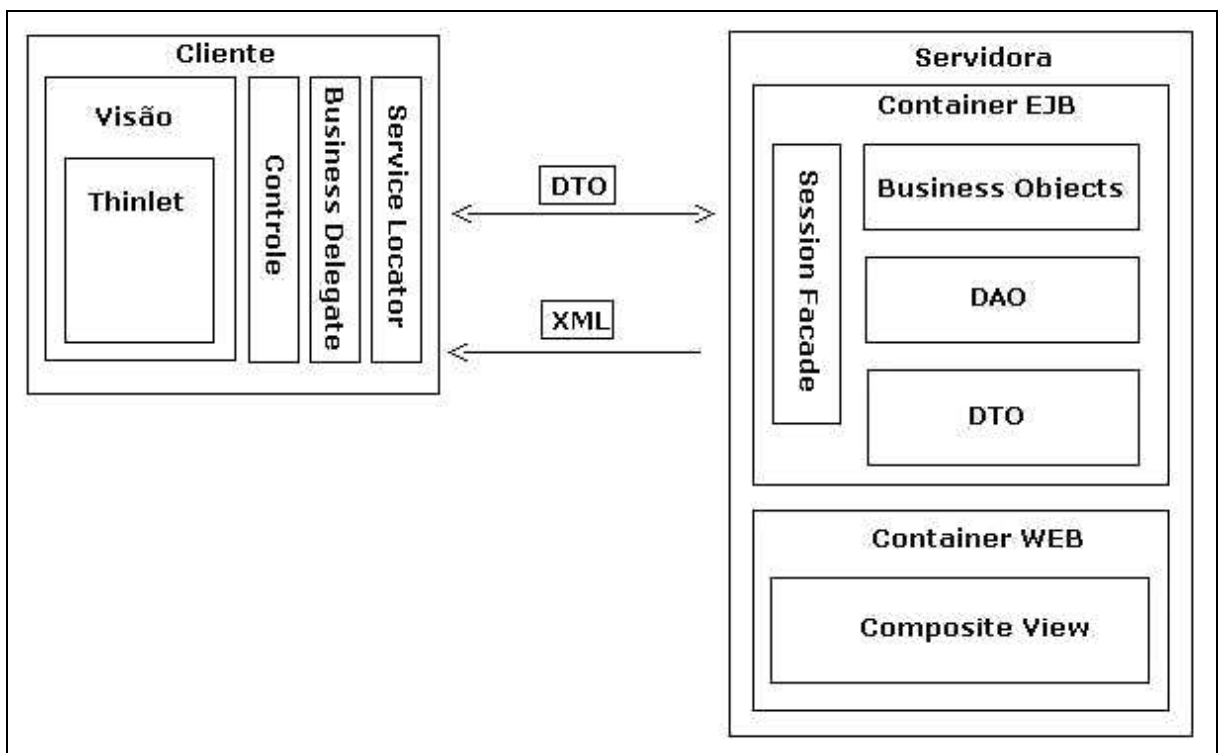


Figura 8 – Padrões utilizados no *framework*

### 3.2.2 *Framework* para a camada servidora

Na camada servidora do *framework* estão as classes que devem ser estendidas pelo

usuário do *framework* para a implementação da camada servidora da aplicação. Essa provê os dados para o cliente do *framework* assim como a regra de negócio do sistema a ser desenvolvido. A camada servidora do *framework* é formada pelas classes: `BaseEntity`, `BaseDAO`, `Provider` e `ProviderBean` conforme mostra o diagrama de classes da figura 9, as quais são:

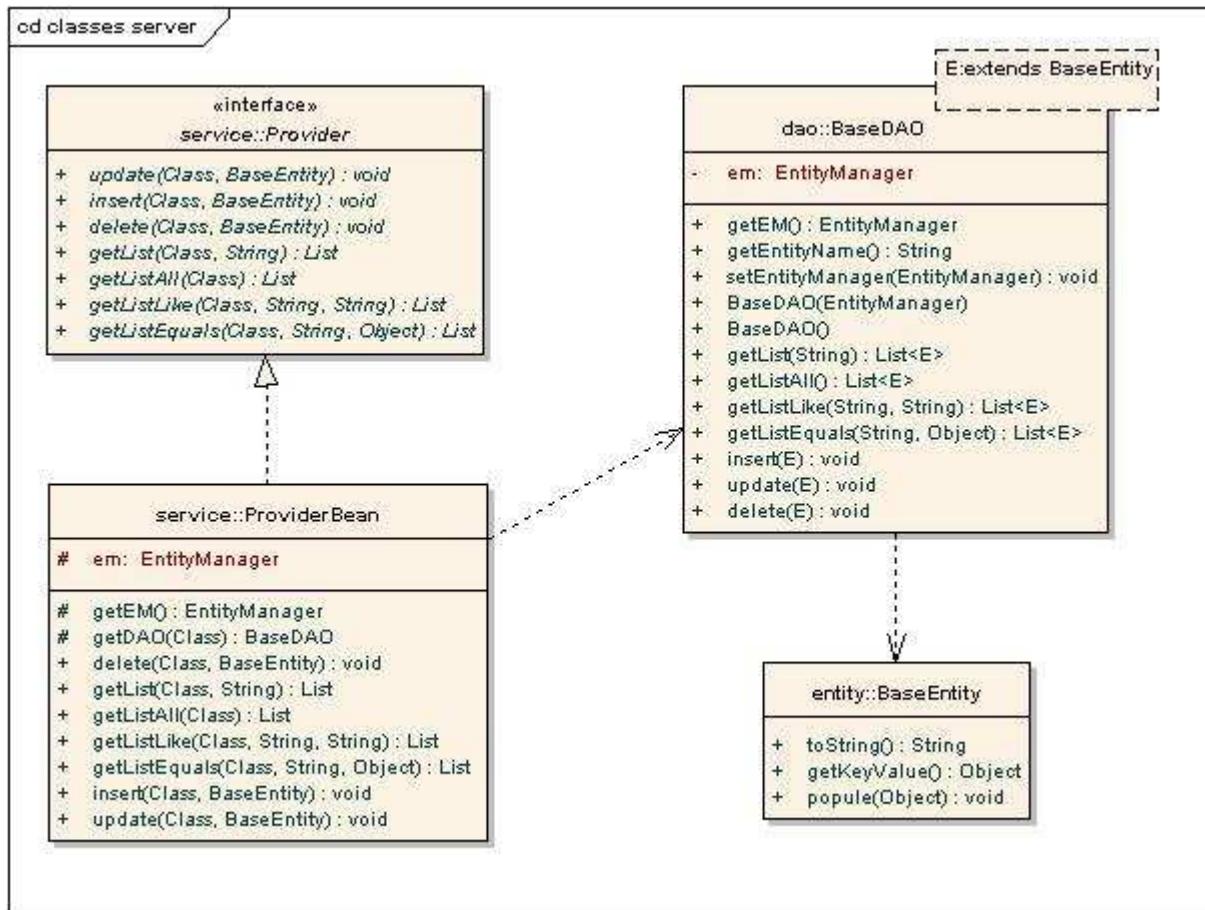


Figura 9 – Diagrama de classes da camada servidora do *framework*

- `BaseEntity`: Essa é uma classe POJO comum que é estendida pelo usuário do *framework* para implementação dos objetos que fazem relação com a base de dados. Essas classes são utilizadas como objetos de transferências de dados, implementando assim o padrão *Transfer Object*. Na especificação EJB3 essas classes são os EJB de entidade;
- `BaseDAO`: Como o nome sugere, essa classe implementa o padrão *Data Access Object*. Ela contém métodos que fazem acesso ao armazenamento persistente dos dados e é estendida pelo usuário do *framework*, para que o mesmo possa implementar novos métodos de acesso aos dados que sejam pertinentes a sua aplicação. Essa classe também possui um tipo genérico de dados que representa o tipo de dados que é utilizado pela classe `BaseDAO`;

- c) `Provider`: É uma interface que possui os métodos do modelo CRUD. Essa interface será utilizada pela camada cliente do *framework* para acessar os métodos remotos da classe `ProviderBean`;
- d) `ProviderBean`: Contém a implementação dos métodos do modelo CRUD. Os padrões utilizados nessa classe são: *Session Facade* e *Business Object*. A mesma serve como fachada para acessar os métodos de negócio do *framework*.

A figura 10 mostra as mensagens trocadas entre as classes da camada servidora do *framework*. Nesse diagrama de seqüência encontra-se uma classe chamada `EntityManager`, que não foi implementada no projeto, mas faz parte da arquitetura EJB3 que é utilizada para gerenciar as entidades de uma base de dados.

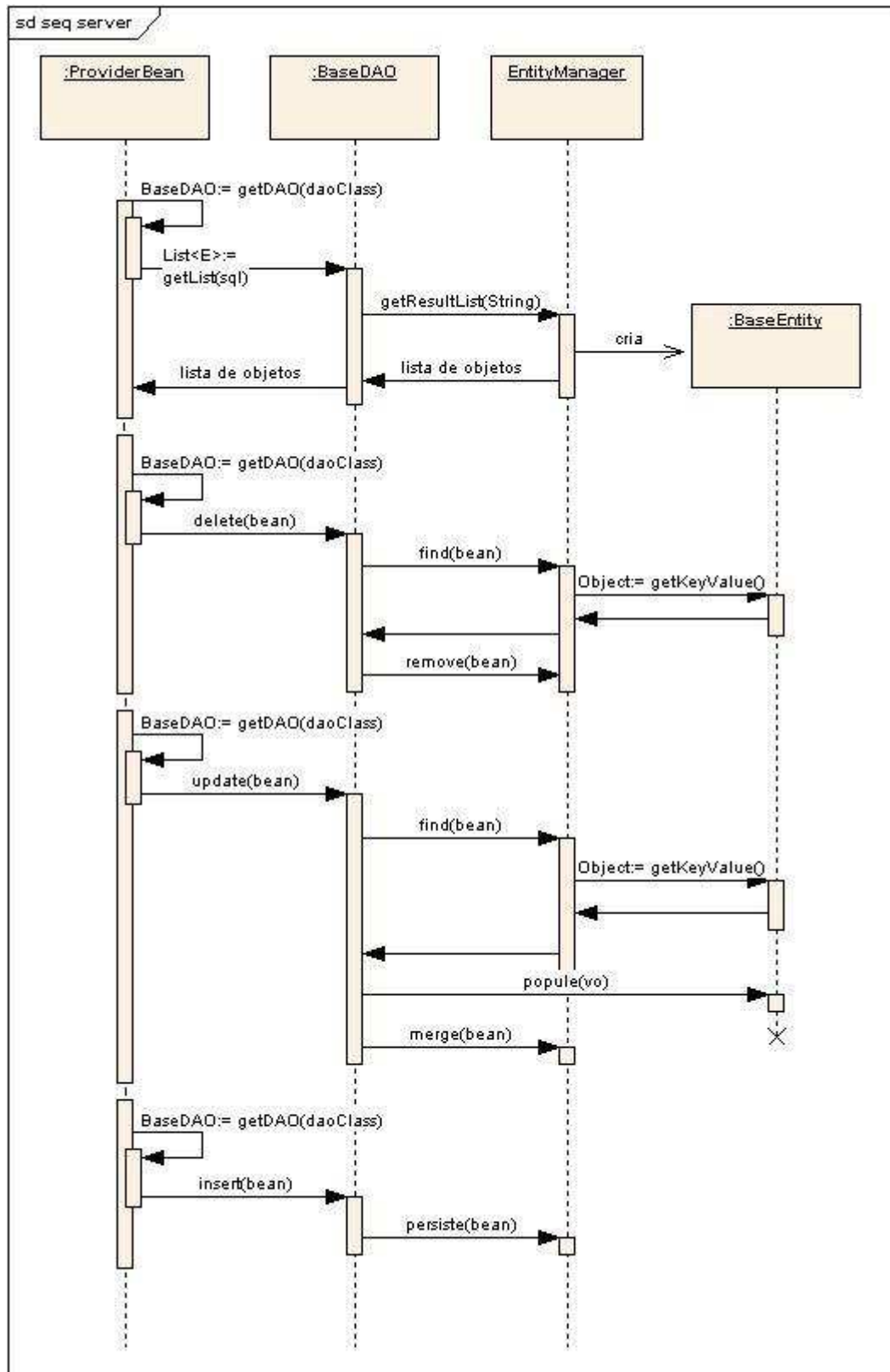


Figura 10 – Diagrama de seqüência da camada servidora do *framework*

### 3.2.3 Fornecimento do XML

O fornecimento do XML da tela para o cliente do *framework* é feito por uma aplicação web, que pode ser desenvolvida pelo usuário do *framework* ou utilizada uma aplicação web que faz parte do *framework* desenvolvido. A parte web do *framework* dividiu-se em:

- a) *FrameworkWeb*: É uma aplicação web com o objetivo de disponibilizar remotamente os arquivos XML de uma pasta local no servidor. Nessa aplicação o usuário cria, edita, exclui e visualiza os arquivos XML, que são usados pela camada cliente do *framework* para a geração das telas Thinlet. A figura 11 mostra um diagrama web da aplicação.

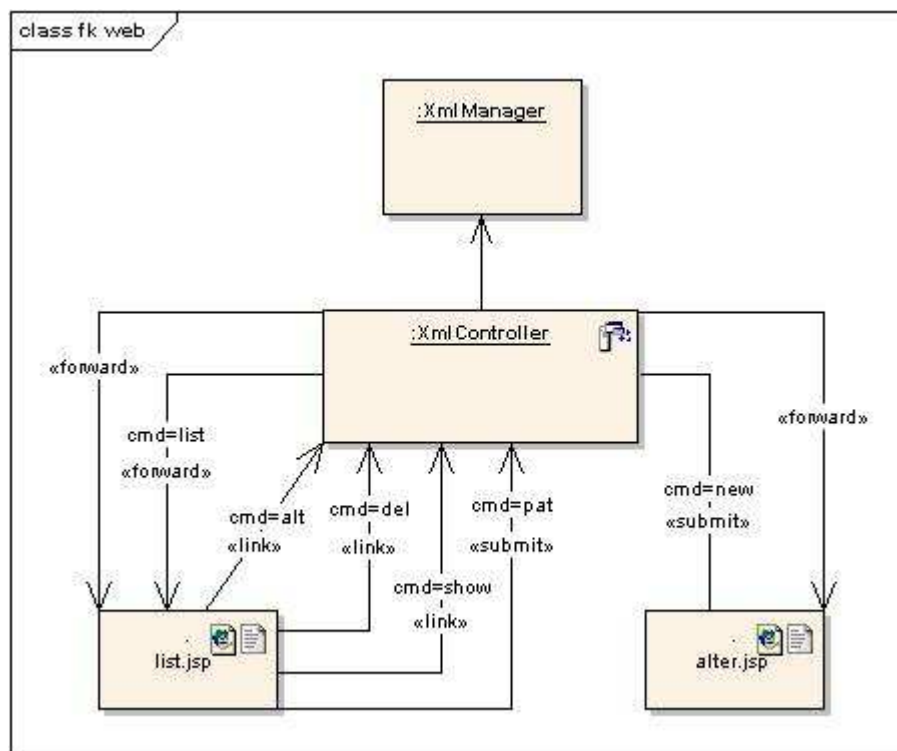


Figura 11 – Diagrama web da aplicação *FrameworkWeb*

- b) *FrameworkTags*: São *Custom Tags* implementadas para auxiliar no desenvolvimento de telas remotas no padrão de DTD do Thinlet. As mesmas são usadas pela aplicação web criada pelo usuário do *framework*, tendo o objetivo de reduzir a duplicação de *tags* XML do Thinlet. Uma *Custom Tag* retorna um XML contendo um ou vários componentes visuais Thinlet. Utilizando as *Custom Tags* em JSP pode ser implementado o padrão *Composite View* de forma mais simples e reutilizável. A figura 12 mostra o diagrama de classes das *Custom Tags* implementadas.

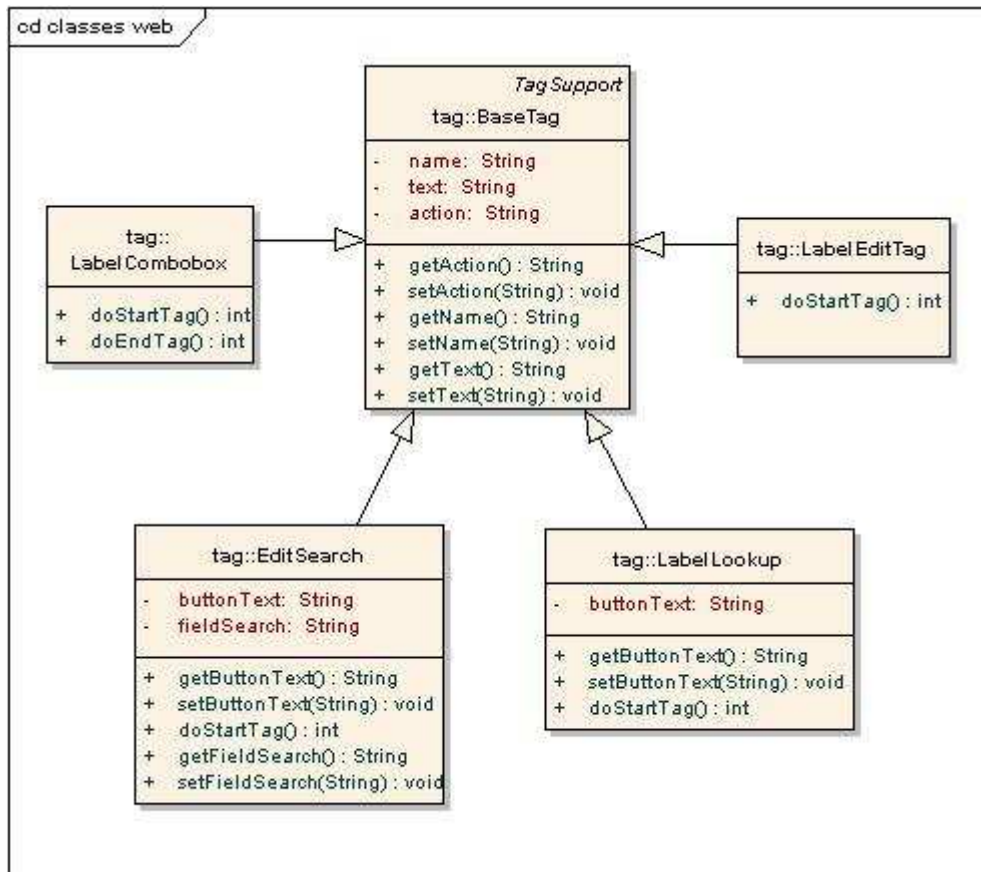


Figura 12 – Diagrama de classes de *Custom Tags* do *framework*

### 3.2.4 *Framework* para a camada cliente

A camada cliente do *framework* deve ser estendida pelo desenvolvedor para a criação das classes controladoras de tela. Nessa camada é implementado o modelo MVC e os padrões *Service Locator* e *Business Delegate*. A camada cliente do *framework* é formada por sete classes principais: `BindThinlet`, `RemoteThinlet`, `RemoteThinletMdi`, `RemoteDialogThinlet`, `BaseController`, `BaseGridController`, `BaseGridSearchController`, `BaseCadController`, `BaseDialogSearchController` e `DataProvider` conforme mostrado no diagrama de classes da figura 13:

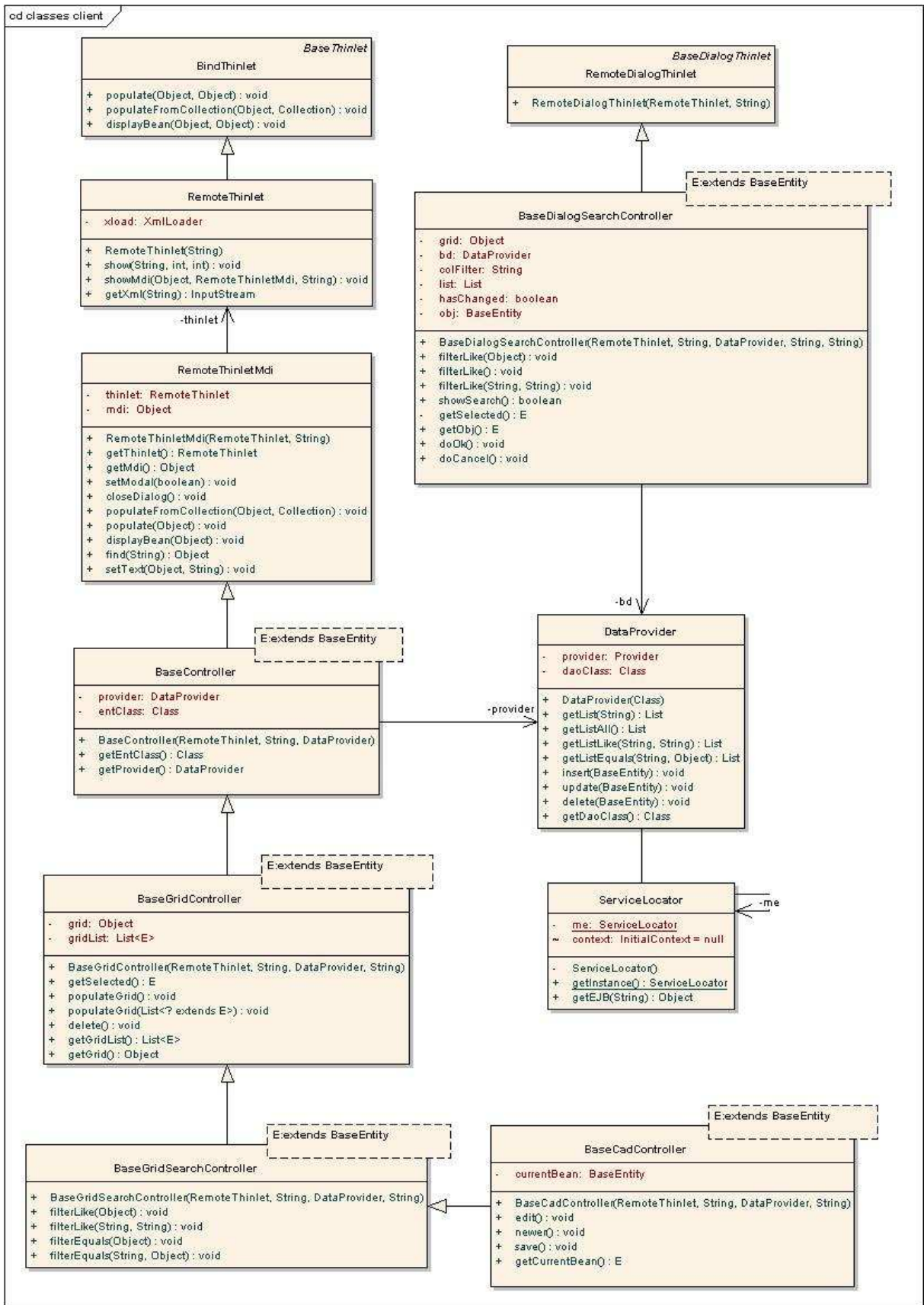


Figura 13 – Diagrama de classes da camada cliente do *framework*

a) BindThinlet: A classe BindThinlet é responsável em fazer a ligação do DTO e

os componentes visuais do Thinlet. A mesma estende a classe `BaseThinlet` do projeto Genesis, que possui facilidades para fazer ligação entre componentes Thinlet e objetos de dados;

- b) `RemoteThinlet`: Essa classe implementa um recurso adicional no Thinlet, que é a criação de formulários Thinlet a partir de XML remotos. Para isso foi utilizado o componente `HttpClient` do projeto Jakarta. A classe estende de `BindThinlet` para que a mesma possua a implementação de ligação dos DTO com o formulário Thinlet;
- c) `RemoteThinletMdi`: Estende a classe `RemoteThinlet`, porém implementa tratamentos para formulário do tipo MDI;
- d) `RemoteDialogThinlet`: Essa classe estende a classe `BaseDialogThinlet` do projeto Genesis que implementa um formulário de diálogo Thinlet. Na mesma foi implementado recursos para criação de diálogos Thinlet a partir de XML remotos;
- e) `BaseController`: Classe base para criar um controlador de tela. A classe será estendida pelo usuário do *framework* para criar um controlador que não possua os métodos padrões do modelo CRUD. Nessa classe foi utilizado um tipo genérico que informa com que tipo de dados a mesma irá trabalhar;
- f) `BaseGridController`: Classe controladora de tela que possua uma grade para visualização de registros. A mesma contém métodos que fazem a ligação de uma lista de objetos com uma grade;
- g) `BaseGridSearchController`: Classe controladora para telas com grade que necessitem fazer busca de registros filtrando os dados mostrados na grade;
- h) `BaseCadController`: Essa classe deve ser estendida pelo usuário do *framework* para criação de um controlador padrão de uma tela no modelo CRUD. Ela possui a implementação dos métodos para criar, ler, atualizar e excluir conforme o modelo CRUD;
- i) `BaseDialogSearchController`: É uma classe que implementa um controlador padrão de diálogo de busca. Ela também possui um tipo genérico que informa qual o tipo de dados que a classe irá trabalhar. É estendida pelo usuário do *framework* para a implementação dos controladores para as telas de busca do sistema. A mesma também pode ser utilizada como controlador para formulários de busca simples sem necessidade de extensão;
- j) `DataProvider`: Implementa o padrão *Business Delegate* e utiliza a classe



`ServiceLocator` para localizar os objetos remotos que estão no servidor conforme o padrão *Service Locator*. A classe é responsável por acessar a regra de negócio do *framework* através de chamadas remotas para um servidor JavaEE.

### 3.3 IMPLEMENTAÇÃO

A seguir são mostradas as ferramentas utilizadas no desenvolvimento do *framework* e do protótipo. Assim como a implementação do *framework*.

#### 3.3.1 Ferramentas utilizadas

O *framework* foi implementado utilizando a ferramenta JBoss IDE para Eclipse. De acordo com Red Hat (2007c) JBoss IDE para Eclipse é uma série de *plugins* do Eclipse integrados para construir aplicações JEMS. JBoss IDE para Eclipse estende o Eclipse e permite aos programadores desenvolver, distribuir, testar e depurar as aplicações baseadas em JEMS sem deixar a IDE. Simplificando o ciclo de vida do desenvolvimento.

Outra ferramenta utilizada para o desenvolvimento foi o ThinG que é um editor gráfico para Thinlet. O ThinG, conforme Möbius (2007), foi desenvolvido em aproximadamente sete dias usando o próprio Thinlet para implementar as telas do editor. O ThinG foi utilizado para criar os XML das telas do protótipo.

Como servidor de aplicação foi utilizado o JBoss por ser uma ferramenta livre, e implementar as especificações EJB3.

#### 3.3.2 Implementação do *framework*

Nas próximas subseções são apresentados as implementações do *framework* que estão divididas em: camada servidor, camada web e camada cliente. Em seguida é demonstrada a utilização do *framework* no protótipo da vídeo locadora.

### 3.3.2.1 Camada servidora

A camada servidora do *framework* foi implementada usando as especificações EJB3. O uso das especificações EJB3 trouxe muitas facilidades para implementação das regras de negócio do modelo CRUD. Ela foi implementada principalmente na classe `BaseDAO` que contém os métodos do modelo. O quadro 9 mostra o código parcial da classe `BaseDAO`. Nessa classe pode-se destacar o uso do objeto `EntityManager` que atua no contexto de persistência dos EJB de entidade na arquitetura EJB3.

```

package piske.server.dao;

import piske.server.entity.BaseEntity;
//...
public class BaseDAO<E extends BaseEntity> {
    private EntityManager em;
    //...
    public List<E> getList(String sql) {
        return (List<E>) em.createQuery(sql).getResultList();
    }

    public List<E> getListAll() {
        return (List<E>) getList("from " + getEntityName());
    }

    public List<E> getListLike(String col, String exp) {
        return (List<E>) getList("from " + getEntityName() +
            " where " + col + " like '" + exp + "%'");
    }

    public List<E> getListEquals(String col, Object exp) {
        Query q = em.createQuery("from " + getEntityName() +
            " where " + col + " = :exp");
        q.setParameter("exp", exp);
        return (List<E>) q.getResultList();
    }

    public void insert(E bean) {
        em.persist(bean);
    }

    public void update(E bean) {
        BaseEntity ent = (BaseEntity) em.find(bean.getClass(), bean
            .getKeyValue());
        ent.popule(bean);
        em.merge(ent);
    }

    public void delete(E bean) {
        Object ent = em.find(bean.getClass(), bean.getKeyValue());
        em.remove(ent);
    }
}

```

Quadro 9 – Código da classe `BaseDAO`

A classe `ProviderBean` tem uma grande importância no *framework*. Ela deve ser estendida pelo usuário do *framework* para criar o EJB de sessão responsável por expor os serviços do *framework*. Nela são instanciados os objetos DAO da aplicação de acordo com a requisição da camada cliente do *framework*. Nessa parte está algo interessante do *framework* que é utilizar os objetos DAO implementados pelo usuário do *framework*. Isso possibilita uma mobilidade para o desenvolvedor em poder reescrever métodos da classe `BaseDAO` para tratamentos específicos. No quadro 10 é mostrado o código em parcial da classe `ProviderBean`.

```

package piske.server.service;

import piske.server.service.Provider;
//...
public class ProviderBean implements Provider {

    @PersistenceContext
    protected EntityManager em;

    protected BaseDAO getDAO(Class daoClass) throws
        InstantiationException, IllegalAccessException
    {
        BaseDAO d = (BaseDAO) daoClass.newInstance();
        d.setEntityManager(em);
        return d;
    }

    public void delete(Class daoClass, BaseEntity bean)
        throws InstantiationException, IllegalAccessException {
        getDAO(daoClass).delete(bean);
    }

    public List getList(Class daoClass, String sql)
        throws InstantiationException, IllegalAccessException {
        return getDAO(daoClass).getList(sql);
    }

    public List getListAll(Class daoClass) throws InstantiationException,
        IllegalAccessException {
        return getDAO(daoClass).getListAll();
    }

    public List getListLike(Class daoClass, String col, String exp)
        throws InstantiationException, IllegalAccessException {
        return getDAO(daoClass).getListLike(col, exp);
    }

    public List getListEquals(Class daoClass, String col, Object exp)
        throws InstantiationException, IllegalAccessException {
        return getDAO(daoClass).getListEquals(col, exp);
    }

    public void insert(Class daoClass, BaseEntity bean)
        throws InstantiationException, IllegalAccessException {
        getDAO(daoClass).insert(bean);
    }

    public void update(Class daoClass, BaseEntity bean)
        throws InstantiationException, IllegalAccessException {
        getDAO(daoClass).update(bean);
    }
}

```

Quadro 10 – Código da classe ProviderBean

### 3.3.2.2 Camada web

A camada web do *framework* está dividida em dois projetos *FrameworkWeb* e *FrameworkTags*. O projeto *FrameworkWeb* foi desenvolvido utilizando JSP e *servlet*. No *servlet* `XmlController` está implementado o tratamento dos comandos submetidos pelas JSPs. Nesse *servlet* é usado o objeto `XmlManager` que implementa rotinas para manipular os arquivos XML.

No projeto *FrameworkTags* estão implementadas as *Custom Tags* para auxiliar o desenvolvimento de formulário dinâmicos Thinlet. O quadro 11 mostra a classe `LabelEdit` que implementa uma *Custom Tag*, tendo como retorno o XML de dois componentes Thinlet: `label` e `textfield`.

```

package piske.server.web.tag;

import java.io.IOException;

public class LabelEditTag extends BaseTag {

    public int doStartTag() {
        String res = "<label alignment=\"right\" text=\"" + getText() +
            "\"/>";
        res += "<textfield weightx=\"1\" name=\"" + getName() + "\"/>";
        try {
            pageContext.getOut().print(res);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return SKIP_BODY;
    }
}

```

Quadro 11 – Código da classe `LabelEdit`

### 3.3.2.3 Camada cliente

A camada cliente é o que possui o maior número de classes. Os principais pacotes implementados nessa camada são `piske.server.ui` e `piske.server.controller`. O pacote `piske.server.ui` contém as classes que geram a interface com o usuário a partir de um XML remoto. No quadro 12 é mostrado a classe `RemoteThinlet` que em seu construtor recebe um parâmetro de um XML remoto, o qual será carregado pelo objeto `XmlLoader`. Nesse objeto é feita a conexão com a aplicação web que provê as telas do sistema. Essa conexão é feita através o componente `HttpClient` do projeto Jakarta.

```

package piske.client.ui;

import piske.client.httpxml.XmlLoader;
//...

public class RemoteThinlet extends BindThinlet {
    private XmlLoader xload;

    public RemoteThinlet(String xmlUrl) {
        xload = new XmlLoader();
        try {
            add(parse(xload.getXml(
                ServerParams.getInstance().getXmlPath()+ "/" +
                xmlUrl)));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void show(String titulo, int largura, int altura) {
        new FrameLauncher(titulo, this, largura, altura);
    }

    public void showMdi(Object desktop, RemoteThinletMdi
        dialogController, String title) {
        setText(dialogController.getMdi(), title);
        this.add(desktop, dialogController.getMdi(), 0);
    }

    public InputStream getXml(String xmlUrl) {
        return xload.getXml(ServerParams.getInstance().getXmlPath() +
            "/" + xmlUrl);
    }
}

```

Quadro 12 – Código da classe RemoteThinlet

Outra classe de grande importância no pacote `piske.client.ui` é a `BindThinlet`, responsável por fazer a ligação do DTO com a tela `Thinlet`. Nessa classe é utilizada a implementação de *binding* de tela do projeto Genesis e o componente `BeanUtils` do projeto Jakarta conforme mostrado no quadro 13.

```

package piske.client.ui;

import net.java.dev.genesis.ui.thinlet.BaseThinlet;
import org.apache.commons.beanutils.BeanUtils;
//...

public class BindThinlet extends BaseThinlet {

    public void populate(Object bean, Object root)
        throws IllegalAccessException, InvocationTargetException,
        NoSuchMethodException {
        HashMap properties = new HashMap();

        // adiciona no hashmap os atributos que não são do tipo
        BaseEntity...
        for (Field f : bean.getClass().getDeclaredFields()) {
            Object o = PropertyUtils.getProperty(bean, f.getName());
            if (!(o instanceof BaseEntity)) {
                properties.put(f.getName(),
                    PropertyUtils.getProperty(bean,
                        f.getName()));
            }
        }

        // popula o hashmap com os atributos do formulario Thinlet
        super.populate(bean, root, properties, false);

        // popula o bean com o hashmap
        BeanUtils.populate(bean, properties);
    }

    public void populateFromCollection(Object component, Collection c)
        throws IllegalAccessException, InvocationTargetException,
        NoSuchMethodException {
        super.populateFromCollection(component, c);
        repaint();
    };

    public void displayBean(Object bean, Object root)
        throws IllegalAccessException, InvocationTargetException,
        NoSuchMethodException {
        super.displayBean(bean, root);
        repaint();
    }
}

```

Quadro 13 – Código da classe BindThinlet

No pacote `piske.server.controller` estão as classes controladoras de tela. Nelas está implementados os tratamentos das interações do usuário com a tela Thinlet. Cada controlador foi implementado de maneira a suprir uma tela com características específicas. O quadro 14 mostra a classe `BaseGridController` que possui implementações para tratar uma tela que possua uma grade de dados. As classes controladoras são responsáveis por acessarem o modelo de dados que é feito através do objeto `DataProvider`.

```

package pisco.client.controller;

import pisco.client.dp.DataProvider;
import pisco.client.ui.RemoteThinlet;
//...
public class BaseGridController<E extends BaseEntity> extends
BaseController<E> {
    public BaseGridController(RemoteThinlet thin, String xmlUrl,
        DataProvider dp, String gridName) {
        super(thin, xmlUrl, dp);
        grid = find(gridName);
    }
    public E getSelected() {
        int index = getThinlet().getSelectedIndex(grid);
        if (index > -1)
            return (E) gridList.get(index);
        else
            return null;
    }
    public void populateGrid() throws InstantiationException,
        IllegalAccessException {
        gridList = getProvider().getListAll();
        try {
            populateFromCollection(grid, gridList);
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
    }
    public void populateGrid(List<E> list) {
        if (gridList == null)
            gridList = new ArrayList<E>();
        gridList.clear();
        gridList.addAll(list);
        try {
            if (list != null)
                populateFromCollection(grid, gridList);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public void delete() throws ScreenNotFoundException,
        InstantiationException, IllegalAccessException,
        InvocationTargetException, NoSuchMethodException {
        Object obj = getSelected();
        if (obj != null) {
            Option option = OptionDialog.displayYesNo(getThinlet(),
                "Confirmação", "Deseja excluir o registro?");
            if (option == Option.YES) {
                getProvider().delete(getSelected());
                populateGrid();
            }
        } else
            MessageDialog.show(getThinlet(), "Informação",
                "Não há item selecionado para excluir!");
    }
//...
}

```

Quadro 14 – Código da classe BaseGridController



### 3.4 VALIDAÇÃO DO FRAMEWORK

Em seguida são apresentadas as especificações e implementação do protótipo da vídeo locadora que estão divididas em: introdução ao protótipo, camada servidora do protótipo, camada web do protótipo, camada cliente do protótipo, utilizando o framework, implementar objetos relacionais, implementar classes DAO, implementar regras de negócio, criar XML das telas do sistema e implementar classes controladoras.

#### 3.4.1 Introdução ao protótipo

O protótipo é um sistema de locação de vídeos simples. Foram desenvolvidos alguns cadastros básicos para a implementação de locação e devolução de vídeos. O diagrama de casos de uso da figura 14 mostra os casos de uso implementados no protótipo.

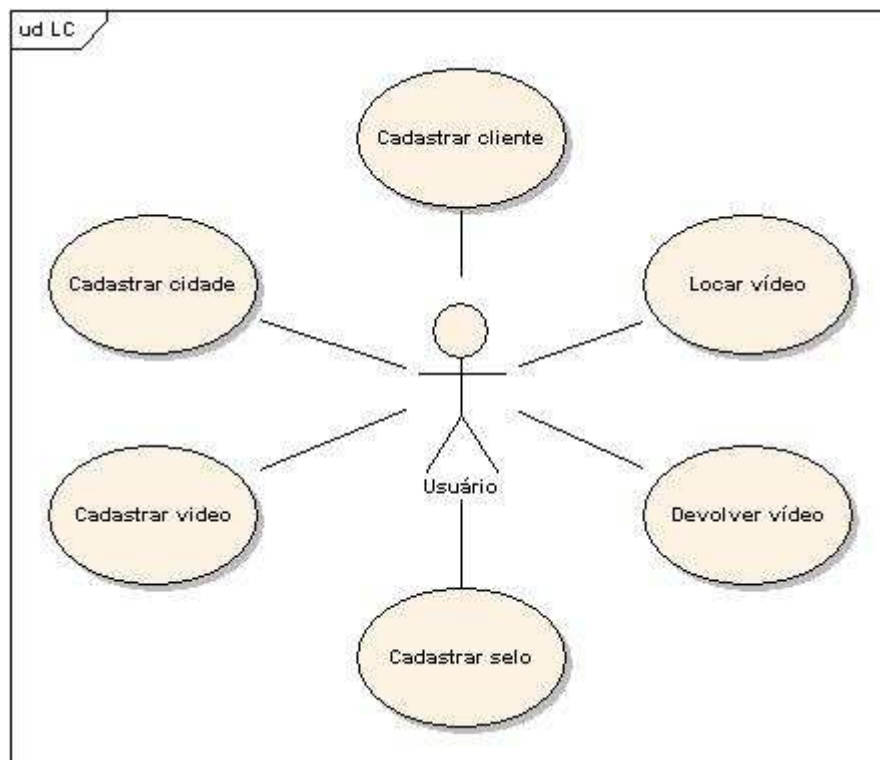


Figura 14 – Diagrama de casos de uso do protótipo

### 3.4.2 Camada servidora do protótipo

Na camada servidora do protótipo foram implementados os EJB de entidade que estendem a classe `BaseEntity`. As classes DAO também estão nessa camada, assim como a regra de negócio do protótipo que está implementado na classe `LocadoraProvider`. O diagrama de classes da figura 15 mostra as classes que compõem a camada servidora do protótipo.

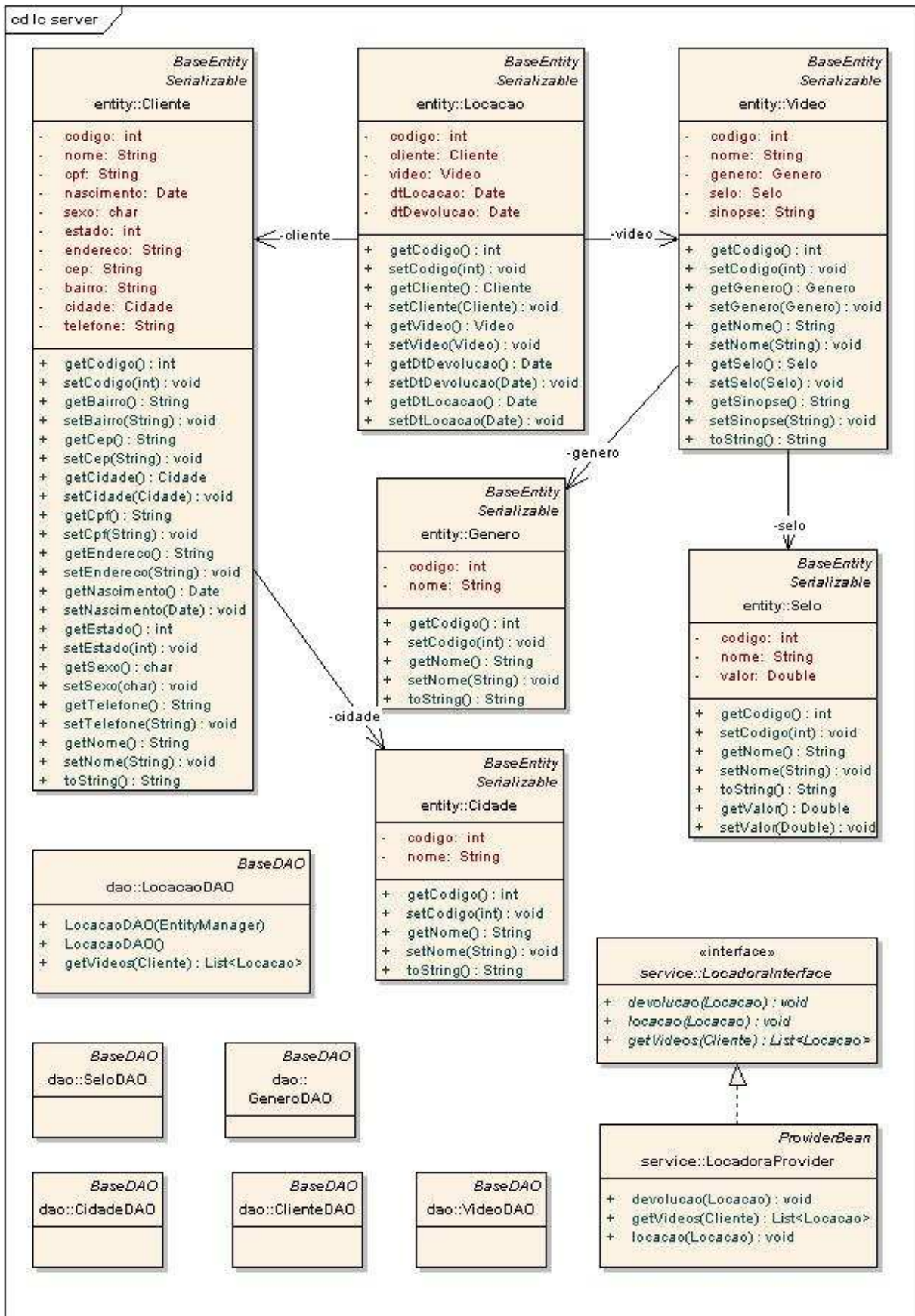


Figura 15 – Diagrama de classes da camada servidora do protótipo

### 3.4.3 Camada web do protótipo

Na camada web do protótipo estão as JSP que retornam um XML de acordo com a DTD do Thinlet. No protótipo foi escolhido desenvolver uma aplicação Web que utilizasse as *Custom Tags* do *framework*. Nessa camada foi usado o padrão *Composite View*, de forma que os códigos que se repetiam nas JSP foram transformados em arquivos de inclusão JSP.

### 3.4.4 Camada cliente do protótipo

Na camada cliente do protótipo estão as classes controladoras de tela. O diagrama de classes da figura 16 mostra as classes controladoras implementadas para o protótipo. Para algumas telas foram usadas apenas as classes controladoras do *framework*, pois essas já provêm a estrutura necessária para a tela desenvolvida.

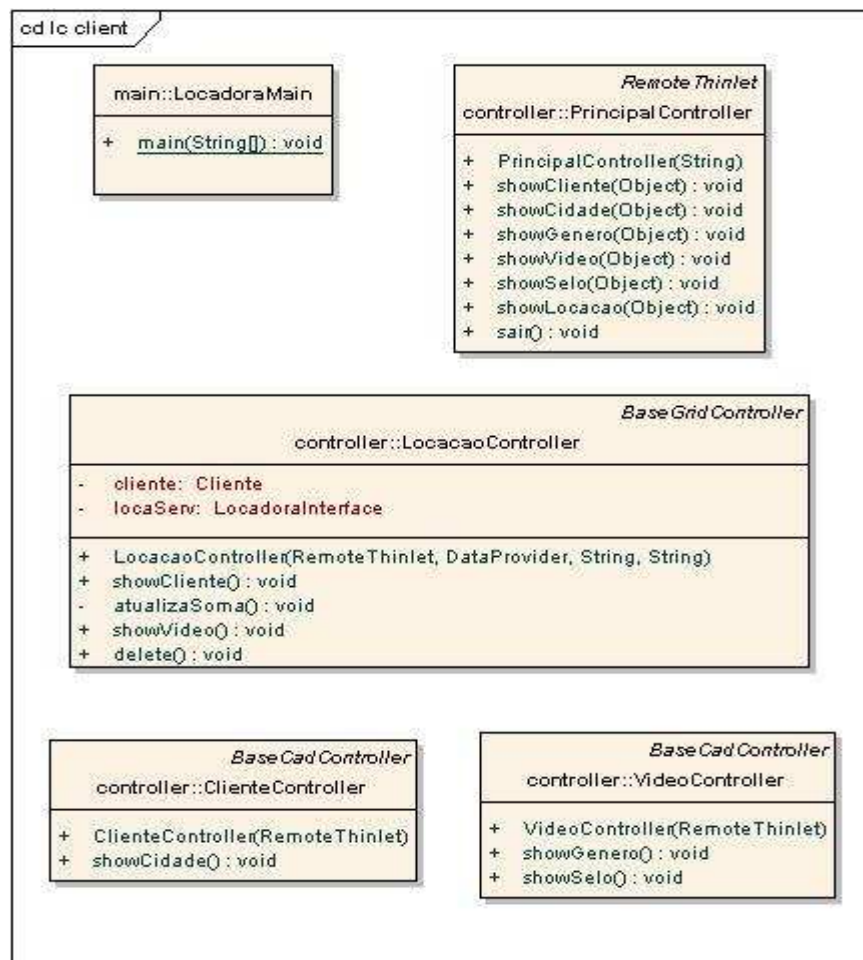


Figura 16 – Diagrama de classes da camada cliente do protótipo

### 3.4.5 Utilizando o *framework*

Para utilização do *framework* é necessário passar por algumas etapas para criar a estrutura necessária para sua execução. Foram definidas cinco etapas de implementação: implementar objetos relacionais, implementar classes DAO, implementar regras de negócio, criar XML das telas, implementar classes controladoras. A seguir são detalhadas essas etapas e também são apresentados exemplos do protótipo de um sistema de vídeo locadora utilizando o *framework* desenvolvido.

#### 3.4.5.1 Implementar objetos relacionais

Os objetos relacionais são implementados como EJB de entidade conforme a especificação EJB3. Esses estendem a classe `BaseEntity` que se encontra no pacote `pische-tcc-server.jar`. No protótipo foi criado um projeto chamado `LocadoraServer` e as classes que fazem relação com o banco de dados encontram-se no pacote `locadora.server.entity`.

O código do objeto relacional é mostrado no quadro 15, que se refere a classe `Cliente` usada no cadastro de Clientes do protótipo. A classe `Cliente` é uma classe POJO simples com anotações de um EJB de entidade. O código no quadro 15 apresenta alguns métodos e atributos. Os demais seguem o mesmo padrão. As outras classes `Cidade`, `Genero`, `Locacao`, `Selo` e `Video` seguem o mesmo padrão de implementação.

```

package locadora.server.entity;

//...
import piske.server.entity.BaseEntity;

@Entity
@Table(name = "cliente")
public class Cliente extends BaseEntity implements Serializable {

    private int codigo;
    private String nome;
    private String cpf;
    private Date nascimento;
    private char sexo;
    private int estado;
    private String endereco;
    private String cep;
    private String bairro;
    private Cidade cidade;
    private String telefone;

    @Id
    @GeneratedValue
    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    @ManyToOne(optional = false)
    @JoinColumn(name="CIDADE", nullable=false)
    public Cidade getCidade() {
        return cidade;
    }

    public void setCidade(Cidade cidade) {
        this.cidade = cidade;
    }

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    @Column(nullable = false)
    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    // ...
}

```

Quadro 15 – Código da classe Cliente do protótipo

### 3.4.5.2 Implementar classes DAO

As classes DAO devem estender a classe `BaseDAO` informando qual o tipo genérico de dados que a mesma deve utilizar. Os principais métodos do modelo CRUD já estão implementados na classe `BaseDAO`, de maneira que o usuário deve apenas implementar rotinas de acesso aos dados que sejam apropriados à sua aplicação. O quadro 16 mostra a classe `LocacaoDAO` que implementa o método `getVideos()` usado na regra de negócio da locadora de vídeo. As classes `CidadeDAO`, `ClienteDAO`, `GeneroDAO`, `SeloDAO` e `VideoDAO` estendem apenas a classe `BaseDAO` e não possuem nenhuma implementação pertinente à aplicação da vídeo locadora.

```

package locadora.server.dao;

import locadora.server.entity.Cliente;
import locadora.server.entity.Locacao;
import piske.server.dao.BaseDAO;

//...
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();
    }
}

```

Quadro 16 – Código da classe `LocacaoDAO` do protótipo

### 3.4.5.3 Implementar regras de negócio

As regras de negócio do sistema de vídeo locadora estão implementadas em um EJB de sessão. O EJB de sessão é utilizado para prover a regras de negócio do *framework* e as do sistema desenvolvido. O quadro 17 mostra o código da classe `LocadoraProvider` que estende a classe `ProviderBean` e implementa duas interfaces: `Provider` e `LocadoraInterface`. A interface `Provider` contém os métodos de negócio do *framework* e a interface

LocadoraInterface contém os métodos de negócio da locadora.

```

package locadora.server.service;

import javax.ejb.Stateless;
import locadora.server.dao.LocacaoDAO;
import locadora.server.entity.Cliente;
import locadora.server.entity.Locacao;
import locadora.server.service.LocadoraProvider;
import piske.server.service.Provider;
import piske.server.service.ProviderBean;

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);
    }

    public void locacao(Locacao loc) {
        loc.setDtLocacao(new Date());
        LocacaoDAO dao = new LocacaoDAO(getEM());
        dao.insert(loc);
    }
}

```

Quadro 17 – Código da classe LocadoraProvider do protótipo

#### 3.4.5.4 Criar XML das telas do sistema

No protótipo foi escolhido desenvolver uma aplicação web utilizando as *Custom Tags* do *framework* para prover os XML das telas. Foi implementado um projeto chamado LocadoraWeb que contém as JSP da aplicação. No quadro 18 é mostrada a JSP `cad_cliente.jsp` que implementa a tela de cadastro de clientes do protótipo da vídeo locadora. Nela são usadas as *Custom Tags* do *framework* e recursos da JSP como a *tag include*.



```

<%@ page language="java" contentType="text/xml; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://piske.server.org/tags" prefix="fk"%>
<dialog columns="1" gap="4" height="400" resizable="true" scrollable="true"
width="500">
  <jsp:include page="inc/menu.jsp"></jsp:include>
  <panel columns="4" gap="4" left="4" right="4">
    <fk:labeledit name="nome" text="Nome:"></fk:labeledit>
    <fk:labeledit name="cpf" text="CPF:"></fk:labeledit>
    <fk:labeledit name="nascimento"
      text="Nascimento:"></fk:labeledit>
    <fk:labelcombobox name="sexo" text="Sexo:">
    <choice name="M" text="Masculino"/>
    <choice name="F" text="Feminino"/>
    </fk:labelcombobox>
    <fk:labelcombobox name="estado" text="Estado:">
    <choice name="1" text="Solteiro"/>
    <choice name="2" text="Casado"/>
    <choice name="3" text="Divorciado"/>
    <choice name="4" text="Junto"/>
    </fk:labelcombobox>
    <fk:labeledit name="endereco" text="Endereço:"></fk:labeledit>
    <fk:labeledit name="cep" text="CEP:"></fk:labeledit>
    <fk:labeledit name="bairro" text="Bairro:"></fk:labeledit>
    <fk:labellookup name="cidade" text="Cidade:" buttonText="..."
      action="showCidade"></fk:labellookup>
    <fk:labeledit name="telefone" text="Telefone:"></fk:labeledit>
  </panel>
  <separator/>
  <panel columns="2">
    <fk:editsearch name="filter" fieldSearch="nome"
      buttonText="Buscar" text="Filtro:"
      action="filterLike(filter)"></fk:editsearch>
    <jsp:include page="inc/navegador.jsp"></jsp:include>
  </panel>
  <table name="grid" weightx="1" weighty="1">
    <header>
      <column name="nome" text="Nome" width="93"/>
      <column name="cpf" text="CPF" width="103"/>
      <column name="cidade" text="Cidade"/>
    </header>
  </table>
  <jsp:include page="inc/fechar.jsp"></jsp:include>
</dialog>

```

Quadro 18 – Código do JSP cad\_cliente.jsp do protótipo

A JSP cad\_cliente.jsp retorna um XML no padrão de DTD do Thinlet. O XML gerado é mostrado no quadro 19. Foram tiradas algumas partes do código para que o mesmo pudesse ser mostrado em uma única página.

```

<dialog columns="1" gap="4" height="400" resizable="true"
  scrollable="true" width="500">
  <panel columns="4" gap="4" left="4" right="4">
    <label alignment="right" text="Nome:" />
    <textfield weightx="1" name="nome" />
    <label alignment="right" text="CPF:" />
    <textfield weightx="1" name="cpf" />
    <label alignment="right" text="Nascimento:" />
    <textfield weightx="1" name="nascimento" />
    <label alignment="right" text="Sexo:" />
    <combobox editable="false" weightx="1" name="sexo">
      <choice name="M" text="Masculino" />
      <choice name="F" text="Feminino" />
    </combobox>
    <label alignment="right" text="Estado:" />
    <combobox editable="false" weightx="1" name="estado">
      <choice name="1" text="Solteiro" />
      <choice name="2" text="Casado" />
      <choice name="3" text="Divorciado" />
      <choice name="4" text="Junto" />
    </combobox>
    <label alignment="right" text="Endereço:" />
    <textfield weightx="1" name="endereco" />
    <label alignment="right" text="CEP:" />
    <textfield weightx="1" name="cep" />
    <label alignment="right" text="Bairro:" />
    <textfield weightx="1" name="bairro" />
    <label alignment="right" text="Cidade:" />
    <panel weightx="1">
      <textfield weightx="1" name="cidade" />
      <button action="showCidade" text="..." />
    </panel>
    <label alignment="right" text="Telefone:" />
    <textfield weightx="1" name="telefone" />
  </panel>
  <separator />
  <panel columns="2">
    <panel halign="right" right="4" weightx="1">
      <button action="newer" alignment="right" halign="right"
        icon="img/novo.gif" tooltip="Novo" />
      <button action="save" alignment="right" halign="right"
        icon="img/salvar.gif" tooltip="Salvar" />
      <button action="delete" alignment="right" halign="right"
        icon="img/excluir.gif" tooltip="Excluir" />
      <button action="edit" alignment="right" halign="right"
        icon="img/editar.gif" tooltip="Editar" />
    </panel>
  </panel>
  <table name="grid" weightx="1" weighty="1">
    <header>
      <column name="nome" text="Nome" width="93" />
      <column name="cpf" text="CPF" width="103" />
      <column name="cidade" text="Cidade" />
    </header>
  </table>
</dialog>

```

Quadro 19 – Código do XML do cadastro de clientes do protótipo





A tela Thinlet gerada pelo código XML do quadro 18 é mostrada na figura 17.

Protótipo Locadora  
Cadastros Serviços Sair

Cadastro de Clientes

Editar

Nome: Leandro Salvatti Pische CPF: 044.222.058-01  
Nascimento: 1983-06-12 Sexo: Masculino  
Estado: Solteiro Endereço: Joinville  
CEP: 890031-00 Bairro: Vila Nova  
Cidade: Blumenau Telefone: 99559835

Filtro:  Buscar    

Nome	CPF	Cidade
Leandro Salvatti Pische	044.222.058-01	Blumenau
Adilson	898.265.056-03	Gaspar

Fechar

Figura 17 – Tela do cadastro de clientes do protótipo

A seguir são apresentadas mais duas telas do protótipo, a figura 18 mostra o cadastro de vídeos e a figura 19 o cadastro de cidades.

**Protótipo Locadora**  
Cadastros Serviços Sair

Cadastro de Vídeos

Editar

Nome:  Gênero:

Selo:

Sinopse:

Após anos procurando seu príncipe encantado, Charlotte Cantilini (Jennifer Lopez) se apaixona por Kevin Fields (Michael Vartan). O problema é a mãe dele, Viola (Jane Fonda), que foi recentemente demitida do cargo de âncora de um jornal de rede nacional. Após perder o emprego, Viola teme perder também o filho e para evitar isto decide atrapalhar ao máximo os planos.

Filtro:      

Nome	Gênero	Selo
A Sogra	Comédia	Lançamento
Orgulho e Preconceito	Romance	Lançamento
Cassino Royale	Aventura	Lançamento
Procurando Nemo	Infantil	Infantil
Efeito Borboleta	Ficção	Acervo

Figura 18 – Tela do cadastro de vídeos

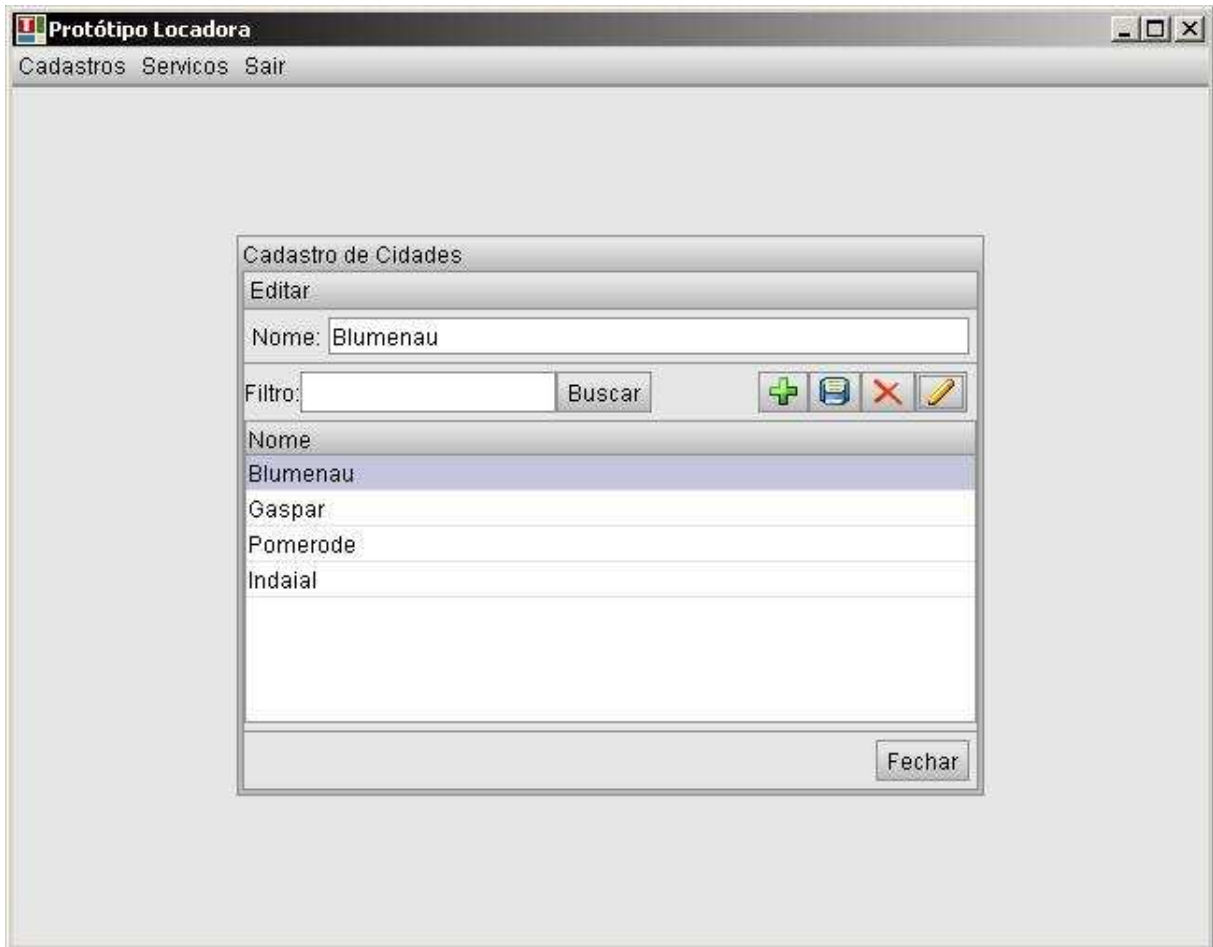


Figura 19 – Tela do cadastro de cidades

#### 3.4.5.5 Implementar classes controladoras

As classes controladoras de tela são implementadas na camada cliente da aplicação. No protótipo foi criado um projeto chamado `LocadoraCliente` que contém essas classes. Para que aplicação cliente faça uso da camada servidora, deve-se criar um arquivo chamado `server.properties` contendo duas propriedades: `server.xmlpath` e `server.providerbean`, conforme o quadro 20. A propriedade `server.xmlpath` indica para a camada cliente do *framework* o caminho onde estão os XML das telas. Na propriedade `server.providerbean` deve ser informado o nome do EJB de sessão que implementa as regras de negócio do *framework*.

```
server.xmlpath=http://localhost:8080/LocadoraWeb/
server.providerbean=LocadoraProvider/remote
```

Quadro 20 – Arquivo `server.properties`

Para implementar uma classe controladora o usuário deve estender uma das classes do pacote `piske.server.controller` que dá suporte à tela implementada. No quadro 21 é mostrada a classe controladora `ClienteController` desenvolvida para o cadastro de clientes do protótipo. Nessa classe é implementado o método `showCidade()` que está definido em uma *tag button* do XML do quadro 19. Todos os outros métodos do modelo CRUD chamados pela tela estão implementados na classe `BaseCadController`. Na classe `ClienteController` é passado o tipo genérico de dados que a mesma deve utilizar. No construtor da classe são passados parâmetros dos tipos:

- a) `RemoteThinlet`: Objeto `Thinlet` pai dos formulários MDI;
- b) `String`: JSP ou arquivo remoto que retorne um XML da tela que se deseja visualizar;
- c) `DataProvider`: Objeto que provê a lógica do *framework* para a aplicação cliente. No construtor desse objeto deve-se passar a classe DAO que o mesmo deve utilizar;
- d) `String`: Nome do componente grade no XML da tela que será usado pela classe controladora.

```

package locadora.client.controller;

import locadora.server.dao.CidadeDAO;
import locadora.server.dao.ClienteDAO;
import piske.client.controller.BaseCadController;
import piske.client.controller.BaseDialogSearchController;

//...
public class ClienteController extends BaseCadController<Cliente> {

    public ClienteController(RemoteThinlet thin) throws Exception {
        super(thin, "cad_cliente.jsp", new
            DataProvider(CidadeDAO.class), "grid");
    }

    public void showCidade() throws Exception {
        BaseDialogSearchController<Cidade> cli = new
        BaseDialogSearchController<Cidade>(
            getThinlet(), "lookup.jsp", new
            DataProvider(CidadeDAO.class), "grid", "nome");
        if (cli.showSearch()) {
            getCurrentBean().setCidade(cli.getObj());
            setText(find("cidade"), cli.getObj().getNome());
        }
    }
}

```

Quadro 21 – Código da classe `ClienteController` do protótipo

### 3.5 RESULTADOS E DISCUSSÃO

Os resultados obtidos com o projeto desenvolvido estão relacionados com a economia de tempo e facilidade no desenvolvimento de um sistema de informação. O protótipo implementado é um sistema simples de locação de vídeo. Porém o mesmo foi desenvolvido em uma arquitetura de três camadas utilizando padrões de projetos. A utilização de arquitetura em três camadas e padrões de projetos tende a aumentar a complexidade de uma aplicação. Com a utilização do *framework* pôde-se eliminar a complexidade no desenvolvimento e diminuir o tempo de implementação do sistema.

Como resultado tem-se uma aplicação desenvolvida de forma que o usuário do *framework* teve maior preocupação em implementar a lógica de negócio de sua aplicação. No protótipo desenvolvido na camada do servidor foram criados três pacotes:

- a) `locadora.server.entity`: Foram definidos as classes que fazem relação com o banco de dados. As mesmas são usadas pelo *framework* como objetos DTO;
- b) `locadora.server.dao`: As classes DAO na maioria foram simplesmente estendidas pelo usuário, com exceção de uma que teve métodos a mais implementados para serem usados na implementação de regra de negócio da locadora de vídeo. As demais classes DAO foram apenas estendidas da classe `BaseDAO` que já implementa os métodos do modelo CRUD de forma genérica;
- c) `locadora.server.service`: Nesse pacote foi implementado apenas a lógica de negócio da locadora de vídeo.

Na camada cliente da aplicação o tempo de desenvolvimento foi menor ainda. Nela contém dois pacotes:

- a) `locadora.client.controller`: Foram criadas quatro classes possuindo a maior delas setenta e quatro linhas, que implementam o tratamento específico para a tela de locação de vídeos. Todas as outras utilizaram recurso do *framework* relacionados a implementação do modelo CRUD;
- b) `locadora.client.main`: Pacote contendo uma classe com o método `main()` para execução do cliente.

Para o desenvolvimento das telas também houve benefícios. Na camada web a utilização de *Custom Tags* e do padrão *Composite View* permitiram reutilizar partes das telas que se repetiam no sistema.

Comparando o *framework* desenvolvido ao trabalho correlato Gênesis, o *framework*

desenvolvido destaca-se por utilizar as especificações EJB3 e facilitar o desenvolvimento de telas no modelo CRUD. O *framework* 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.



## 4 CONCLUSÕES

Durante o desenvolvimento do presente trabalho houveram vários desafios relacionados às tecnologias usadas, porém o maior deles foi projetar um software reutilizável e orientado a objetos. Gamma et al. (2000, p. 15) afirma que um projeto desses deve ser específico para o problema a resolver, mas também genérico o suficiente para atender futuros problemas e requisitos. O *framework* desenvolvido baseou-se nesses princípios. A escolha de padrões de projeto teve suma importância para o desenvolvimento do trabalho. Eles deram o norte para a especificação e desenvolvimento do projeto.

Verificou-se que a separação entre as camadas de uma aplicação é um dos mais importantes princípios do projeto de software. Sistemas desenvolvidos utilizando padrões e arquiteturas como o MVC aumentam a reutilização de suas camadas. Os *frameworks* têm grande importância na construção de um sistema, por proverem soluções para um conjunto de problemas, facilitando o processo de desenvolvimento.

Todos os objetivos específicos do trabalho foram desenvolvidos. A implementação da camada servidora do *framework* foi a parte complicada do desenvolvimento. A falta de conhecimento da especificação EJB3 e também a inexperiência da utilização do servidor de aplicação JBoss, fizeram com que fosse investido várias horas de estudo.

No desenvolvimento do presente trabalho também houve a reutilização de *frameworks* e componentes de terceiros. Essa reutilização facilitou a implementação do projeto e trouxe um ganho considerável no tempo de desenvolvimento do *framework*.

Os resultados também apresentados pelo *framework* desenvolvido comprovaram que sistemas de informação podem ser simplificados e desenvolvidos com o foco na implementação da lógica de negócio da aplicação.

O *framework* desenvolvido possui algumas limitações referentes à formatação de campos e notificação das telas Thinlet quando o modelo é alterado, para que as telas possam ser recarregadas. Os dois itens foram incluídos na seção de extensões para o trabalho.

### 4.1 EXTENSÕES

A seguir são sugeridas extensões para o *framework* desenvolvido:

- a) desenvolver novas *Custom Tags* que representem componentes de tela Thinlet tais como: Grade com paginação e Calendário;
- b) para as *Custom Tags* desenvolvidas também será necessário criar classes controladoras para as telas que contenham esses novos componentes visuais;
- c) criar novas classes controladoras para telas que representem dados de forma diferentes tal como árvore de dados;
- d) implementar notificação das telas Thinlet quando o modelo é alterado, para que as mesmas possam ser recarregadas;
- e) desenvolver tratamento de exceções na camada cliente e servidora do *framework*.

## REFERÊNCIAS BIBLIOGRÁFICAS

AHMED, Khawar Z.; UMRYSH, Cary E. **Desenvolvendo aplicações comerciais em Java com J2EE e UML**. Tradução Eveline Vieira Machado. Rio de Janeiro: Ciência Moderna, 2003.

ALMEIDA, Rodrigo Rebouças de. **Model-View-Controller**. [S.l.], 2007. Disponível em: <<http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/arqu/mvc/mvc.htm>>. Acesso em: 22 abr. 2007.

ALUR, Deepak; CRUPI, John; MALKS, Dan. **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, Robert. **Thinlet**. [S.l.], 2005. Disponível em: <<http://thinlet.sourceforge.net/home.html>>. Acesso em: 01 set. 2006.

BOND, Martin et al. **Aprenda J2EE: com EJB, JSP, Servlets, JNDI, JDBC e XML**. Tradução João Eduardo Nóbrega Tortello. São Paulo: Pearson Education, 2003.

COCKBURN, Alistair. **Escrevendo casos de uso eficazes: um guia prático para desenvolvedores de software**. Tradução Roberto Vedoato. Porto Alegre: Bookman, 2005.

FOWLER, Martin. **Padrões de arquitetura de aplicações corporativas**. Tradução Acauan Fernandes. Porto Alegre: Artmed, 2006.

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

RED HAT. **JBoss EJB3**. [S.l.], 2007a. Disponível em: <<http://labs.jboss.org/portal/jbossejb3>>. Acesso em: 09 set. 2006.

\_\_\_\_\_. **JBoss TrailBlazer**. [S.l.], 2007b. Disponível em: <<http://trailblazer.demo.jboss.com/EJB3Trail/>>. Acesso em: 23 abr. 2007.

\_\_\_\_\_. **JBoss IDE for Eclipse**. [S.l.], 2007c. Disponível em: <<http://labs.jboss.com/jbosside/>>. Acesso em: 14 maio 2007.

SOUZA, Gabriela; PIRES, Carlos; BARROS, Márcio. PATI-MVC: padrões MVC para Sistemas de Informação. In: CONFERÊNCIA LATINO-AMERICANA EM LINGUAGENS DE PADRÕES PARA PROGRAMAÇÃO, 3., 2003, Pernambuco. **Anais...** Recife: CIN/UFPe, 2003. Disponível em:

<[http://www.cin.ufpe.br/~sugarloafplop/final\\_articles/13\\_Pati-MVC.pdf](http://www.cin.ufpe.br/~sugarloafplop/final_articles/13_Pati-MVC.pdf)>. Acesso em: 26 ago. 2006.

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. **Core J2EE Patterns**. [S.l.], 2007a. Disponível em: <<http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html>>. Acesso em: 22 abr. 2007.

\_\_\_\_\_. **The J2EE tutorial**. [S.l.], 2007b. Disponível em:

<[http://java.sun.com/j2ee/tutorial/1\\_3-fcs/doc/EJBConcepts3.html](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts3.html)>. Acesso em: 25 abr. 2007.

\_\_\_\_\_. **Custom tags in JSP pages**. [S.l.], 2007c. Disponível em:

<[http://java.sun.com/j2ee/tutorial/1\\_3-fcs/doc/JSPTags.html](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags.html)>. Acesso em: 12 maio. 2007.

\_\_\_\_\_. **Java naming and directory interface**. [S.l.], 2007d. Disponível em:

<<http://java.sun.com/products/jndi/>>. Acesso em: 14 maio. 2007.

\_\_\_\_\_. **Annotations**. [S.l.], 2007e. Disponível em:

<<http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>>. Acesso em: 29 jun. 2007.

MÖBIUS, Dirk. **Thing** - a GUI editor for Thinlet. [S.l.], 2007. Disponível em:

<<http://thing.sourceforge.net/>>. Acesso em: 15 maio. 2007.

YODER, Joseph; JOHNSON, Ralph; WILSON, Quince. 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.