Sérgio Crespo Coelho da Silva Pinto

Composição em WebFrameworks

Tese apresentada ao Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro com parte dos requisitos para a obtenção do grau de Doutor em Ciências em Informática.

Orientador: Prof. Carlos José Pereira de Lucena

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 04 de Agosto de 2000.

Esta tese é dedicada a duas pessoas que tive o prazer de conhecer no decorrer de minha vida acadêmica: Marisa e Carlos Lucena. Com ambos tive o privilégio de aprender e realizar grandes projetos.

Agradecimentos

Esta tese não teria sido possível sem a ajuda indispensável de tantas pessoas. A elas, eu gostaria de manifestar os meus mais sinceros agradecimentos.

Ao Mestre dos Mestres, Professor Carlos José Pereira de Lucena, com que tive o prazer de conhecer, ser seu orientado, ser seu aluno e hoje amigo. Pelo seu constante encorajamento, apoio, paciência, comentários, incansáveis leituras e importantes contribuições que ajudaram a manter sempre no rumo esta tese.

Aos professores Arndt von Staa, Asterio Kiyoshi Tanaka, Raul Martins, Luiz Tucherman por participarem de minha banca.

A Marisa Lucena, que nos conhecemos lá na COPPE e desde então participamos de vários projetos e que algumas vezes foi uma segunda mãe.

Aos meus irmãos Eduardo, Alexandre e Nelson que sempre me ajudaram nas horas difíceis mesmo que algumas vezes à distância.

Aos professores do DI.

As funcionárias Ruth Maria de Barros Fagundes de Sousa, Rosane Teles Lins Castilho, Isabela Achkar de Mendonça e Fatima Maria Holanda da Silva.

À Vera Sotero e Sandra Schwabe que sempre moveram montanhas quando precisei de ajuda.

Aos amigos da Fundação Padre Leonel Franca e LES, em especial ao Luis Fernando Barbosa que faz milagres em nossos laboratórios!

Aos amigos Angelo Cister, Laurinha Cister, Rosa Costa e Paulo Cabral por sua amizade desde que cheguei ao Rio.

Aos amigos Clifton e Gisela Clunie no Panamá por repartirem comigo os bons e maus momentos desde o mestrado.

A Regina Lima pela sua amizade neste vários anos de Rio de Janeiro.

A Adriana Portella e sua familia por me receberem em sua casa quando um disco resolveu sair do lugar!

Aos amigos Mafe, Viviane, Ayrton, Clécio, Mariela e Alessandro Fabrício pelos importantes comentários, idéias e contribuições ao meu trabalho.

Aos amigos Renato Cerqueira e Carlos Cassino, meus gurus em Lua, por sua constante disposição em ajudar quando um "bug" aparecia.

Aos amigos Fábio Porto, Fernanda Lima, Rivera, Natacha pelas conversas e palpites nos corredores do DI.

Aos amigos do TecComm que durante muito tempo compartilharam comigo deste trajeto, fornecendo palpites, dando sempre força e principalmente pela amizade sincera. Esta galera, companheira dos fins de semana, são: Marcus Eduardo Markiewicz, João Alfredo P. de Magalhães, Luiz Mário C. P. M. de Farias, Matheus C. Leite, Daniel Abadi Orlean, Francisco dos Reis Ferreira, Otavio Rezende Silva, Flávia Ejzenberg, Luidi Xavier Fortunato, José Alberto Sardinha e Akeo Tanabe.

Aos amigos de KidLink, onde partilhamos vários projetos.

Aos amigos Osório, MarceloW, AnaP, Denise e Brandão por quebrarem vários galhos na universidade.

A Arete Porciumcúla de Ávila e Cláudia Raupp pela sua ajuda na fase inicial de meu doutorado.

Aos professores Silvia Dutra e Candido Fonseca da Silva pela constante ajuda e apoio do centro 6 durante os 4 anos de estudo.

Resumo

Esta tese apresenta os problemas identificados sobre composição de *WebFrameworks* em plataforma Web. Como forma de abordagem do problema, uma linguagem específica de domínio (*DSL*) chamada *WebCompose* foi desenvolvida. O referencial teórico para a construção da *WebCompose* foi a teoria de *Abstract Design View e Abstract Design Object*, que enfatiza reutilização e *"separation of concerns*", e define um conjunto de propriedades para a construção de *design* flexível.

Um estudo de caso foi implementado, de forma a consolidar a *DSL*, os métodos de composição e validar a sua utilização no mundo real. O estudo de caso utilizou todos os comandos da *DSL* e abordou todos os aspectos problemáticos relacionados à composição em *WebFrameworks*.

Abstract

This thesis presents all identified problems related to the composition of WebFramework in the Web Platform. The problem was approached through a domain specific language (DSL) called *WebCompose* developed for this purpose. The theoretical background used for the design of *WebCompose* comes from the Abstract Design Views and Abstract Design Object which emphasizes separation of concerns and define a set of properties for the realization of flexible design. The case study illustrates the use of all the comands in the DSL and all composition situations identified.

Índice

Índice de figuras

Índice de tabelas

Capítulo 1

Introdução

Resumo

Este capítulo apresenta uma introdução ao trabalho realizado, enfocando a motivação para o tema, e descreve a estrutura da tese.

1.1 Introdução

Em todos os países industrializados e em desenvolvimento, os sistemas de computação tornam-se cada vez mais importantes economicamente. Mais e mais produtos incorporam, de alguma forma, sistemas de computação em suas atividades. O *software* representa, então, uma grande e crescente parcela no custo total do produto final. O funcionamento da moderna economia depende da habilidade de se produzir *software* de forma confiável, a um preço competitivo e em tempo reduzido [Sommerville 1995], [Umar 1997].

Com o advento da Web, uma nova plataforma se tornou disponível, introduzindo uma nova forma de disponibilizar a informação para consulta. A interface mais utilizada é o *browser*, e a forma pela qual a informação se torna disponível é por meio de servidores *World Wide Web*(WWW) que utilizam protocolo *HTTP*.

Hoje, diversos tipos de serviços, antes somente disponíveis nos próprios locais de instalação, tais como caixas eletrônicos para consultas a saldo, transferência de valores, etc., estão disponíveis para que qualquer usuário possa realizar uma série de operações sem que seja necessário seu deslocamento. Basta utilizar sua conexão *Internet* e o seu *browser* para poder ter acesso aos mais variados serviços.

A *Internet*, em números de hoje, representa um total de 86.689 milhões de servidores instalados que oferecem algum tipo de serviço. Existem, aproximadamente, 2.1 bilhões de páginas Web difundindo informação. Estes números crescem de forma exponencial [NetSizer site], [Nua site] e [Statistics on the Web site].

A velocidade com que surgem novas necessidades provoca um constante repensar sobre a forma de se desenvolver sistemas de *software*.

Diversas metodologias são projetadas de forma a atender a estas necessidades; porém, dois aspectos chamam mais atenção: a tecnologia a ser utilizada para o desenvolvimento de *software* e a possibilidade da reutilização de rotinas, bibliotecas e serviços já desenvolvidos.

Segundo Umar [Umar 1997], as novas tecnologias para o desenvolvimento de aplicações têm-se centrado em :

- Uso da *World Wide Web* (Web);
- Aplicações Cliente/Servidor.

Esta troca de paradigma, de Orientação a Objetos para plataforma Web, procura responder questões como [Bassett 1997] e [Mattsson 2000]:

- Como construir novas aplicações usufruindo das vantagens deste novo paradigma?
- Que estratégias são necessárias para a reutilização de serviços já existentes?
- Que tipo de estrutura é necessária para dar suporte à reutilização e ao desenvolvimento rápido e confiável de sistemas de *software*?

Estas perguntas levam-nos à motivação deste trabalho.

1.2 Motivação

Historicamente, desde os anos 60 com Simula 67, componentes de *software* reutilizáveis têm sido viabilizados por meio de bibliotecas [Mattsson 2000].

Com a integração de dados e operações dentro de objetos e classes, proporcionou-se um maior grau de reutilização. Desta forma, bibliotecas passaram a utilizar a herança como forma de facilitar a especialização de classes já existentes.

A reutilização de classes em bibliotecas permite somente o aproveitamento de pequenos pedaços de código, não proporcionando, assim, uma granularidade maior de reuso.

A tecnologia de *frameworks* orientados a objetos procura realizar um alto grau de reutilização.

Um *framework*, segundo [Johnson 1991][Gamma et alli 1995], é um conjunto de objetos que colaboram entre si, com o objetivo de atender a um conjunto de responsabilidades para uma aplicação específica ou um domínio de aplicação.

"Um *framework* é um conjunto de classes que constitui um *design* abstrato para a solução de uma família de problemas" [Forehlic et alli 1998][Johnson e Foote 1988].

A idéia de se utilizar *frameworks* para o desenvolvimento de *software* com alto grau de flexibilização e reuso tem demonstrado ser eficiente em projetos já realizados no Laboratório de Engenharia de Software – LES – da PUC-Rio, onde um conjunto de *frameworks* foi desenvolvido para as áreas de educação (ContentNet [Silva 2000], ALADIN [Crespo, Fontoura, Lucena, Moura 1998]) e comércio eletrônico (Vmarket [Ripper 1999], CommercePipe [Maia 2000], [Ripper, Fontoura, Maia, Lucena 2000] e [Fortunato 1999]).

Michael Mattsson [Mattsson 2000], em sua tese de doutorado, fez um levantamento dos problemas relacionados à composição em *frameworks* orientados a objetos.

A partir deste levantamento, realizamos um estudo sobre como este trabalho pode se refletir em aplicações para a plataforma Web. Esta é a principal motivação para esta tese.

Em aplicações na plataforma Web, a principal característica que podemos observar é o assincronismo no que se refere à execução de serviços e à predominância de sistemas baseados em linguagens não orientadas a objetos, e sim orientadas a *scripts*.

Assim, alguns novos problemas foram identificados, e uma nova teoria foi utilizada para dar suporte aos problemas identificados inicialmente por Mattsson e aos novos decorrentes da mudança da plataforma de desenvolvimento. A teoria necessária deveria contemplar questões sobre:

- Reutilização;
- "*Separation of Concerns*";
- Separação da interface do usuário da aplicação.

A teoria de *Abstract Design View e Abstract Design Object* (*ADV/ADO)*, que está fortemente solidificada em [Cowan and Lucena 1995], [Alencar, Cowan and Lucena 1995a], [Alencar, Cowan and Lucena 1995b], [Alencar, Cowan and Lucena 1995c] e [Alencar, Cowan, Lucena et alli 1995], aborda de forma precisa estas questões e foi utilizada como base para que o conceito de *frameworks* orientados a objetos pudessem ser transportados para o universo da Web.

ADVs/ADOs mostrou-se muito eficaz para os problemas identificados de composição de *WebFrameworks*, que pode ser visto como um conjunto de serviços que permitem colaboração entre si e composição por meio de *roles*. As composições dos serviços definirão o perfil de uma aplicação a ser gerada.

Desta forma, a teoria de *ADVs/ADOs* foi estendida para dar suporte aos novos problemas identificados, como a adaptação do termo *ADV* para *ADVService* e do

termo *ADO* para *ADOService*. Também novas propriedades foram anexadas as já existentes em *ADVs/ADOs*.

Em resumo, para os problemas de composição de *WebFrameworks*, foi utilizada a tecnologia de *frameworks* para proporcionar flexibilização e reutilização em grande escala, e *ADVs/ADOs* como referencial teórico.

A seguir, descreveremos a estrutura da tese.

1.3 Estrutura da Tese

A tese apresenta no capítulo 2, intitulado "Composição de *frameworks*: análise das principais abordagens", um apanhado geral sobre o tema composição de *frameworks* orientados a objetos. Inicialmente, apresenta-se uma introdução aos conceitos básicos sobre *frameworks* que inclui: conceitos, classificação, papéis dos diversos usuários, arquitetura básica, pontos de flexibilização e comparação com outras abordagens. Após esta introdução, apresenta-se uma seção que discute o tema composição com base nos dois principais trabalhos na área desenvolvidos pelos autores Wolfgang Pree e Michael Mattsson. Apresentam-se os principais pontos dos enfoques dos dois autores com destaque para os problemas e propostas de soluções sugeridas.

O capítulo 3, cujo título é "*Frameworks* e Plataforma Web", apresenta, inicialmente, algumas abordagens de desenvolvimento de *software* baseadas em *frameworks*. A seguir, apresenta-se uma nova alternativa para o desenvolvimento de *frameworks* para a Web, de forma a poder realizar comparações entre os estilos. Uma vez apresentada esta nova abordagem, reposicionamos os problemas de composição elicitados no capítulo 2 para esta nova plataforma. A seguir, identificamos novos problemas decorrentes da mudança de ambiente e uma nova denominação para *frameworks* em plataforma Web é proposta: *WebFrameworks*.

O capítulo 4 chama-se "*ADVs/ADOs* como uma teoria para suporte à composição de *WebFrameworks*". Este capítulo apresenta a teoria de "*Abstract Design View* - *ADV* e *Abstract Design Object* - *ADO*" que é estendida para a plataforma Web com seus conceitos adaptados. Os conceitos de *ADVService/ADOService* são introduzidos, com destaque para a noção de *roles* neste novo contexto. Suas propriedades são apresentadas junto com suas principais características e exemplos. Esta teoria oferece suporte à composição de *WebFrameworks*. No final, propõe-se uma notação gráfica de *design* para permitir o *design* detalhado de aplicações baseadas na Web com a

explicitação de *roles* e de serviços: o *ADVServiceChart*, baseado na teoria de *ADVChart*. Esta notação gráfica desempenhará o mesmo papel da *UML* na especificação de *frameworks* orientados a objetos.

No capítulo 5, intitulado "*WebCompose*: uma linguagem de *script* específica de domínio baseada em *framelets* para composição de *WebFrameworks*", apresentamos uma linguagem específica de domínio (*DSL*), chamada *WebCompose*. Utiliza-se uma abordagem de *framelets* para a sua construção em conjunto com os conceitos de *ADVServices* e *ADOService*. Realiza-se uma comparação com outras linguagens que tratam problemas similares. *WebCompose* permite a construção e composição de serviços de *WebFrameworks*.

No capítulo 6, apresenta-se um estudo de caso no qual um portal horizontal é desenvolvido utilizando-se os comandos da *WebCompose* bem como *ADVServiceChart/ADOServiceChart* para a representação gráfica do *design*.

No capítulo 7, apresenta-se a conclusão da tese e trabalhos futuros, onde fica evidenciada a facilidade com que aplicações podem ser geradas por meio de composição de serviços na Web. A *DSL WebCompose*, mostrou-se robusta para permitir todas as formas de composição identificadas no Capítulo 3. Em trabalhos futuros, uma nova arquitetura para *WebFrameworks* é proposta usando-se a tecnologia de *JavaSpace* [Freeman, Hupfer, Arnold 1999] .

Capítulo 2

Composição de *frameworks*: análise das principais abordagens

Resumo

Este capítulo apresenta um apanhado geral sobre o tema "Composição de *frameworks* orientados a objetos". Inicialmente será fornecida uma introdução aos conceitos básicos sobre *frameworks* que inclui: conceitos, classificação, papéis dos diversos usuários, arquitetura básica, pontos de flexibilização e comparação com outras abordagens. Após esta introdução, será apresentada uma seção que discute o tema da composição com base nos dois principais trabalhos na área desenvolvidos pelos autores Wolfgang Pree e Michael Mattsson. Serão apresentados os principais pontos dos enfoques dos dois autores com destaque para os problemas e propostas de soluções sugeridas.

2.1 Introdução

Segundo [Sommerville1995], a Engenharia de Software preocupa-se com teorias, métodos e ferramentas necessárias para o desenvolvimento de sistemas de *software*. Em muitos casos, os sistemas de *software* são grandes e complexos. Eles são abstratos, diferenciando-se dos diversos outros produtos da área de engenharia. Engenheiros de *softwares* modelam partes dos problemas tratados no cotidiano para um mundo computacional, utilizando-se de diversos modelos para uma melhor representação dos problemas elicitados.

Com o crescente desenvolvimento das áreas de *hardware*, *software* e das áreas de telecomunicações, uma nova abordagem de desenvolvimento de sistemas faz-se necessária para que possamos ter produtos de *softwares* mais confiáveis, portáveis e, principalmente, para que possamos construir *softwares* de forma rápida e competitiva para um mercado hoje muito mais veloz e exigente [Mattsson 2000].

Um dos principais objetivos da Engenharia de Software é a reutilização de sistemas já existentes. A orientação a objetos fornece-nos facilidades para que classes possam ser reutilizadas, bem como métodos por meio de seus mecanismos de herança e polimorfismo [Lundberg & Mattsson 1996]. Design Patterns [Gamma et alli 1995] é outra abordagem para reuso em maior escala do que simples classes ou métodos, aqui o enfoque é a reutilização de *designs* e soluções já vivenciadas.

Com a necessidade de um desenvolvimento rápido e confiável, surgiu uma nova tecnologia em Engenharia de Software que permite que uma família de produtos de *softwares* possa ser gerada a partir de uma **única matriz**. Essa nova tecnologia chama-se *frameworks*.

Algumas definições de frameworks:

- Segundo [Johnson1991][Gamma et alli 1995], um *framework* é um conjunto de objetos que colaboram com o objetivo de atender a um conjunto de responsabilidades para uma aplicação específica ou um domínio de aplicação.
- Para Mattsson [Mattsson 1996, 2000], um *framework* é uma arquitetura desenvolvida com o objetivo de atingir a máxima reutilização, representada como um conjunto de classes abstratas e concretas, com grande potencial de especialização.
- Para [Forehlic et alli 1998][Johnson e Foote1988], um *framework* é um conjunto de classes que constitui um *design* abstrato para solução de uma família de problemas.
- Outra definição de *framework* interessante é dada por Buschmann em [Buschmann et alli 1996] e em [Pree 1995], na qual um *framework* é definido como um *software* parcialmente completo (subsistema) projetado para ser instanciado. Isto define uma arquitetura para uma família de subsistemas e oferece os construtores básicos para criá-los. Também definem-se os lugares ou pontos[1] nos quais uma adaptação do código para um funcionamento específico de certos módulos deve ser feita.

[1] Estes pontos na literatura são conhecidos como *Hot-Spots* e são, em orientação a objetos, classes abstratas que possuem métodos vazios que devem ser especializados por herança ou delegação para funcionamento específico (customizado) de determinadas partes do sistema a ser gerado.

2.2 *Frameworks*: conceitos gerais

Iniciaremos esta seção destacando os termos básicos utilizados e os papéis que os diversos atores assumem nesta abordagem.

Um *framework*, como já foi definido por vários autores, é uma aplicação semi-acabada que, muitas vezes, pode ser um sistema inteiro ou, às vezes, um subsistema. Ele procura descrever os objetos e suas interfaces presentes no *design*, fornecendo informações sobre seus fluxos de controle entre as diversas classes, e mapeia, também, as responsabilidades dos objetos [Wirfs-Brock, Johnson 1990], [Wirfs-Brock 1999] e [Fayad 1999].

Frameworks também promovem reutilização de implementações; porém, isto é menos importante do que as suas interfaces internas e a maneira pela qual elas se comunicam. Este é um reuso de alto nível de *design*, e *frameworks* são uma boa abordagem para esta proposta de reutilização. Tipicamente um *framework* é implementado utilizando-se alguma linguagem de programação orientada a objetos tipo Java, C++, Smalltalk, Eiffel, dentre outras [Fayad 1999].

Uma das características dos *frameworks* é a inversão do controle. Tradicionalmente, um desenvolvedor reutiliza componentes de uma biblioteca em que o programa principal chama os componentes quando necessário. O desenvolvedor decide quando e como o componente será chamado e sua interação com os demais. Em um *framework*, o programa principal é reutilizado e o desenvolvedor decide o que será *plugado* dentro dele, determinando a estrutura geral e o fluxo de controle dos programas[Fayad 1999].

2.2.1 Benefícios da utilização de *frameworks* no desenvolvimento de *software.*

Os benefícios primários que podemos identificar no uso de *frameworks*, segundo [Fayad 1999], [Jacobsen, Kristensen,Nowack 1997] e [Froehlich, Hoover, Liu and Sorenson 1997a], são:

- Modularidade: *frameworks* aumentam a modularidade, pois encapsulam detalhes de implementação sob interfaces bem definidas e estáveis. Esta modularidade auxilia na qualidade do *software*, localizando os lugares de impactos no *design* e nas trocas de implementações, reduzindo o esforço necessário para entender o *design* e para realizar futuras manutenções;

- Reutilização: interfaces estáveis presentes nos *frameworks* aumentam o potencial de reutilização pela definição de componentes abstratos que podem ser redefinidos para criarem novas aplicações. O aproveitamento dos componentes já definidos aumentam também a produtividade dos programadores, aumentando, por sua vez, a qualidade, desempenho, confiabilidade e interoperabilidade do *software*;
- Extensibilidade: um *framework* aumenta a extensibilidade na medida em que oferece métodos *hook* explícitos [Pree 1995]. Estes permitem que aplicações possam ser estendidas usando-se as interfaces estáveis já presentes.

2.2.2 Classificação dos *frameworks*

Frameworks orientados a objetos podem ser caracterizados por diferentes dimensões. As mais impotantes são: quanto ao domínio do problema, quanto à estrutura do *framework* e quanto ao seu uso [Fayad 1999] e [Froehlich, Hoover, Liu and Sorenson 1997a], Froehlich, Hoover, Liu and Sorenson 1997b].

Quanto ao domínio do problema

- *Frameworks* de aplicação cobrem funcionalidades que podem ser aplicadas a diferentes domínios. Exemplo: *frameworks* de interfaces gráficas (GUI) [Weinand 1994].
- *Frameworks* de domínio capturam o conhecimento e as peculiaridades de um problema específico. *Frameworks* de controle de produção e multimídia são exemplos de *frameworks* de domínio [Schmidt 1995].
- *Frameworks* de suporte oferecem serviços de baixo nível em sistemas como, por exemplo, *drivers* para dispositivos e acesso a arquivos.

Quanto à estrutura

Se a descrição da estrutura interna do *framework* for visível, torna-se fácil o entendimento do seu comportamento. A estrutura interna de um *framework* está relacionada à concepção da arquitetura de *software*. O princípio da estrutura interna de um *framework* orientado a objetos é descrito pela arquitetura do *framework*. As principais arquiteturas para *frameworks* são [Buschmann 1994]:

- *Framework* com arquitetura em camadas

Ajuda a estruturar aplicações que podem ser decompostas em grupos de subtarefas com diferentes níveis de abstrações.

- *Framework* com arquitetura *pipes and filters*

São utilizados para estruturar aplicações que podem ser divididas em várias tarefas complexas e independentes, que devem ser realizadas em alguma seqüência fortemente determinada ou em paralelo.

- *Framework* com arquitetura MVC (*Model View Controller*)

Define uma estrutura para aplicações interativas que separam a sua interface com o usuário do seu núcleo de funcionalidade.

- *Framework* com arquitetura PAC (*Presentation Abstraction Controller*)

É uma arquitetura utilizada para *software* que possui muita interação com o usuário, possibilitando controles múltiplos da apresentação dos seus modelos de abstração que podem ser decompostos em subfunções independentes.

- *Framework* com arquitetura reflexiva

É utilizado em aplicações que necessitam considerar adaptações futuras, como mudanças do ambiente, tecnologia, e requisitos, mas sem nenhuma alteração em sua estrutura e implementação.

- *Framework* com arquitetura *Microkernel*

É utilizado em sistemas de *software* que promovem diferentes visões de acordo com suas funcionalidades e que tenham que adaptar novos requisitos ao sistema. Ex.: sistemas operacionais.

- *Framework* com arquitetura *Blackboard*

Ajuda a estruturar aplicações complexas que contêm diversos subsistemas especializados para diferentes domínios. Esses subsistemas cooperam entre si para construir a solução para o problema.

- *Framework* com arquitetura *Broker*

É utilizado em sistemas distribuídos, nos quais a interação de componentes é feita por chamadas remotas.

No próximo capítulo, introduziremos o conceito de *frameworks* em plataforma Web, que intitulamos *WebFrameworks*.

Quanto ao uso

Um *framework* orientado a objetos pode ser utilizado de duas maneiras. O usuário deriva novas classes a partir de sua estrutura ou combina classes já existentes.

O primeiro método é conhecido como *architecture-driven* ou *white-box framework* e o segundo como *data-driven* ou *black-box framework*.

- White-Box *framework*:

O principal artifício é desenvolver aplicações baseadas no mecanismo de herança. O usuário do *framework* realiza suas adaptações derivando classes e reescrevendo métodos já definidos em sua estrutura. *Frameworks* desse tipo são mais difíceis de serem utilizados, uma vez que o seu usuário necessita conhecer detalhes de implementação e *design*.

- Black-Box *framework*

O segundo método procura adaptar o *framework* às necessidades específicas da aplicação com base na composição de objetos e componentes existentes. Configurar um *framework* selecionando componentes tende a ser mais simples do que herdar de classes já existentes. Desta forma, *frameworks black-box* tendem a ser mais fáceis de serem utilizados.

2.2.3 Atores envolvidos no processo de criação, utilização e manutenção de *frameworks*.

Existem vários usuários que interagem para o *design*, especialização e manutenção do *framework,* como mostrado a seguir [Froehlich, Hoover, Liu and Sorenson 1997a], [Froehlich, Hoover, Liu and Sorenson 1997b]:

Projetistas de framework: são responsáveis pelo desenvolvimento de toda a estrutura básica do *framework* que é composto por um conjunto de classes e relacionamentos e pela determinação de que pontos, dentre essas várias classes, serão ou não flexíveis. Também fazem toda a tarefa de levantamento de requisitos.

Usuários do framework: são usuários que utilizam o *design* já existente do *framework* para especializá-lo e, desta forma, gerar diferentes aplicações no domínio para o qual o *framework* foi projetado.

Mantenedores do framework: são os responsáveis por redefinir e/ou acrescentar novas funcionalidades ao *design* do *framework*, permitindo que novas características possam ser geradas no momento da geração de aplicações por parte dos usuários do *framework*.

Os diferentes papéis apresentados não devem ser, necessariamente, desempenhados por pessoas diferentes.

2.2.4 *Framework* vs outras abordagens

Nesta seção, faremos uma comparação entre *frameworks* e outras abordagens de desenvolvimento/reutilização de *software*.

***Design pattern* orientado a objeto**

Um *framework* difere de um *design pattern* em três aspectos:

- Um *Design Pattern* é mais abstrato do que um *framework*. Os *frameworks* têm a sua apresentação na forma de código (implementado), enquanto somente os exemplos de *design patterns* são apresentados de forma implementada;
- Os *Design Patterns* são arquiteturas menores do que os *frameworks*. Um *framework* pode conter vários *designs patterns*, mas o contrário não é possível.
- Os *frameworks* são mais especializados do que os *Designs Patterns*, sendo mais específico para um domíno de aplicação. Já os *Design Patterns* são mais gerais e podem ser aplicados em diversos domínios de aplicação.

Linguagem de *pattern*

Linguagens de *patterns* diferem de *frameworks* no sentido em que uma linguagem de *pattern* descreve como fazer um *design*, enquanto um *framework* é o próprio *design*. Linguagens de *pattern**s*** complementam *frameworks*, desde que ensinem aos projetistas como utilizá-los e descrevê-los.

Biblioteca de classes

Bibliotecas de classes são um conjunto de classes relacionadas que têm funcionalidades de propósito geral. As classes em uma biblioteca de classes não são relacionadas a um domínio de aplicação específico, como no caso de classes em um *framework*. A diferença entre uma biblioteca de classes e um *framework* é o grau de reutilização e o seu impacto na arquitetura da aplicação. Uma classe da biblioteca de classes é reutilizada sozinha, enquanto uma classe de um *framework* é reutilizada juntamente com as outras classes para resolver uma instância de um certo problema.

***Framework* arquitetural**

Um *framework* é a implementação de uma arquitetura comum para uma família de aplicações. Já um *framework* arquitetural é independente de linguagem e determina a estrutura básica de uma aplicação, podendo servir de base para a construção da arquitetura do *framework*.

Como exemplos de *frameworks* arquiteturais, podemos citar a arquitetura do *Model View Controller* [Krasner 1988] e a estrutura de *Pipes e Filters* [Meunier 1995].

Uma aplicação orientada a objeto

Uma aplicação orientada a objeto descreve a execução de um programa que atende a todos os requisitos de uma dada especificação. No entanto, um *framework* captura as

funcionalidades de diversas aplicações no domínio, mas não é executável, já que não cobre o comportamento de uma aplicação específica.

2.2.5 Arquitetura básica e componentes de um *framework*

Como a figura 2.1 mostra, um *framework* possui um *kernel* que contém um conjunto de classes que não são passíveis de adaptação e que permanecem presentes em todas as aplicações geradas. Todas estas classes possuem implementações reais de seus vários métodos e interfaces.

O que diferencia, basicamente, as aplicações geradas são as diversas formas de se adaptar a estrutura básica do *framework* por meio de seus diversos pontos de flexibilização (*hot-spots*).

Um *hot-spot* é uma classe abstrata que não possui implementação e deve ser especializada (customizada) para necessidades específicas da aplicação a ser gerada. A especialização pode ser realizada tanto por herança como por delegação, dependendo da forma como os *hot-spots* foram planejados.

Como um *framework* cobre os requisitos de um determinado domínio, às vezes, faz-se necessário atualizá-lo e, nem sempre, esta é uma tarefa fácil de ser contemplada.

Em alguns casos, surge a necessidade de se utilizar dois ou mais *frameworks* para que um determinado objetivo possa ser atingido. Como exemplo, podemos pensar na existência de um *framework* para editoração de texto. Com esse *framework*, poderíamos gerar os mais variados tipos de editores de texto e com muitas características específicas. Como exemplos, poderíamos citar: editores de propósito geral, editores direcionados a um segmento específico como programação em Cobol, Fortran, C ou Pascal. Todos estes editores têm as suas características especiais como, por exemplo, Fortran que deve ter seus comandos digitados a partir da 7ª coluna e na 6ª são colocados caracteres de controle tais como desvios etc.; já os editores para C, Pascal e Cobol podem iniciar na 1ª coluna sem problemas. Poderíamos também ter um *framework* que satisfizesse os requisitos de editoração gráfica, como por exemplo editores de desenho tipo *PaintBrush* ou *Lview* ou editores para temas específicos, como editores de diagrama de dados ou um editor para a notação *UML* [Booch, Jacobson, Rumbaugh 1998].

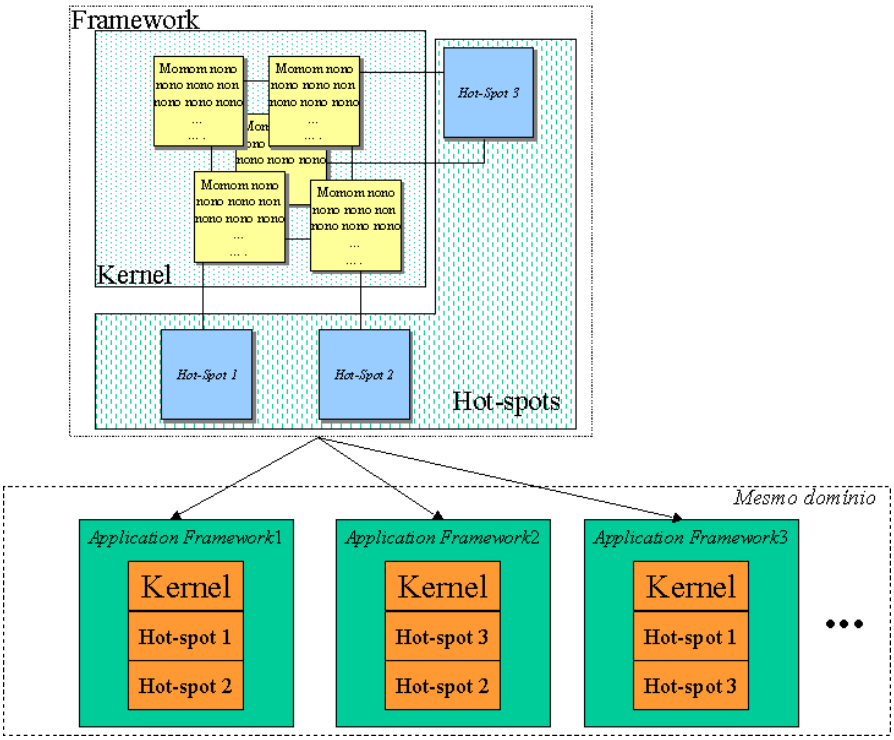

Figura 2.1- *Framework*: visão geral

Um dos vários problemas que poderiam surgir seria o de acrescentar novas funcionalidades dentro dos editores gráficos da seguinte forma: fazer com que o *design* pudesse ser feito de forma gráfica, e toda a documentação e implementação dos métodos na forma de texto.

Teríamos duas abordagens: uma seria redefinir um dos *frameworks* para que em seu *design* tais necessidades fossem contempladas, e a segunda seria realizar a composição de ambos para atingir o mesmo objetivo, o que nos parece mais apropriado. Porém, alguns problemas surgem quando pensamos em composição de *frameworks*. Estes problemas serão discutidos a seguir.

2.3 Composição de *frameworks*: problemas e causas

Os problemas de composição de *frameworks* serão revistos segundo quatro enfoques diferenciados dos seguintes autores Michael Mattsson em ([Bosh et alli 1999], [Mattsson ett alli 1999] e [Mattsson 2000]) e Wolfgang Pree em ([Pree 1999a] e [Pree 1999b]).

Serão apresentadas, a seguir, as duas visões que os autores possuem sobre composição de *frameworks*, apontando dificuldades e soluções por eles propostas.

2.3.1 Visão de Michael Mattsson

Segundo Mattsson [Mattsson & Bosh 1998][Mattsson 2000], a possibilidade de se considerar *frameworks* orientados a objetos como um componente e estes serem passíveis de composição ocasiona vários problemas listados a seguir:

a) Problemas relacionados ao controle do *loop* de execução em *frameworks*.

Uma das grandes funcionalidades que distinguem *frameworks* de outros modelos de *software* é a possibilidade de fazer uso extensivo de ligação dinâmica. Em bibliotecas ou classes tradicionais, a aplicação gerada chama, sempre que necessário, um conjunto de funções que estão presentes nestas bibliotecas, e a própria aplicação é que detém o controle do laço de execução desta rotina. Porém, em *frameworks* orientados a objetos, a situação é diferente, pois o próprio *framework* tem a responsabilidade do controle e da chamada da aplicação. Isto é conhecido como o princípio *Hollywood*: "*Don't call us-we'll call you*", como pode ser visto na figura 2.2.

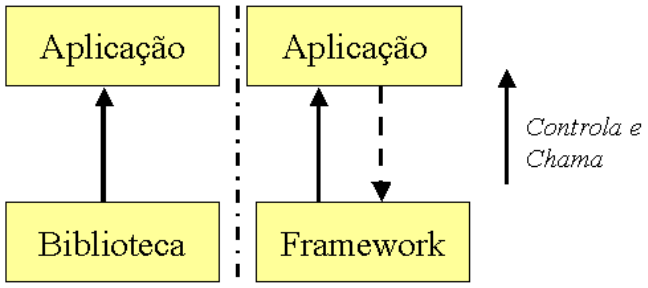

Figura 2.2 - Inversão de controle

Segundo [Mattsson & Bosh 1998], [Mattsson 2000], [Kuikka1998], [Sparks et alli 1996], [OriginalSim1997] e [Tong1998], *frameworks* podem ser caracterizados pela forma como a chamada e o correspondente controle a uma função/método são realizados: *frameworks* ativos(*Calling frameworks)* e passivos*(Called frameworks*), figura 2.2.

Called frameworks não detêm o controle da execução da chamada, ficando esta sob a responsabilidade da função/método chamado. Em outras palavras, correspondem a códigos de bibliotecas como (*Eiffel libraries, Booch's libraries*), nos quais *a "Application code*" usa o *framework* chamando funções e métodos dentro de uma *library* (biblioteca), quando necessário.

Calling frameworks, ao contrário, são entidades ativas presentes na aplicação que chamam e controlam a execução de outras funções/métodos, como na figura 2.3.

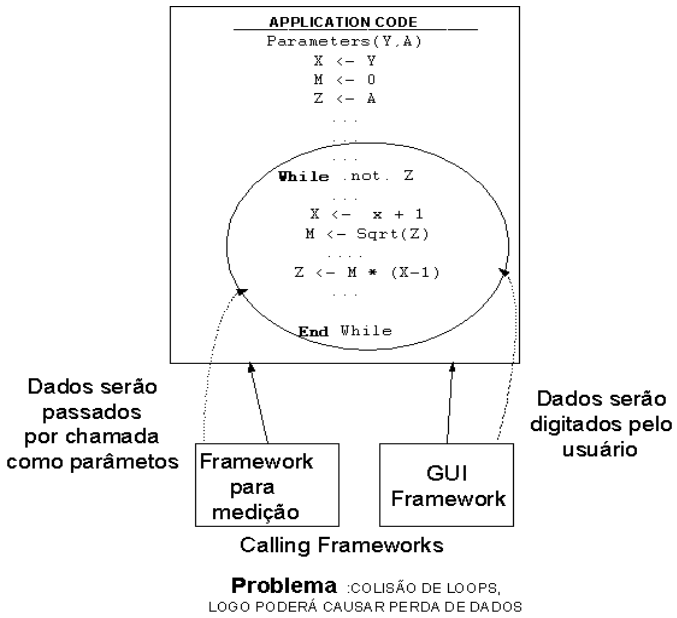

Figura 2.3 - Problema do controle de execução

Na figura 2.3, podemos ver um exemplo do problema acima citado. Neste exemplo, temos 2 *frameworks* ativos (*Calling frameworks*) que disputam a chamada e o controle de um método. Neste caso, podem ocorrer erros quando os dois *frameworks* chamam o mesmo método e este possui algum laço de execução. No exemplo, ambos interagem de forma distinta com o método. Enquanto um aciona a sua interface passando parâmetros, o outro recebe do usuário informações que serão processadas e transformadas para serem enviadas ao método. Devido às velocidades com que os parâmetos são enviados, podem ocorrer problemas de retorno de valor, pois os dados podem ser mesclados e uma resposta não satisfatória será enviada a ambos os *frameworks*, por não se saber por quem ou quando os dados foram passados.

A solução deste problema pode ser simples a partir do momento em que se dispõe do código fonte; porém, nem sempre, o laço de controle está dentro de uma única função/método (este pode estar fracionado entre várias) o que dificulta muito a correção. Este problema toma proporções maiores se não houver o código fonte disponível.

b) A composição de *frameworks* com sistemas legados.

Um *framework* apresenta o *design* para uma aplicação em um domínio particular. Quando uma nova funcionalidade não prevista durante a fase de levantamento de requisitos que originaram o *design* do *framework* é necessária, o engenheiro de *software* pode pensar em utilizar algum sistema legado que contenha essa solução e, então, tentar integrar as classes do sistema legado com as do *framework* de modo a

acrescentar mais funcionalidades para uma futura geração de uma aplicação que necessite destes novos requisitos.

A integração entre *frameworks* e componentes ou sistemas legados não é uma tarefa fácil, uma vez que o sistema legado não pode ser considerado como uma subclasse do *framework*. Isto porque *frameworks* são construídos para adaptação ou extensão e não para composição.

Um dos vários problemas de composição com sistemas legados pode ser observado na figura 2.4.

O *framework* representado na figura 2.4 possui uma classe sensor que necessita de adaptação, porém esta classe possui como atributo uma variável lógica que sinaliza quando um determinado padrão de temperatura foi atingido de forma a poder ser acionado algum mecanismo de precaução. O sistema legado, por sua vez, possui uma estrutura mais rica na definição de sua classe Sensor. Esta é abstrata e foi generalizada para três tipos de medição que são: calor, fumaça e umidade, de forma que, quando um determinado padrão é atingido, a classe Sensor informa ao seu observador que uma ocorrência se originou e uma ação é tomada.

Um processo natural seria compor ambas as classes que possuem o mesmo nome de forma a gerar uma aplicação mais robusta na medição do sensor.

Neste caso, a classe Sensor NÃO pode ser utilizada porque os nomes não designam a mesma classe em termos de estrutura.

Uma possível solução seria utilizar o *Design Pattern Adapter* [Gamma et alli 1995], porém, para cada elemento da interface que necessite adaptação, o engenheiro de *software* deverá definir métodos que enviem uma chamada para o correspondente método no sistema legado.

Outro problema que também pode ocorrer é a utililização de um objeto do sistema legado em várias partes do *framework*, aumentando, assim, a quantidade de adaptação necessária para que o problema de similaridade de nome e diferenças de estrutura e ou semântica possam ser resolvidos [Lundberg & Mattsson 1996].

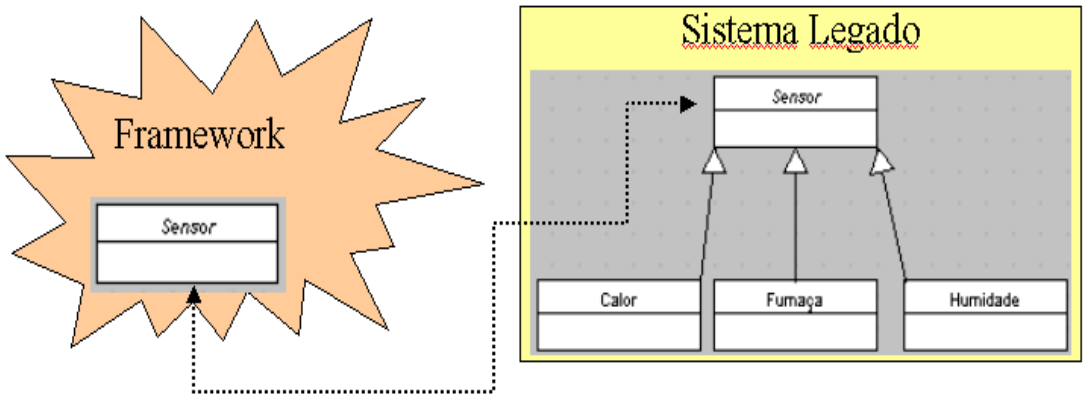

Figura 2.4 - Composição com sistemas legados

Uma outra abordagem para a solução deste problema de composição com sistemas legados é a utilização de papéis ("*roles*") na definição das interfaces das classes [Reenskaug 1996]. Reenskaug considera a interação entre um conjunto de objetos como uma instância de um "modelo de papéis", no qual todos os papéis podem ser selecionados por todos os objetos.

A idéia principal é utilizar o conceito de papéis ("*roles*") de forma a abstrair a noção de classes e enfatizar os requisitos do *framework* expressos em termos de funcionalidades ("*features*"). O próximo capítulo expandirá a discussão sobre o uso de *roles* em *frameworks*.

c) Problemas de carência de requisitos.

Neste tópico, Mattsson enfoca o problema que ocorre quando a composição de dois ou mais *frameworks* para a solução de um problema não cobre todo a conjunto de requisitos em um dado domínio. Este problema é também abordado por Sparks em [Sparks1996].

Se o *framework* é do tipo passivo ("*called framework*"), a carência de requisitos pode ser resolvida com uma interface adicional para ambos os *frameworks*. Esta interface atuaria como um envoltório adicionando o que falta de requisitos de forma a contemplar todo o espaço do domínio, como pode ser observado na figura 2.5.

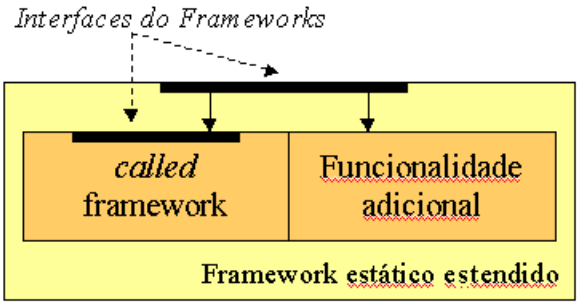

Figura 2.5 - *Framework* estático estendido para completar os requisitos do domínio

Se o problema residir em *frameworks* ativos("*calling frameworks*"), será necessário um *software* do tipo mediador("*mediator*") [Gamma et alli 1995] para minimizar o problema. Neste caso, o *software* mediador deve prover um canal de comunicação entre ambos os *frameworks* ativos, pois eles possuem a característica de controlar o *loop* do evento que chamam. Três abordagens para esse problema são apresentadas:

- *Wrapping*. Para *frameworks* passivos("*called frameworks"*), esta é uma abordagem que viabiliza uma camada de interface para integrar o *framework* que será composto com outros clientes(*frameworks* e/ou bibliotecas) de forma a fechar a lacuna de carência de requisitos;
- Software Mediador. Para *frameworks* ativos(*calling frameworks*), o engenheiro de *software* deveria desenvolver algum tipo de *software* mediador que gerenciasse a interação entre os dois e que contivesse a falta de requisitos, de forma a cobrir a carência que ambos os *frameworks* apresentam;
- Nova análise e extensão. Se o desenvolvedor possuir o código fonte das partes envolvidas, ele poderá fazer com que essa carência seja anexada ao *design*, proporcionando, assim, uma melhor reutilização em futuras aplicações.

d) Superposição de perspectivas.

Este problema, segundo Mattsson, ocorre quando classes do *framework*(e/ou outro *framework* ou componente) a ser reutilizado possuem representações diferentes para o mesmo problema, ou seja, as duas classes foram modeladas com perspectivas diferentes.

Para o autor, podem ocorrer dois tipos de problemas:

- Se as propriedades que as classes representam forem mutuamente exclusivas, então a utilização de herança múltipla pode resolver o problema;

- Se as propriedades forem complementares ou compartilhadas, então o processo de composição torna-se muito mais complexo. Três casos são identificados a seguir:

1. Este caso é muito semelhante aos problemas já citados na composição com sistemas legados na qual um *framework* possui uma propriedade que modela uma situação do mundo real por meio de uma classe, e um outro *framework* modela a mesma situação, porém com representações distintas da classe do primeiro *framework*. Por exemplo, se em ambos os *frameworks* o serviço para modelar uma operação aritmética fosse modelado da seguinte forma, como mostra a figura 2.6.

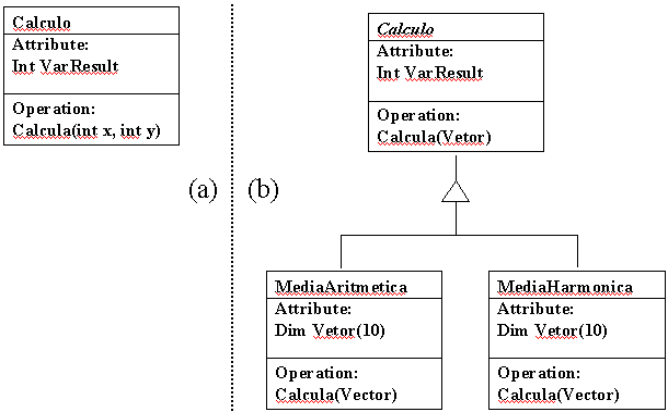

Figura 2.6 - Diferentes perspectivas para um mesmo problema

Na figura 2.6 (a), a classe *Calculo* efetua o seu serviço utilizando um método que recebe dois valores inteiros como entrada e efetua a sua operação que, por exemplo, poderia ser somar os dois números e mostrá-los ao usuário. Como pode ser visto, esta classe não admite adaptação, pois a implementação do método ja está definida na própria definição e *design* da classe. Na figura 2.6 (b), o caso já é bastante diferente, pois existe uma classe abstrata chamada Cálculo que é especializada em outras duas chamadas *MediaAritmetica* e *MediaHarmonica* que implementam o método abstrato de sua super classe "*Calculo*".

Existem duas perspectivas para representar o mesmo problema de executar um cálculo, sendo que de um lado (a) a entrada é por meio de dois inteiros e do outro (b) por meio de um vetor de inteiros com possibilidades de serem anexados outras formas de cálculos de média.

2. Um segundo caso identificado por Mattsson ocorre quando, em um *framework*, uma classe representa uma propriedade usando, por exemplo, um simples atributo e, no outro, esta mesma classe para representar a mesma propriedade utiliza um método.

3. E o terceiro caso ocorre quando a execução de um determinado método exige uma troca de estado em alguma classe no outro *framework*.

As soluções propostas para esse tipo de problema envolvem o uso de herança múltipla, em geral, agregação e generalização.

O inconveniente das três alternativas propostas reside no fato de existirem *frameworks* tipo *black-box* nos quais todas as alternativas ficam inoperantes, uma vez que não se tem acesso ao código fonte e nem a possibilidade de conhecimento do *design* interno para tentar implementar a programação que um determinado método exige. Neste tipo de enfoque, o que é permitido são somente parametrizações. Mattsson resume este estudo apresentando um quadro que contém os problemas elicitados e as suas causas. Isto pode ser visto na tabela 2.1.

Onde:

Coesão do comportamento: é a capacidade de uma classe, presente em um *framework,* se comunicar com outras classes em outro *framework* ou biblioteca, proporcionando atualizações mútuas.

Abrangência de domínio: é a capacidade do *framework* de cobrir todos os requisitos para um determinado domínio de aplicação.

Projeto do framework: é identificar como foi projetado o *framework* e como suas classes foram implementadas de forma a ser permitido o maior reuso possível. Na maioria dos casos, os *frameworks* são projetados para serem especializados por meio de herança ou generalizações e, geralmente, são *black-box* ou *white-box*.

Acesso ao código fonte: é a possibilidade do engenheiro de *software* possuir acesso aos fontes dos *frameworks* para realizar composições de forma mais eficiente e ou adaptar o *design* do *framework* para dar suporte mais *facilmente* a adaptações.

Problemas **Causas**	controle do *loop* de execução em *frameworks*	sistemas legados	carência de requisitos	superposição de perspectivas
Coesão de comportamento	#	+	-x-	#
Abrangência do domínio	-x-	-x-	+	+
Projeto do *Framework*	+	#	-x-	#
Acesso ao código fonte	#	#	#	#

Tabela 2.1 – Causas e problemas em composição de *frameworks*

Legenda:

[+] causa primária do problema

[#] Fator de complicação

[-x-] nenhuma opção selecionada

2.3.2 Visão de Wolfgang Pree

Segundo Wolfang Pree em [Pree 1999a] [Pree 1999b], *frameworks* são um conjunto de classes que formam um esqueleto ou *kernel* que procura abranger todos os requisitos de um determinado domínio de aplicações. Entretanto, este esqueleto é, geralmente, muito complexo e possui um grande número de classes o que gera uma série de problemas para permitir composição [Pree 1999a].

Os problemas encontrados com mais freqüência nestes casos [Pree 1999b] são:

- *design* de *frameworks* típicos é muito complexo e de difícil compreensão, necessitando, freqüentemente, de adaptação de suas numerosas classes e longos ciclos de desenvolvimento.
- reutilização de *frameworks* é difícil. *Frameworks* compreendem, geralmente, um *kernel* de classes de um domínio, sendo necessário um bom entendimento de sua arquitetura básica de forma a ser possível extendê-la.
- a composição de *frameworks* é complexa. Muitas vezes um *framework* assume o controle da aplicação (o que Mattsson chama de *calling frameworks*). Logo dois ou mais *frameworks* assumindo estes controles tornam o processo de composição muito problemático, sendo que, geralmente, é necessário, por vezes, quebrar a integridade dos *frameworks* para atingir objetivos de composição[Pree 1999b][Mezini, Seiter, Lieberherr 2000].

A lista acima, segundo Wolfgang Pree, sugere que o problema não está localizado no princípio básico de construção dos *frameworks*, mas sim na granularidade em que seu *design* é construído.

A proposta de Pree é realizar um *downsizing* radical da granularidade dos *frameworks*, ao que ele chama de *framelets* [Pree 1999a][Pree 199b]. *Framelet* segue o princípio geral dos *frameworks* e, em particular, o princípio *Hollywood*.

Framelet é um pequeno *framework* que possui as seguintes características:

- É pequeno em sua estrutura interna (*design*) (< 20 classes);
- É sempre um *called framelet*, ou seja, nunca assume o controle da aplicação;
- Possui uma interface muito simples de ser entendida.

Tal qual um *framework* convencional, um *framelet* pode ser especializado tanto por composição como por herança. Um *framelet* não é somente um componente reutilizável, mas uma unidade fundamental do *software* em geral [Pree 1999b].

Se um *software* pode ser visto como um conjunto de serviços que suas interfaces implementam, um *framelet* pode ser visto como um subconjunto de tais sistemas. Uma interface que pertence ao *framelet* sem sua implementação (*hot-spot*) (e sendo dentro do *framelet*) é parte da especialização da interface do *framelet*. Uma interface que pertence ao *framelet* junto com sua implementação (e usada fora do *framelet*) é parte do serviço da interface que o *framelet* oferece. Esta é a base para a utilização de *framelets* na reestruturação de sistemas legados. Esta abordagem pode ser vista como uma forma de modularização de *frameworks*. Em um sistema de ponto grande, uma aplicação pode ser construída usando *framelets* como sendo *black-box*, e, em ponto pequeno, o *framelet* propriamente dito é um pequeno *framework white-box* [Pree 1999b].

2.4 Conclusão

Este capítulo apresentou uma visão geral do conceito de *frameworks* destacando seus conceitos, classificação, atores envolvidos, comparação com outras abordagens de desenvolvimento de *software* e arquitetura.

Na seção 2.3, que trata dos problemas sobre composição, os problemas descritos baseiam-se nos trabalhos mais expressivos da literatura, ou seja, nos artigos de Michael Mattsson em [Mattsson & Bosh 1998], [Mattsson ett alli 1999], em uma recente tese de doutorado [Mattsson 2000]) e artigos de Wolfgang Pree em ([Pree 1999a] e [Pree 1999b]). Esses problemas serão agora analisados e resumidos, sendo as soluções propostas aqui discutidas.

Podemos classificar os problemas apresentados em dois grupos: problemas de *design* e problemas de implementação. Estes grupos possuem características que devem ser analisadas nas soluções propostas para os problemas de composição.

- **Problemas de *design***:

a) Problema de carência de requisitos: surge quando a composição de dois ou mais *frameworks* não contemplam todos os espaços de requisitos necessários para um determinado domínio de atuação.

As soluções propostas para este problema não estão localizadas no âmbito de *design* e sim de implementação. As propostas são:

1) ***Wrappers***: utilização de uma camada que disponibilize uma interface para o *framework* e outros(*frameworks* e ou softwares) envolvidos de forma que esta camada contemple os requisitos ausentes. Esta solução é válida somente **para *called frameworks*.**

Discussão: Metodologias como *OMT*, *UML* estão fortemente relacionadas com o paradigma de programação orientada a objetos, o que, em muitos casos, transforma-se em vantagens e, em outros, representa o problema no momento em que se necessita de um outro tipo de representação. Quando citamos o problema de carência de requisitos ou "*framework gap*", como é conhecido, a tendência natural é procurar-se por uma solução de *design* que possa ser refletida em níveis mais operacionais. A utilização de um *software* tipo *wrapper* resolve o problema no nível de programação, mas não reflete a solução no nível de *design.* Sempre que uma nova versão do *framework* for gerada, será necessário reavaliar o *software* desenvolvido e, muitas vezes, refazê-lo.

2) ***Software* mediador**: Seria utilizado para *calling frameworks* e teria o objetivo de formar um canal de comunicação entre os *frameworks* envolvidos no processo de composição, fornecendo também os requisitos que estão ausentes nos *design*.

Discussão: Novamente uma solução de implementação para um problema de *design*. Esta abordagem apresenta o inconveniente de que o *software* mediador desenvolvido fica muito dependente da versão do *framework*, gerando, assim, uma maior complexidade nas atualizações futuras que venham a ocorrer.

3) **Nova análise (extensão)**: Esta alternativa prevê que o usuário que instanciará o *framework* possua os códigos fonte para poder adicionar e alterar o que for necessário para que os requisitos ausentes façam parte do *design* dos *frameworks*.

Discussão: Esta é uma proposta que apresenta muitos problemas dentre os quais destacamos os seguintes:

- Nem sempre o código fonte está disponível para alterações. Em *frameworks* tipo *black-box*, fica inviável;
- Dificuldades de alteração. Neste caso, mesmo que o engenheiro de *software* tenha acesso aos fontes dos *frameworks*, eles foram previamente projetados para atuarem em domínios nem sempre semelhantes. Isto torna muito trabalhoso o

processo de aprendizado dos *designs* dos *frameworks* envolvidos de forma a possibilitar que sejam feitas com sucesso alterações sem impactar as instâncias já construídas a partir dos mesmos. Um *framework* típico hoje em dia não possui menos do que 80 classes e centenas de relacionamentos, o que deixaria esta alternativa praticamente inviável de ser executada com êxito em um período de tempo de curto para médio. Outro ponto que impactaria esta solução seria a forma com que os *frameworks* foram documentados.

b) Problemas de superposição de perspectivas: Ocorre quando mesmas entidades do mundo real são modeladas de forma distintas.

As soluções propostas por [Mattsson 2000] são a utilização de agregação, herança múltipla e delegação.

Discussão: Este é um problema tratado na literatura como *Viewpoints*[Cowan, Lucena 1995], [Finkelstein 1996] e outros.

O uso de herança múltipla possui os seguintes inconvenientes: nem todas as linguagens de programação implementam esta característica. Este tipo de abordagem é amplamente criticado na literatura e acarreta uma sobrecarga muito grande no *design* para o caso de futuras atualizações.

A utilização do mecanismo de agregação implica que o engenheiro de *software* deve possuir o código fonte e toda a documentação do *framework*, se existir, de forma a poder entender o *design*. Se a composição envolver um *framework* tipo *black-box*, isto fica inviável de ser realizado.

Outra forma de abordar o problema é utilizando delegação. Desta forma, não seria necessário alterar o *framework*. Seria necessário que os métodos abstratos fossem implementados pelas classes concretas, utilizando-se *hooks* apropriados. Porém, para *frameworks* tipo *black-box*, esta solução fica comprometida.

- **Problemas de implementação apresentados**:

a) **Controle da Aplicação**: Também conhecido como *framework control*, caracteriza-se pelo fato do *framework* controlar a execução da aplicação. Essa característica presente em *frameworks* tipo "*calling frameworks*".

Discussão: São apresentadas soluções utilizando-se mecanismos de múltiplos *threads*, *wrapping* e re-escrita.

Utilizar múltiplos *threads* de forma que cada *framework* tenha um *thread* próprio para o controle da aplicação possui o inconveniente de se precisar de mecanismos de sincronização caso alguns dos *threads* necessitem de comunicação entre eles.

O mecanismo de *wrapping* deve encapsular cada *framework* de forma a interceptar as chamadas para repassá-las às entidades interessadas. Este tipo de abordagem não funciona se os eventos a serem interceptados não forem externos ao *framework*, e sim internos. Aqui também, nota-se que a solução dada neste nível não possui correspondência no nível de *design*.

A opção de re-escrita deveria excluir o trecho comprometido tanto no *framework* como no sistema legado e substituí-lo por outro adequado. Esta opção é totalmente dependente do código fonte das partes envolvidas, o que nem sempre é possível. Também exige um grande esforço inicial de estudo dos *designs* das aplicações.

b) **Sistemas legados**: este problema diz respeito à dificuldade de se compor um *framework* com um sistema legado de forma que se possa gerar uma aplicação com requisitos não previstos, inicialmente, no *design* do *framework*. Uma vez que *frameworks* são construídos visando extensão e adaptação, geralmente não existe, no *design*, mecanismos que beneficiem futuras composições.

Os autores apresentam duas soluções para o tratamente deste problema:

1) **Utilização do *Design Pattern Adapter***[Gamma 1995].

Discussão: o *Pattern Adapter* converte a interface de uma classe em outra, esperada pelo cliente. O *Adapter* permite que classes que antes não poderiam trabalhar juntas, por incompatibilidade de interfaces, possam agora fazê-lo.

Vale a pena lembrar que os *Patterns Decorator* (que anexa dinamicamente responsabilidades adicionais a um objeto; provê uma alternativa flexível para extensão de funcionalidade, sem uso de mecanismos de herança) e *Visitor* (que representa uma operação a ser executada sobre elementos da estrutura de um objeto) permitem que se crie um nova operação sem que se mude a classe dos elementos sobre os quais ela opera são alternativas para flexibilização.

Porém, o ponto fundamental destas abordagens é que o engenheiro de *software* deverá ter conhecimento das estruturas que necessitam ser compostas adicionando um *overhead* para o desenvolvimento da solução que estará muito dependente da versão do *framework* e do sistema legado.

Este problema tem um reflexo direto no nível de *design* no momento em que uma solução é apresentada em um nível mais baixo, sem contrapartida no *design,* ocasionando um descompasso entre os dois modelos. A solução ideal, neste caso, seria poder obter uma solução de implementação que pudesse ser refletida de forma fácil no *design* do *framework* e dos sistemas legados envolvidos.

c) Utilização de Framelets.

O *framelet* opera em um universo de granularidade muito pequena, de forma a ter o seu *design* muito simplificado e sua interface bem definida e estável. Esta interessante abordagem é recomendada para que um *framework* seja composto de vários *framelets*, facilitando, em muito, o entendimento do *framework*. Porém, quando a composição for entre uma aplicação gerada por um *framework* composto por vários *framelets*, esta vantagem desaparece, pois, neste caso, o que se tem agora é um sistema grande no qual o código fonte não existe, e a colagem do *framelet* com as classes do sistema agora legado fica comprometida pois não há como se saber nada sobre o *design* deste novo sistema legado. Outro problema encontrado também nas outras alternativas é a não reflexão da solução proposta para a camada de *design*.

No próximo capítulo, será feita a transposição dos problemas de composição para a plataforma Web, acrescida de novos problemas que são particulares desta plataforma.

Capítulo 3

Frameworks e Plataforma Web

Resumo

Este capítulo apresenta, inicialmente, algumas abordagens de desenvolvimento de software baseadas em *frameworks*. A seguir, será apresentada uma nova alternativa para o desenvolvimento de *frameworks* baseada na Web, de forma a podermos realizar comparações entre os estilos. Uma vez apresentada esta nova abordagem, reposicionaremos os problemas de composição já elicitados no capítulo 2 nesta nova plataforma e identificaremos novos problemas decorrentes da mudança de ambiente.

3.1 Introdução

No processo de desenvolvimento de uma aplicação, cria-se um programa que atende a todos os requisitos, passando pelas fases de análise e *design*.

Segundo [Mattsson 1996], [Bosh ett alli 1999] e [Froehlich 1999 ett alli], o processo de desenvolvimento de um *framework* é mais difícil e trabalhoso, já que se estuda, agora, soluções para um conjunto de problemas de um determinado domínio. Na análise do domínio, observam-se as características comuns das diversas aplicações. Desenvolve-se o *framework*, e as diversas aplicações serão produzidas a partir da sua instanciação.

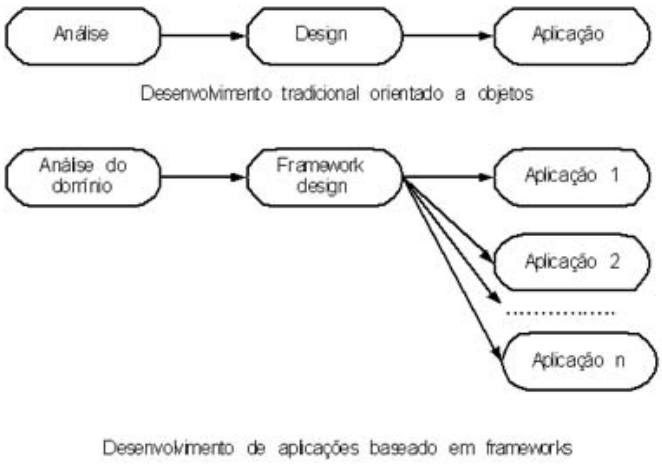

Figura 3.1- Desenvolvimento de aplicações baseados em *frameworks*

3.2 Processo de desenvolvimento

O processo de desenvolvimento de um *framework* depende do grau de experiência da organização no domínio do problema. Uma organização com maior experiência poderá adotar um processo mais avançado no desenvolvimento. Em seguida, serão apresentados alguns processos propostos para o desenvolvimento de um *framework* [Mattsson 1999, 2000].

3.2.1 Processo convencional para o desenvolvimento de *frameworks*

O elementos comuns aos processos de desenvolvimento são:

Análise do domínio do problema;

A primeira versão do *framework* é desenvolvida utilizando-se as abstrações encontradas;

Uma ou mais aplicações são desenvolvidas baseadas no *framework*. O teste é importante para verificar se o *framework* é realmente reutilizável;

Problemas encontrados no desenvolvimento da aplicação baseado em *framework* são capturados e resolvidos na sua próxima versão;

Após repetir o ciclo várias vezes, o *framework* vai atingindo um nível de maturidade aceitável, possibilitando, desta forma, a sua reutilização por vários usuários.

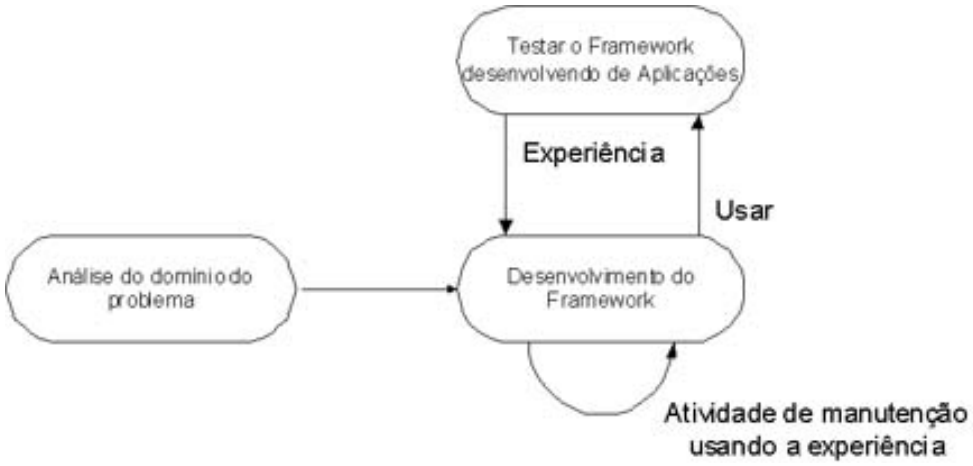

Figura 3.2 - Processo geral de desenvolvimento de *frameworks*

3.2.2 Processo baseado em experiências de aplicações já desenvolvidas

Inicia-se com o desenvolvimento de n aplicações (mínimo de 2) baseadas no domínio do problema. Quando prontas, observam-se as características em comum das aplicações, que são colocadas no *framework*. Para verificar se as características

extraídas estão corretas, refazem-se as aplicações com base no *framework* desenvolvido. No caso de demandar muito esforço, reescreve-se o *framework*, quando necessário, e utiliza-se essa experiência no desenvolvimento da nova versão, desenvolvendo-se outras aplicações com o *framework*, repetindo-se as iterações quantas vezes forem necessárias. Veja o esquema na figura 3.3:

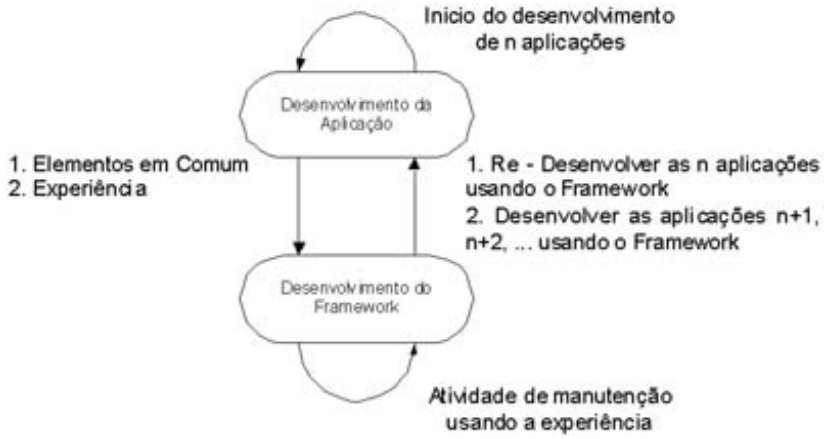

Figura 3.3 - Processo de desenvolvimento de *frameworks* baseado em *viewpoints*

[Ainsworth ett alli 1994] [Filkelstein ett alli 1992] [ITU 1995]

3.2.3 Processo baseado na análise do domínio

A primeira etapa é analisar o domínio do problema para identificar e entender possíveis abstrações. Analisar o domínio requer a análise de aplicações existentes, e isto só é possível se a organização já possuir aplicações desenvolvidas. Após identificar-se as abstrações, desenvolve-se o *framework* juntamente com uma aplicação de teste, modificando o *framework* quando necessário, revisando as aplicações anteriores para verificar se continuam funcionando.

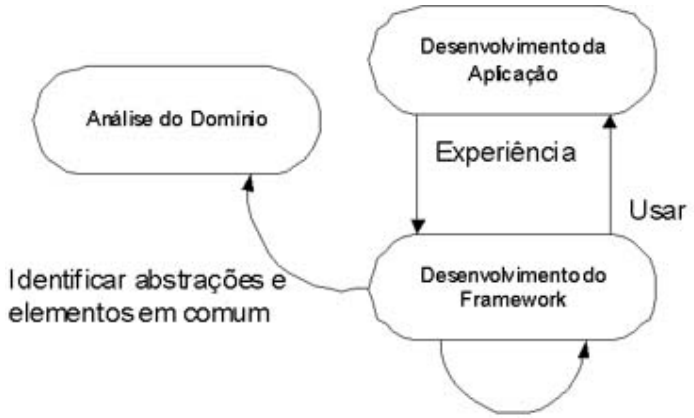

Figura 3.4 - Processo de desenvolvimento de *frameworks* baseado em análise de domínio

3.2.4 Processo de desenvolvimento utilizando design patterns

O primeiro passo consiste em desenvolver uma aplicação e, em seguida, aplicar sistematicamente um conjunto de *design patterns* para criar o *framework*. A partir desse momento ocorrem as interações entre as aplicações e o *framework*.

3.2.5 Processo baseado em componentes

Em Fayad [Fayad1999b], o processo de desenvolvimento de *frameworks* tem uma particular interseção com a palavra reutilização. Para que uma reutilização possa ser atingida no maior grau possível, um *framework* consiste de:

- Um repositório de *patterns* estruturais ou estilos formalmente definidos;
- Um repositório de componentes;
- Regras para instanciar *patterns* selecionados a partir do repositório de *patterns* estruturais com componentes selecionados do repositório de componentes. Ou seja, as regras servem como mecanismos no auxílio à instanciação de *patterns* estruturais utilizando-se os componentes já existentes em outro repositório (repositório de componentes).

Esta metodologia oferece reutilização estrutural em três níveis de abstração e utiliza diferentes mecanismos.

O nível mais alto de abstração é chamado de topológico. Aqui, o a maior parte da organização de um sistema é especificado em termos de reutilização em alto nível, por meio de abstrações utilizando-se grafos teóricos chamados de estilos. Neste nível, utiliza-se, basicamente, o repositório de *patterns* estruturais para a especificação do sistema em sua primeira instância. Um estilo pode especificar um *pattern* genérico como, por exemplo, para uma configuração de uma rede com topologia serial, paralela ou estrela, de um conjunto de componentes. Componentes são representados por nodos e as conexões por setas. As funcionalidades ainda não são especificadas neste momento;

O próximo nível é o arquitetural, no qual mais detalhes são adicionados tais como: definição de interfaces para os componentes e definições de conexão entre as várias portas das interfaces. A reutilização nesta fase é feita pela parametrização da arquitetura (podemos fazer uma associação direta com componentes tipo *black-box*).

O último nível de abstração é chamado de nível de sistema, no qual as funcionalidades dos componentes são concretizadas utilizando-se alguma linguagem de programação.

Nesta abordagem para desenvolvimento de aplicações baseadas em *frameworks*, a reutilização de componentes é a palavra chave e podemos identificar várias formas de reutilização.

A figura 3.5 mostra a reutilização de um componente por várias estruturas abstratas. No processo de instanciação destas estruturas ou estilos, um determinado componente é utilizado de forma a contemplar os requisitos já selecionados.

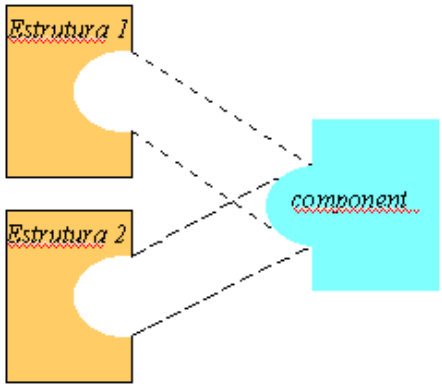

Figura 3.5 - Reutilização de componentes

Na figura 3.6, o que se apresenta é a reutilização extensiva, na qual uma estrutura abstrata (*pattern*) faz uso de outras estruturas (*patterns*). Esta é uma reutilização de alto nível, sendo que, após esta etapa, faz-se também necessário que esta nova estrutura como um todo seja especializada ou instanciada baseando-se nas funcionalidades que possui e, para isso, pode-se utilizar algum componente do repositório de componentes.

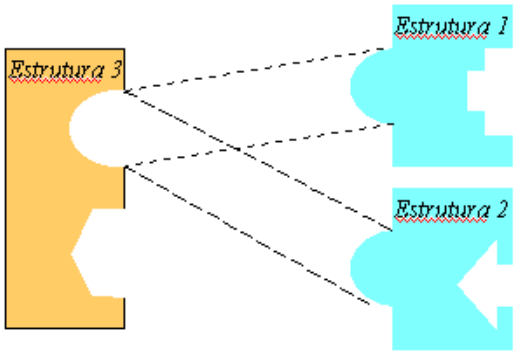

Figura 3.6 - Extensa reutilização estrutural

Na figura 3.7, o reuso ocorre em escala intencional. É intencional se um *pattern* genérico é instanciado em um *pattern* específico, ou um determinado *pattern* fixo é instanciado com os componentes concretos formais para produzir o sistema desejado.

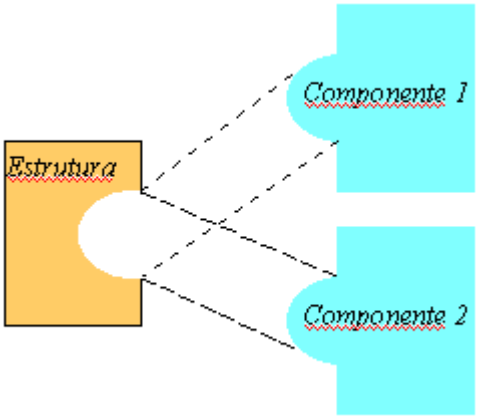

Figura 3.7 - Reuso estrutural intensional

Em resumo, para se gerar um sistema específico, o usuário primeiro seleciona um estilo dentro do repositório de estilos, ou *patterns* para uma aplicação específica. Se o estilo selecionado for abstrato, ele deve ser, então, instanciado dentro de uma topologia em particular, após o que, o resultado é mapeado dentro de alguma arquitetura. Para lograr este resultado, o usuário inicialmente define ou seleciona um conjunto de regras que é oferecido dentro do repositório de estilos. A etapa seguinte é a transformação da arquitetura gerada para uma notação apropriada em alguma linguagem de programação, obtendo os componentes específicos do repositório de componentes e interconectando-os de acordo com a arquitetura previamente gerada.

3.3 O papel de *roles* na caracterização de *frameworks*

Segundo Jan Bosch, linguagens orientadas a objetos convencionais não permitem que comportamentos sejam facilmente adicionados ou removidos em classes, mesmo quando mecanismos de heranças são utilizados. Por sua vez, *design patterns* e *frameworks* também carecem de mecanismos mais elaborados de descrição de comportamento para classes que necessitam ser instanciadas.

3.3.1 Processo baseado em fragmentos arquiteturais.

Em [Bosch 1998] [Bosch 1999b], o processo de desenvolvimento de *frameworks* está centrado na utilização de fragmentos arquiteturais.

A solução apresentado por Bosh em [Bosch1998] é a utilização de fragmentos arquiteturais. Um fragmento arquitetural descreve, para um conjunto de componentes que executam um determinado papel ("*role*") em um fragmento, a parte do

comportamento específico para uma arquitetura desejada. Um fragmento consiste de um conjunto de papéis e códigos que são executados no momento da instanciação do fragmento. Um *role* é especificado como uma classe, **mas ele oferece maior expressividade na definição dos requisitos de interface**, de forma que outra classe possa mais facilmente realizar algum relacionamento.

A idéia básica é a de que todos os comportamentos de interfaces das classes sejam realizados utilizando-se *roles*. Desta forma, podemos ter vários *roles* para uma mesma classe enfocando aspectos distintos de comportamento, permitindo que as classes cooperem entre si mais facilmente.

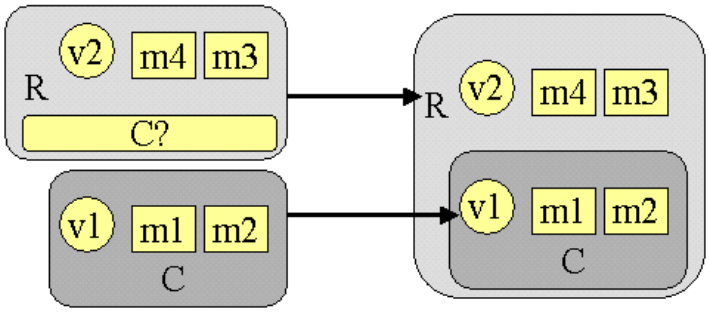

Figura 3.8 - Composição de um *role* com um componente

A figura 3.8 ilustra a composição de um *role* com um componente. O componente C consiste de uma variável de instância chamada v1 e métodos chamados m1 e m2. O papel R possui uma variável de instância chamada v2 juntamente com métodos m3 e m4. O resultado da composição pode ser visto na figura 3.8. A técnica de composição utilizada por Bosch chama-se *Superimposition*[Bosch 1999b][Bouge & Francez 1988].

Um fragmento arquitetural, ou simplesmente fragmento, descreve dois ou mais papéis e suas interações. *Superimposition* permite que composições possam ser realizadas entre fragmentos e outras classes. Como suporte ao mecanismo de *Superimposition*, uma linguagem chamada *LayOm* [Bosch 1996] é utilizada de forma a gerar uma saída em C++ das especificações de comportamento criadas por meio de *roles*. Um objeto *LayOM* contém, como qualquer outro modelo de objeto, variáveis de instância e métodos.

Quando um objeto é especificado utilizando a linguagem *LayOM*, é formada uma barreira para interceptar qualquer evento que chegue ou saia de uma classe.

Como uma classe é a composição da classe propriamente dita e seus *roles*, estes fazem o papel de uma interface principal ou filtro gerenciando as mensagens

recebidas ou retornadas pelos métodos originais das classes envolvidas. A figura 3.9 exemplifica o que foi dito.

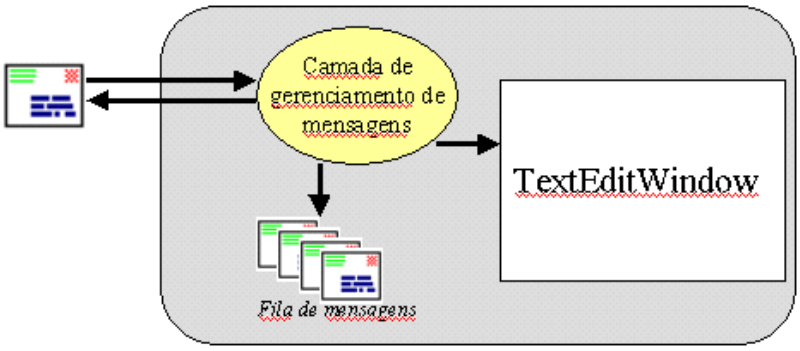

Figura 3.9 - Camada de restrição

No exemplo da figura 3.9, qualquer mensagem que for passada ao objeto *TextEditWindow* é previamente interceptada por um *role* que verifica e decide se encaminha a mensagem ao objeto, ou se põe em alguma fila de mensagens, ou simplesmente a rejeita. Da mesma forma, o *role* intercepta a mensagem de retorno do objeto *TextEditWindow* de forma a transformar essa saída, caso seja necessário.

3.3.2 Processo baseado em *Roles*.

Segundo Dirk Riehle em [Riehle,Gross 1998], o conceito de *frameworks* está centrado no desenvolvimento de sistemas orientados a objetos com grandes modelos de classes/objetos. Este enfoque promove reuso, desenvolvimento rápido e aplicações com maior qualidade. Porém, esta forma de se desenvolver sistemas de *software* ainda não está totalmente madura. **Em particular, não existem teorias convergentes e consolidadas para o desenvolvimento de *frameworks* e sua integração**.

A primeira razão para todos esses problemas de *design* e integração de *frameworks* é a sua tradicional abordagem baseada nas técnicas de orientação a objetos, ou seja, baseado em um forte modelo de classes. Classes são excelentes para a descrição de conceitos e abstrações de um domínio de aplicação, porém mostram-se fracas quando necessitamos descrever comportamento e colaboração entre objetos [Helm 1990].

Quando um *framework* é projetado, o desenvolvedor deve clarificar as responsabilidades que os objetos possuem, suas dependências nos vários contextos e como os objetos se relacionam entre os contextos. Ele também deve definir os comportamentos e colaborações e como eles podem ser compostos. Estas

considerações não podem ser facilmente obtidas utilizando-se somente classes e seus artifícios de heranças.

A idéia de modelagem baseada em *roles* não é nova. A metodologia OOram, desenvolvida por Reenskaug [Reenskaug1996], apresenta uma abordagem para modelagem de objetos, seus comportamentos e colaborações. Entretanto, esta metodologia enfatiza somente objetos e não oferece suporte para *frameworks*.

Utilizando-se *roles*, uma classe representa um domínio abstrato no qual é possível representar suas abstrações intrínsecas por meio de técnicas comuns de orientação a objetos como heranças, polimorfismos e seus comportamentos por meio de *roles*.

Um modelo de *role* é a descrição de um conjunto de propriedades de comportamento e colaborações entre objetos usando *roles*. A diferença fundamental entre um modelo de classes e um modelo de classes usando *roles* reside justamente nos conceitos representados pelas classes. Restrições também são enfatizadas nos modelos baseados em *roles.*

A figura 3.10 exemplifica o uso de *roles*.

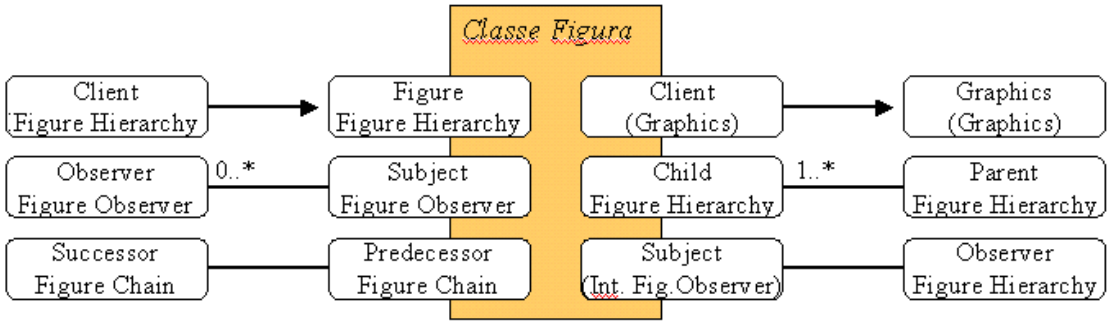

Figura 3.10 - Classe Figura e seus vários papéis (*roles*)

[Riehle, Gross 1998]

Na figura 3.10, os vários tipos de *roles* estão visíveis no diagrama dentro das caixinhas. Na classe *Figura*, podemos identificar vários papéis a ela associados. Estes papéis fazem parte de um espaço de nomes no qual o *role* é identificado pelo seu tipo que preenche todos os seus requisitos e comportamentos. Como exemplo, podemos identificar o *role Figure* com seu tipo *Figure Hierarchy*. Cada *role* possui um tipo que o identifica, e este tipo é único dentro deste espaço de nomes. Com vistas a minimizar conflitos de nomes, utiliza-se a seguinte notação para identificar o tipo de um *role*: (RoleModel.RoleType)

Então, para o tipo *Figure Hierarchy* do *role Figure* podemos escrevê-lo como:

FigureHierarchy.Client: isto indica que o tipo *Figura Hierarchy* do papel *Figure* possui um relacionamento com o papel *Client*, da mesma forma que o tipo *Graphics* do papel *Graphics* possui um relacionamento com o papel *Cliente* (pertencente a classe *Figure*).

Modelos baseados em *roles* fornecem excelente *separation of concerns*[2] na medida em que o ponto de convergência é feito sobre um *role* em particular, dos vários que podem existir para uma determinada classe.

Neste ponto, fica clara a diferença entre um modelo baseado em classes e um modelo baseado em classes utilizando *roles*. A figura 3.11 mostra um exemplo.

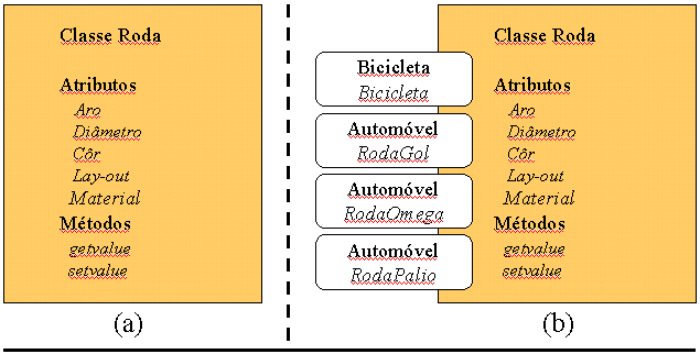

Figura 3.11 - Modelo de classes e classes com *roles*

Na figura 3.11 (a), podemos ver uma classe com seu vários atributos que a descrevem e seus métodos. Na figura 3.11 (b), a mesma classe é adornada com vários *roles* que descrevem, além de seus atributos, todo um conjunto de funcionalidades ou comportamentos que não pode ser encontrado na figura 3.11(a). Para que a figura 3.11(a) pudesse ter todo esse conjunto de comportamentos ou colaborações, deveríamos utilizar artifícios como herança múltipla para tentar obter alguma semelhança. **Classes e objetos são dispensáveis na caracterização de um *framework*; *role* é um conceito essencial.**

3.4 *Frameworks* e plataforma Web

No capítulo 2, discutiu-se a composição de *frameworks* e apresentou-se uma introdução aos conceitos básicos sobre *frameworks* que incluem: conceituação,

[2] Separation of Concerns é um conceito central em engenharia de software. Ele trata da habilidade de identificar, encapsular e manipular partes relavantes de um software para um objetivo particular. Este conceito permite que softwares possam ser organizados e decompostos em partes gerenciáveis e de fácil manipulação.

classificação, papéis dos diversos usuários, arquitetura básica, pontos de flexibilização e comparação com outras abordagens. Neste capítulo, nas seções 3.2 e 3.3 foram apresentadas metodologias para o desenvolvimento de *frameworks* na ótica de orientação a objetos e enfatizando o aspecto essencial de *roles*.

Esta nova seção mostrará como podemos trabalhar o conceito de *frameworks* por uma outra ótica, a plataforma Web. Esta plataforma remete-nos a novos problemas que são decorrentes da sua não permanente sincronização de serviços e aplicações. Neste escopo, um sistema de *software* pode estar totalmente fragmentado e ser composto por vários subsistemas elaborados em diversas linguagens de programação. Outro ponto que merece bastante destaque é que, diferentemente do mundo de objetos, na plataforma Web, muito das aplicações são desenvolvidas utilizando-se linguagens de *script* tais como: *Perl*, *Phyton*, *TCL*, JavaScript, *CGI-Lua*, *Activex*, e outras. Em virtude disto, outro problema apresenta-se que é o de adaptar o conceito de *framework* para linguagens não orientadas a objetos.

Aplicações com característica cliente-servidor são encontradas, em sua maioria, na plataforma Web e, com isso, uma série de problemas decorrentes deste tipo de tecnologia soma-se aos já discutidos anteriormente para criação e composição de *frameworks*.

3.4.1 Caracterização preliminar de um *framework* em plataforma Web.

A seguir, serão descritos os principais conceitos de *frameworks* utilizados dentro da ótica de plataforma web.

- *Kernel*: um conjunto de serviços básicos na *Internet* e que estejam presentes em aplicações baseadas na Web;

Um conjunto de *roles* que permita aos serviços serem estendidos de forma a gerar novas aplicações;

- *hot-spots*: os pontos a serem flexibilizados não necessitam estar explicitamente definidos na implementação dos serviços. À medida que um novo serviço seja necessário, um *hot-spot* é definido por meio de mecanismos de *roles*. Um serviço já existente passa a implementar este *hot-spot* com uma camada extra de requisitos particulares presentes no *role*; ou um novo serviço é criado e o *role* faz um *glue* da aplicação com este novo serviço; ou um novo serviço é criado para cooperar com outro já existente e, neste caso, o *role* faz a interface entre os dois serviços. É importante frisar que os serviços já existentes não necessariamente foram

projetados para serem compostos ou permitirem cooperação com outros serviços. Nestes casos, um *role* efetua esta tarefa.

- Um serviço dentro da Web é composto por um ou vários *scripts* que realizam uma determinada tarefa a partir do *browser*. Geralmente serviços na Web podem ser vistos como caixas pretas onde os parâmetros passados pelo *browser* são decodificados por um *script* no Servidor WWW da aplicação. Um mesmo *script* pode ser acionado por uma grande variedade de aplicações diferentes e de forma assíncrona. *Scripts* que produzem contagens de *click* em *banners* de anúncios, por exemplo, são acionados por muitas aplicações que contratam tais serviços para a contagem de seus acessos ou *hits*.

Uma aplicação na Web requer que todos os seus serviços estejam disponíveis em servidores de *CGI*. Para que um determinado problema possa ser solucionado, uma coleção de serviços é utilizada neste caso. A seguir, basearemos nossa discussão em um exemplo de uma aplicação que permite que usuários de uma determinada máquina UNIX possam ter acesso a seu *e-mail*.

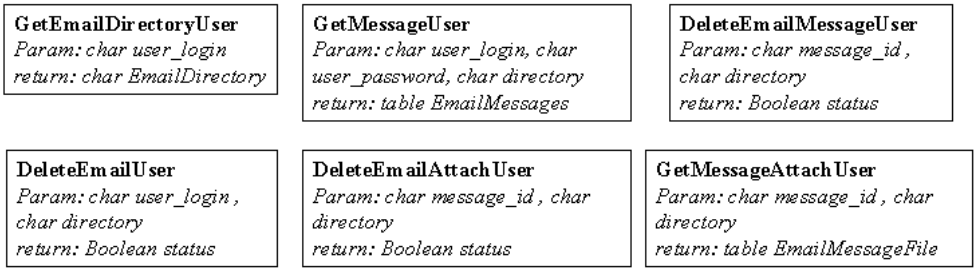

Figura 3.12 - Conjunto de serviços localizados em um servidor *X*

Na figura 3.12, podemos ver o conjunto de *scripts* que permite criar uma aplicação para receber *e-mail*. A aplicação por hora batizada de *WebMail* necessita dos seguintes serviços apresentados na figura 3.12:

Serviço	Descrição
GetEmaildirectoryUser:	Este *script* permite identificar onde está localizado o arquivo em que as mensagens de um determinado usuário estão localizadas. Neste ambiente, todos os *e-mails* ficam disponíveis em arquivos de formato texto. Estes arquivos podem residir em diretórios específicos para cada usuário.
GetMessageUser:	Este *script* realiza uma pesquisa em um arquivo texto e formata todos os *e-mails* recebidos por um usuário em

	uma tabela de mensagens.
GetMessageAttachUser:	Este *script* identifica todos os arquivos anexados *aos e-mails* enviados a um determinado usuário.
DeleteEmailMessageUser:	Este *script* permite excluir um determinado *e-mail* de uma lista de *e-mails* do usuário.
DeleteEmailAttachUser:	Este *script* remove todos os arquivos que estão incorporados a uma determinada mensagem.
DeleteEmailUser:	Este *script* remove a caixa postal de um determinado usuário.

Tabela 3.1 – *Scripts* do *WebMail*

Estes serviços, uma vez relacionados, permitem que diversos tipos de aplicações para gerenciamento de mensagens possam ser desenvolvidos. Por exemplo:

Vamos utilizar este conjunto de serviços para gerar dois tipos distintos de aplicações para leitura de *e-mail*. A figura, abaixo, mostra uma primeira versão de uma aplicação que permite a um usuário ler suas mensagens (somente ler e excluir *e-mails* recebidos).

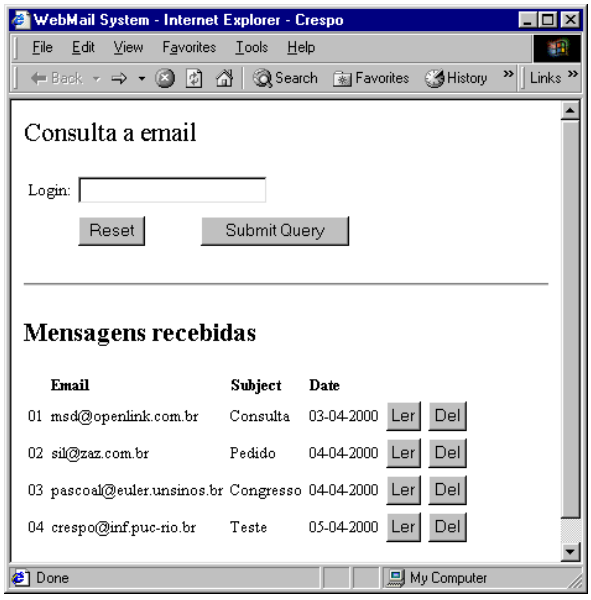

Figura 3.13 - WebMail

Neste primeiro exemplo, vários serviços foram utilizados para que a aplicação pudesse ser criada. Os serviços utilizados são:

Para leitura e apresentação das mensagens recebidas por um determinado usuário:

- GetEmailDirectoryUser

- GetMessageUser
- GetMessageAttachUser(para a exclusão das mensagens já recebidas e ou lidas por um usuário).

DeleteEmailMessageUser

DeleteEmailAttachUser

Os serviços acima descritos são básicos para uma grande variedade de aplicações de sistemas de recebimento de mensagens. Estes serviços podem ser o *Kernel* do *framework* para aplicações neste domínio.

Agora veremos uma variação desta aplicação utilizando outros serviços que não estavam presentes neste servidor *X*. A figura, abaixo, mostra alguns serviços presentes em outro servidor, o servidor *Y*.

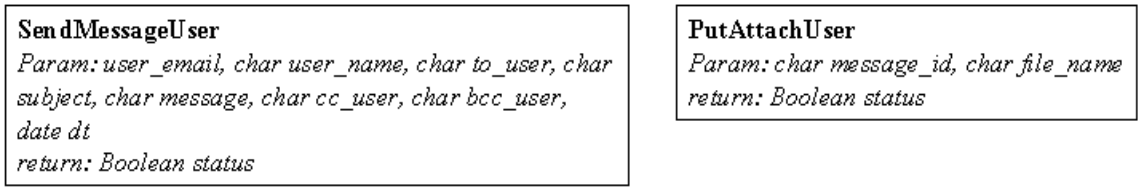

Figura 3.14 - Conjunto de servidor localizado em um servidor *Y*

Serviço	Descrição
SendmessageUser:	Este *script* recebe de um formulário dados para serem passados como parâmetros a um sistema de *e-mail* e, desta forma, enviar a mensagem. Este *script* chama outro *script,* PutAttachUser, para complementar sua tarefa.
PutAttachUser:	Este *script* trabalha junto com o SendMessageUser e implementa um file *upload* para que arquivos possam ser anexados a mensagens por *e-mail*.

Tabela 3.2 – *Scripts* do servidor Y

A idéia agora é estender a aplicação inicialmente oferecida para a recepção de mensagens e procurar permitir que o usuário possa também enviar um *e-mail*. Porém, esta nova necessidade nunca foi antes planejada. A estratégia neste caso é utilizar um serviço que já existe em outro servidor utilizando-se de dois *scripts* que permitem que uma mensagem seja enviada junto com um arquivo anexado.

Figura 3.15 - WebMail com a opção de enviar uma mensagem

O *hot spot*, neste caso, foi a inserção de mais dois serviços(*SendMessageUser* e *PutAttachUser*), para o envio de mensagens. Estes serviços não estavam disponíveis no servidor *X*, porém no servidor *Y* eles já existiam. E desta forma, foi possível a sua utilização sem que fosse necessário modificar os *scripts* existentes no servidor *X*.

Para permitir que outros *scripts* possam ser utilizados junto aos já projetados, é necessário que uma mensagem possa ser enviada através de um outro sistema. Neste caso, o *role* utilizado deve possuir as seguintes informações:

- Nome do Servidor que possui o serviço de enviar mensagens
- *Path* do diretório
- Nome do *script*
- Method [Get | Post]

A figura 3.16, abaixo, exemplifica o que foi dito:

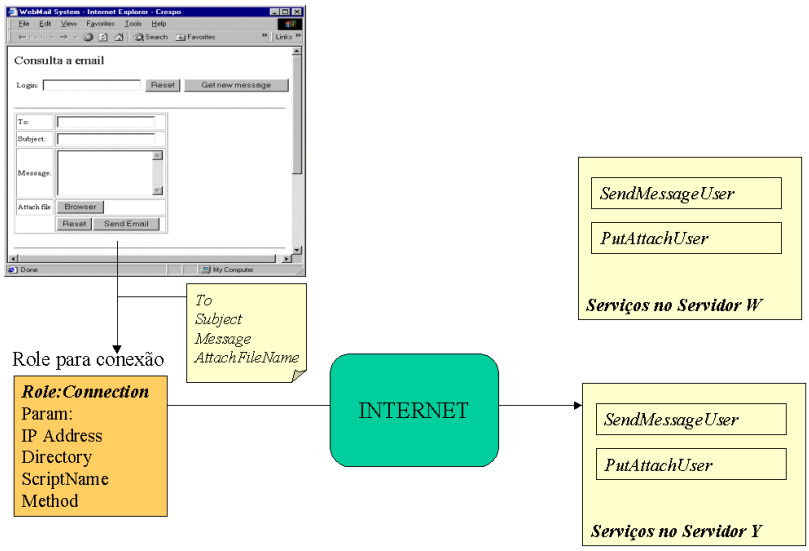

Figura 3.16 - *Roles* na nova versão do sistema de leitura de mensagens

Com a utilização de um *role* que implementa o *hot-spot*, podemos deixar flexível o local onde está o servidor que contém o serviço necessário para o envio de mensagens, bem como algumas informações da interface de destino para que os dados enviados possam ser bem processados.

Uma vez verificado o que caracteriza um *framework* em plataforma Web, proporemos um novo conceito para *frameworks* que atuam nesta plataforma. Este conceito chama-se: *WebFramework*.

3.5 *WebFramework*: *frameworks* baseados em serviços disponíveis na Web.

A Web é composta por uma imensidão de servidores conectados entre si e que fornecem grande quantidade de informação por serviços.

Um *WebFramework* pode ser visto como um conjunto de serviços que permite colaboração entre si e composição por meio de *roles*. As composições dos serviços definirão o perfil de uma aplicação a ser gerada. Um serviço poderá ter vários *roles*, permitindo visões diferenciadas a partir de diversas aplicações.

O relacionamento entre os vários serviços encontrados na Web serão realizados por *roles* que fazem o papel de uma interface para a colaboração dos serviços. A composição de serviços também segue esta regra, ou seja, quando um serviço necessita de vários outros para cumprir suas obrigações, torna-se necessária a definição de um *role* que componha esses serviços, permitindo que as interfaces envolvidas possam cooperar livremente e sem alterações em suas implementações originais. As figuras 3.17 e 3.18, a seguir, exemplificam o que foi dito.

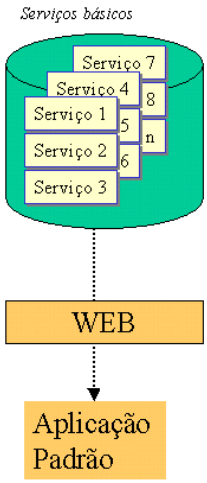

Figura 3.17 - Aplicação Padrão

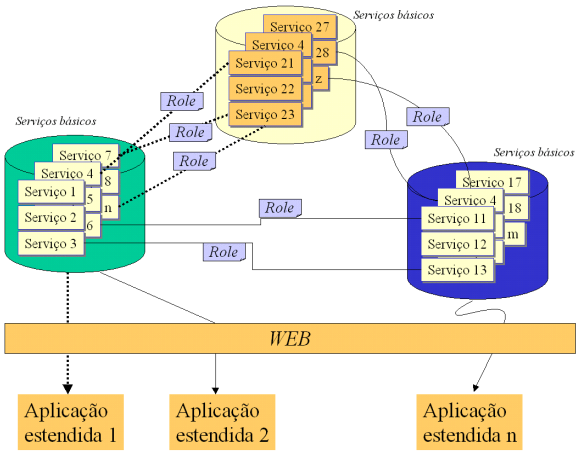

Figura 3.18 - WebFramework em plataforma Web

Uma aplicação padrão (figura 3.17) é um sistema que utiliza um conjunto de serviços para atingir uma determinada obrigação.

A instância do *WebFramework* representada pela figura 3.18 são todas as aplicações estendidas resultantes da composição e ou utilização de serviços na Web para a geração de diferentes tipos de sistemas.

Características de um *WebFramework*:

(1) O *Kernel* de um *WebFramework* é formado por um conjunto básico de serviços. Estes serviços podem pertencer a um mesmo domínio, como no exemplo anterior do sistema de *e-mail*, ou não. A idéia principal do *kernel* é a existência de um conjunto básico de serviços que permitam a construção de uma aplicação na Web.

(2) *Hot-spot*: a flexibilização em *WebFrameworks* tem uma peculiaridade típica comum a todos os sistemas da Web: a não obrigatoriedade da presença física em *design* de sistemas. O desenvolvimento de *software* em plataforma Web tem um caráter predominantemente *BOTTOM UP*, ou seja, são construídos em pequenos módulos e depois distribuídos dentro de páginas *HTML* para a formação de sistemas mais complexos. Isto se deve à fácil manipulação de linguagens não compiladas pelos diversos *browsers* existentes. Assim, mudanças são facilmente feitas em tempo de execução e, toda vez que o browser for carregado, um novo item pode estar presente na aplicação do usuário.

(3) A instanciação de novas aplicações a partir de um conjunto básico de serviços não precisa, *a priori*, gerar sistemas para um domínio específico, como no exemplo anterior. Como os serviços na Web são do tipo *black-box*, podemos utilizar tais

serviços sem que estes causem impactos em outros serviços, uma vez que, geralmente, estes trabalham de forma autônoma. Impactos podem ocorrer no momento em que uma determinada tarefa requer que serviços sejam compostos ou cooperem entre si por meio do envio de dados para que sejam processados por outros serviços. No exemplo anterior do *WebMail*, a nova instância gerada por meio da utilização do *kernel* existente no Servidor *X* previa a possibilidade de que o usuário também pudesse enviar uma mensagem. Foi, então, utilizado um serviço externo a este servidor que necessitava de um *role* que permitisse a passagem dos dados ao *script* de destino, bem como identificar a forma como estes dados devem ser passados e a sua localização na Web (uma vez que os servidores e *WebFrameworks* podem ser ativos e passivos, *paths* são freqüentemente alterados).

(4) *WebFrameworks* ativos: São caracterizados pela presença de componentes que não necessitam de formulários para sua ativação. Exemplos destes componentes são *applets Java* que, uma vez carregados pelo *browser,* executam um determinado serviço independentemente de uma ação externa tal como um clique de *mouse*. Sistemas de captura de informações em servidores *Push* são um bom exemplo disto.

(5) *WebFrameworks* passivos: São caracterizados pela necessidade de uma ação externa para que um serviço seja executado, como um clique de *mouse*. Um exemplo claro são os formulários contidos em páginas *HTML* para a emissão de *e-mail.* Para que os dados do formulário possam ser processados por um *script,* que está no servidor da aplicação, é necessário um estímulo externo que é o clique de *mouse* no botão que aciona a transmissão dos dados do formulário para o servidor de *CGI*.

3.5.1 Formas de utilização do *WebFramework*:

Um *WebFramework* pode ser utilizado para gerar novas aplicações utilizando-se tanto o conceito de *black-box* como de *white-box* e também através de uma abordagem híbrida (que seria uma mescla de *black-box* com *white-box*).

3.5.1.1 *WebFramework Black-Box*:

Nesta forma de uso, uma nova aplicação é gerada a partir da aplicação inicial, na qual todas as formas de adaptação já foram projetadas no *design* inicial da aplicação. Desta forma, uma nova aplicação pode ser definida pela entrada de dados (ou parâmetros) na área de configuração da aplicação inicial. Com os dados sendo inseridos nos lugares de dados *default*, uma nova aplicação é produzida. Os serviços utilizados

neste caso já estão definidos em sua maioria, assim como os *roles*, pois estes fazem o papel dos configuradores que podem estar sob a forma de *cookies* na máquina do usuário ou sobre a forma de *scripts* no servidor. Estes roles são carregados a partir do *login* do usuário para leitura e adequação da interface da nova aplicação. Adaptação com esta abordagem gera sistemas que são um subconjunto do sistema original preservando suas principais características e serviços. Esta forma de adaptação tende a ser mais fácil, porém o usuário fica um pouco restrito aos vários padrões já definidos para a configuração dos pontos flexíveis. A figura 3.19, abaixo, mostra um exemplo disto.

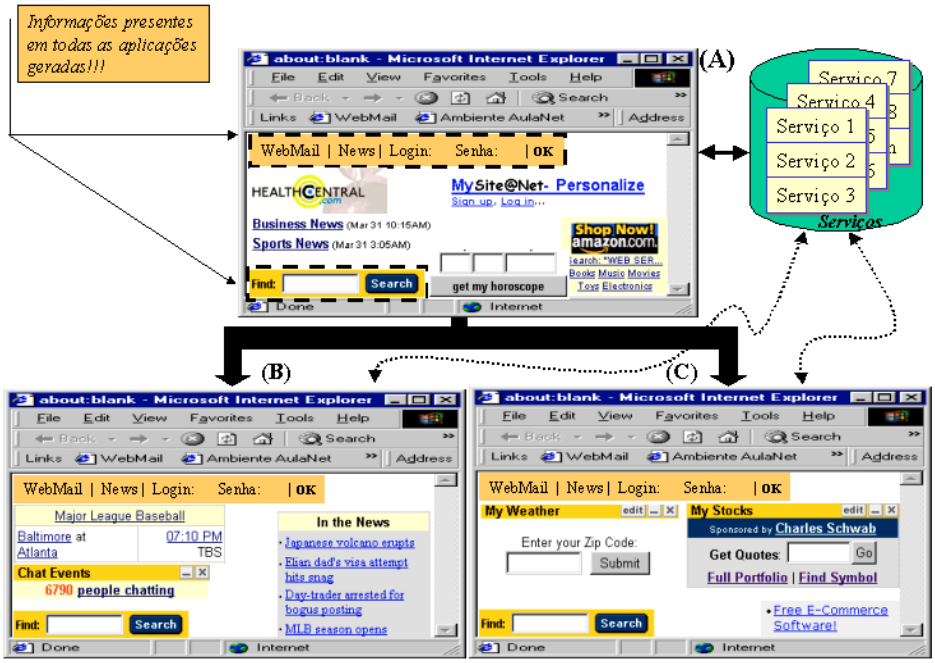

Figura 3.19 - Customização tipo *Black-Box*

A figura 3.19 mostra que, a partir da aplicação (A), pode-se configurar outras duas aplicações que são um subconjunto desta aplicação. As aplicações (B) e (C), apesar de possuírem um *layout* diferente da (A), preservam alguns serviços básicos e atuam na mesma área de provisão de informação que a aplicação (A).

A aplicação (A) foi projetada com uma série de pontos a serem flexibilizados por um usuário caso este assim o deseje. Para cada *hot-spot* disponível na aplicação (A), um conjunto de prováveis soluções foi disponibilizado no qual o usuário apenas informa qual das opções é a desejada e quais são os parâmetros de entrada necessários. Estes parâmetros podem ser: seu *e-mail*, para que o sistema verifique se ele possui novas mensagens, um certo endereço de provedor de informações financeiras para a cotação de moedas no mercado internacional, etc. Pode-se ou não, dependendo de como foi

projetado, ser feita a utilização de *roles* já implementados para a cooperação de tais serviços ou até mesmo a sua composição. No sistema de *WebMail,* por exemplo, quando o sistema identifica que um usuário possui uma mensagem, este utiliza um outro serviço que busca por arquivos anexados a esta mensagem. Este serviço de busca por arquivos anexados, *GetMessageAttachUser,* poderia ter sido desenvolvido posteriormente e depois composto com o serviço de *GetMessageUser* por meio de um *role* como uma nova forma de mostrar as mensagens recebidas.

Em resumo, a aplicação (A) foi projetada como um *WebFramework* tipo *black-box*.

3.5.1.2 *WebFramework White-Box*:

Com esta ênfase, a adaptação para a geração de novas aplicações requer que o projetista procure por novos serviços e implemente como este novo serviço integrar-se-á ao conjunto de serviços básicos, para a geração de uma nova aplicação. Neste caso, o usuário não tem a liberdade de definir os componentes e configurá-los.

Esta abordagem proporciona um maior grau de especialização no sentido que o novo serviço será melhor especificado e implementado, sendo feito sob medida.

A figura 3.20 mostra um exemplo do que foi dito.

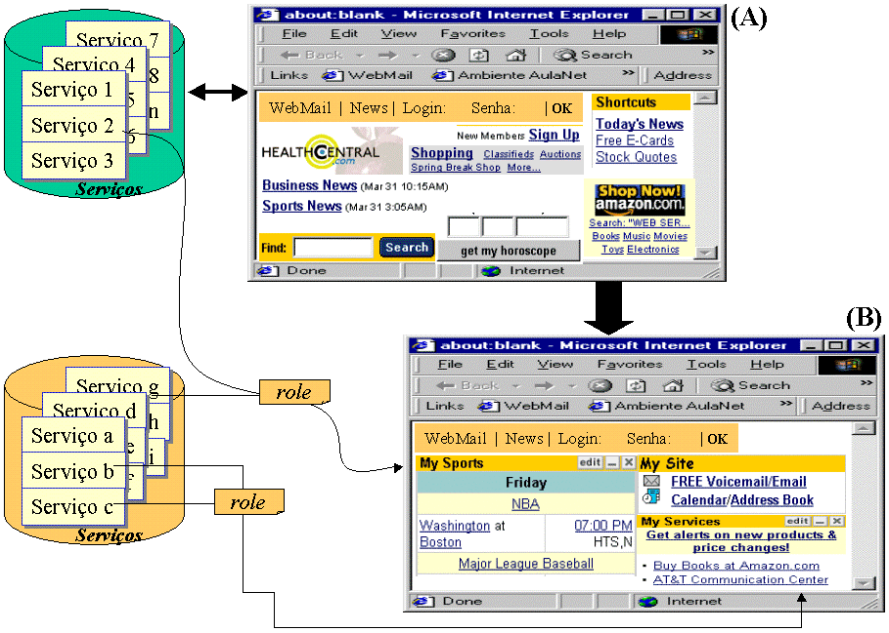

Figura 3.20 - Customização tipo *White-Box*

Na figura 3.20, podemos ver um conjunto de serviços distribuídos geograficamente em servidores distintos. Neste exemplo o único serviço que ficou presente na aplicação (B) gerada a partir da aplicação (A) foi o serviço que permite a verificação de *e-mail* e *login* do usuário. A aplicação (B) utiliza alguns serviços da aplicação (A)

em conjunto com os serviços pertencentes a outro servidor (*Serviço 2* e *Servico G*). *Roles* são utilizados para que os serviços possam cooperar entre si. Como um exemplo, na aplicação (B) o item My Sports é o resultado da utilização de dois *scripts* que oferecem os seguintes serviços: *Serviço 2* (relação de notícias referentes aos esportes) e *Serviço G* (relação de todos os esportes que um determinado usuário gosta). Um *role* é utilizado para informar ao *Serviço 2* o resultado de uma *query* contendo os dados para que, durante sua busca, somente os esportes selecionados pelo usuário sejam consultados. O *Serviço 2* é genérico e o *role* faz a especialização deste definindo uma *query* adequada. Este *role* precisou ser explicitamente desenvolvido em virtude da aplicação (A) não prover nenhum mecanismo de adaptação estilo *black-box*, como no exemplo anterior. Desta forma, o projetista deve identificar a interface do *Serviço 2* e desenvolver um *role* que manipule esses dados e crie um canal comum entre os dois serviços.

A seguir, faremos uma comparação entre as duas abordagens: *frameworks* vs. *WebFrameworks.*

Comparação entre *frameworks* e *WebFramework*

	Frameworks	*WebFrameworks*
Plataforma	*Windows* e *Unix*	Somente Web
Linguagem de desenvolvimento	Geralmente em linguagens compiladas e orientadas a objetos tais como *C++* e *Java*	Geralmente em linguagens Scripts tais como *CGI-Lua*, *Python*, *Perl* e outras.
Tipos de servidores	Baseados em *Servlets*.	Baseados em códigos *CGI* interpretado.
Kernel	Conjunto de classes baseadas em orientação a objetos.	Conjunto básico de serviços presentes na Web. Um serviço pode ser implementado por um ou vários *scripts*.
hot-spots	Devem existir fisicamente no projeto do *framework*. São classes abstratas que necessitam ter seus métodos implementados ou por	Não precisam estar predefinidos no *design* da aplicação. Podem ser criados em qualquer momento.

	mecanismos de herança ou delegação.	
Custo de definição dos *hot-spots*	Alto: uma vez implementado o *framework*, os *hot-spots* já estão também definidos. Um acréscimo de novos *hot-spots* pode impactar muito as aplicações já geradas e ser necessário realizar um novo levantamento de requisitos.	O custo é baixo na medida em que a inclusão de novos *hot-spots* não impacta as aplicações geradas. Este baixo impacto deve-se, em primeira instância, à facilidade que aplicações Web possuem em alterar suas implementações, pois o *HTML* e os *scripts* são interpretados em sua maioria.
Metodologia de desenvolvimento	Geralmente *Top-down*	Geralmente *Bottom-Up* para a implementação dos serviços *e Top-Down* para a implementação da interface da aplicação.

Tabela 3.3 – Comparação: *frameworks* vs. *WebFrameworks*

A seguir, verificaremos os problemas levantados no Capítulo 2 sobre composição em *Framework*s que estão presentes em composição de *WebFrameworks*.

3.5.2 Composição de *WebFrameworks* em plataforma Web

Os problemas estudados serão divididos em dois grupos:

1- Problemas genéricos:

- Problema do controle do *loop* de execução;
- Composição com sistemas legados;
- Problemas de carência de requisitos;
- Superposição de perspectivas;

2- Problemas particulares da plataforma Web:

- Assincronismo de tarefas;
- Recuperação em caso de falhas;
- Trocas de mensagens entre serviços;

***WebFramework* : Problema do controle do loop de execução**

Este problema, tal qual descrito no Capítulo 2, torna-se existente para *WebFrameworks* ativos, ou seja, para aqueles que possuem algum componente ou serviço ativo. Este serviço caracteriza-se pela sua auto-execução e conexão com o seu servidor. Neste caso, não há necessidade de nenhum estímulo externo para realizar a sua tarefa. A figura 3.21, abaixo, exemplifica o que foi dito.

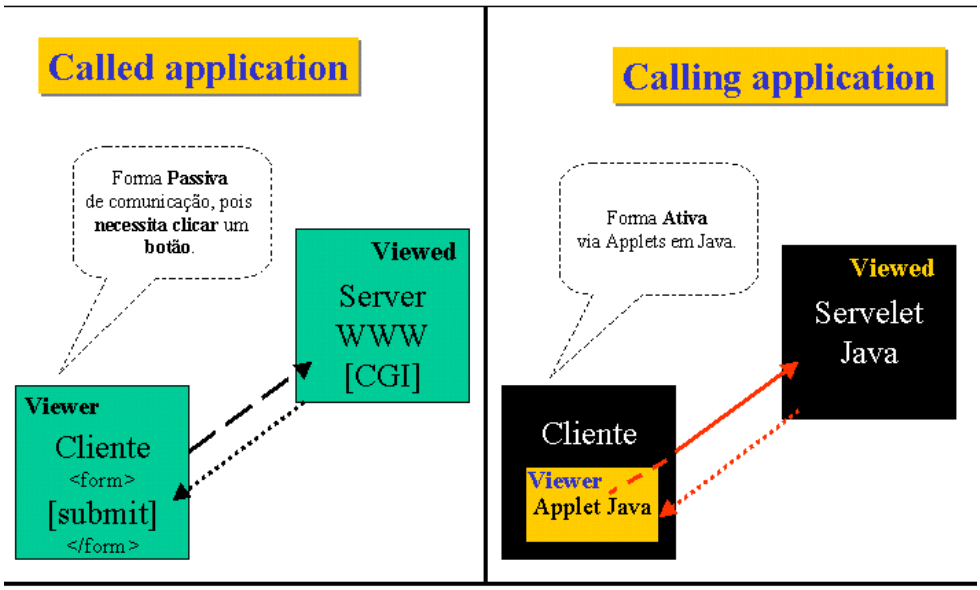

Figura 3.21 - *WebFrameworks* ativos e passivos

O problema de controle de execução também existe nesta plataforma, pois um *applet Java* pode encontrar conflitos não na execução do código fonte, mas sim na obtenção dos dados oriundos de *queries* que podem ser atualizadas, sem que o *applet* tenha sido notificado. Desta forma, uma informação pode estar sendo transmitida de forma errônea.

Também podemos ter aplicações que são ao mesmo tempo ativas e passivas, ou seja, uma abordagem híbrida.

A figura 3.22, abaixo, mostra um exemplo deste problema.

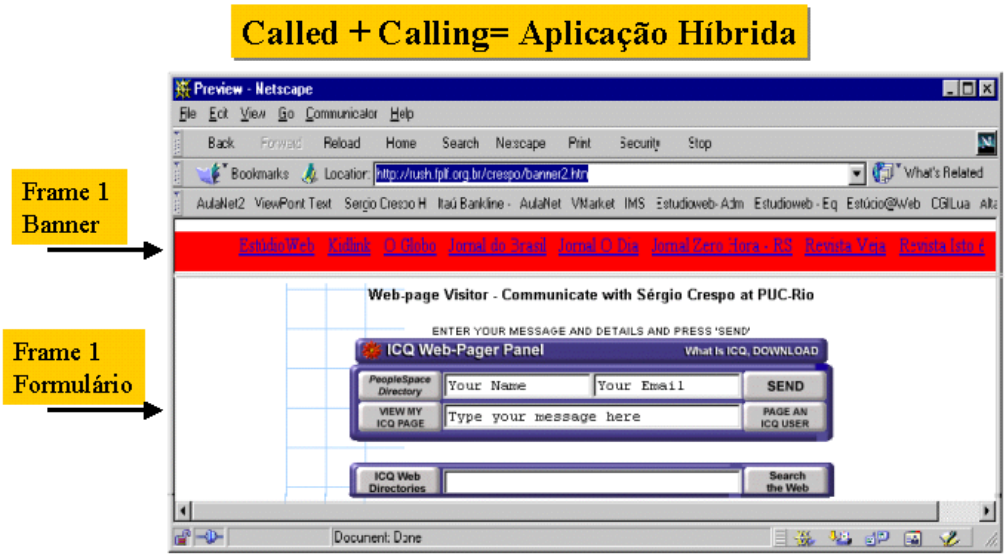

Figura 3.22 - *WebFramework* híbrido

Na figura 3.22, um *WebFramework* híbrido apresenta no *frame* superior um serviço de obtenção de mensagens *linkadas* oriundas de um servidor Web padrão. Este serviço seleciona por meio de uma *query* uma mensagem a ser postada como um *banner* e não faz nenhum controle da versão da informação. Uma alteração nesta tabela pode ocasionar uma mudança na informação a ser exibida na tela do usuário.

***WebFrameworks*: Composição com sistemas legados**

A composição com sistema legado na plataforma Web remete-nos ao problema sobre como um sistema que necessite de um novo requisito pode incorporá-lo em seu ambiente por meio de um sistema legado, estando este localizado em seu servidor ou remoto.

Se o serviço estiver localizado em seu servidor, esta tarefa tem o seu problema minimizado por ser possível obter maiores detalhes de como este serviço foi implementado. Porém, fica ainda em aberto a forma de se adaptar o serviço ou os serviços com os quais o sistema legado (implementado por um conjunto de serviços) possa cooperar com o sistema atual.

Se o sistema legado estiver em outro servidor remoto, fica a questão de assincronismo em aberto. Como saber se a informação recebida pelo sistema legado é confiável?

Este problema existe na medida em que o sistema legado tem seu acesso independente de outros sistemas e, para tanto, uma série de variáveis e tabelas podem ser atualizadas, sem que este tenha conhecimento de quem o está utilizando externamente como um complemento.

Por exemplo:

O ambiente VBookM@rket[Maia 2000] , que é uma instância do *framework* Vmarket[Ripper 1999], implementa um mercado virtual para a comercialização de livros utilizando agentes para a negociação. Com o decorrer da utilização do ambiente, ficou constatada a necessidade de adicionar um novo item em sua interface: pesquisa de preços. Este item foi solicitado pelos usuários na medida em que, para habilitar o agente de compra ou venda de um livro, um preço deve ser fornecido. O *framework* não possui este *hot-spot*, logo a saída foi utilizar um sistema legado. Neste caso foi criado um *role* para que fosse adicionado o novo serviço disponível em um sistema legado. O ambiente agora possui acesso a diversos sistemas de comércio de livros na *Internet*, tais como ***Amazon*** e ***FatBrain***. Desta forma, o usuário por meio de um formulário pode optar por saber o preço do livro novo destas livrarias para poder ter uma base na hora de fornecer o preço ao seu agente.

O grande problema desta solução é garantir que, quando uma mudança na interface de consulta das livrarias virtuais ocorra, não cause impacto no sistema que o está utilizando, neste caso o VBookM@rket. O item "*Price Positioning*" utiliza um *role* para realizar a consulta A figura abaixo, mostra um exemplo desta solução.

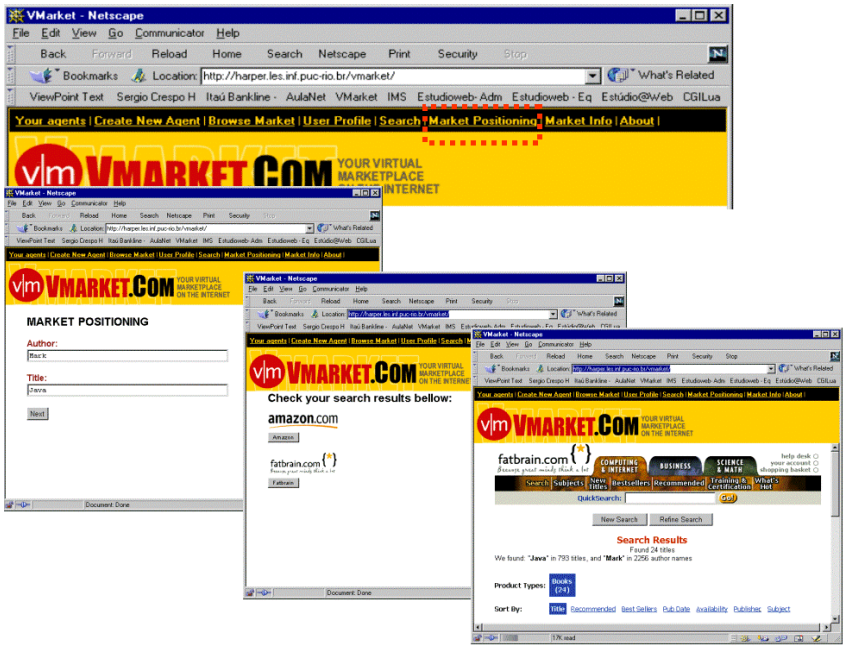

Figura 3.23 - VMarket com sistema legado

Neste ambiente, a solução foi implementada usando o conceito de *roles*, uma vez que qualquer alteração na interface da aplicação legada pode ocasionar uma operação incorreta no sistema principal. A figura 3.24, abaixo, mostra os detalhes de implementação.

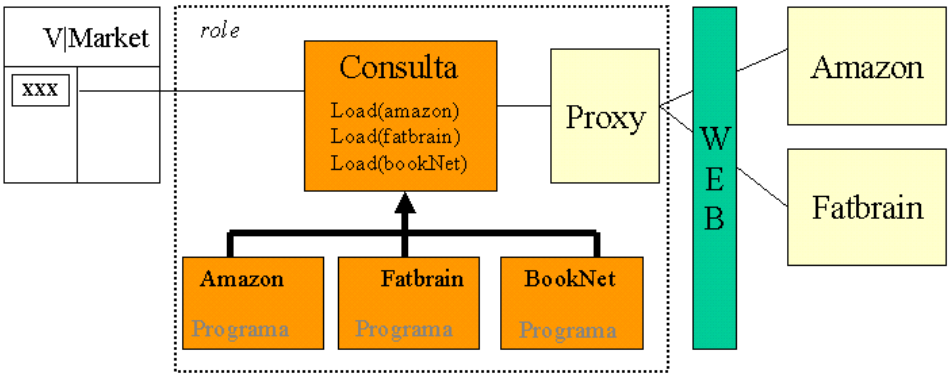

Figura 3.24 - V|Market com sistema legado: detalhes de implementação

Na figura 3.24, o Sistema V|Market utiliza dois sistemas legados para o fornecimento de preços sobre livro. Cada um dos sistemas legados possui detalhes particulares de interface para o fornecimento da informação desejada. O *role* utilizado permite que várias visões possam ser implementadas para o serviço "*Price Positioning*" do V|Market. A forma da solução utilizada será apresentada com maior profundidade nos próximos capítulos.

***WebFrameworks*: Problemas de carência de requisitos**

Este problema, já discutido no Capítulo 2, trata da carência de requisitos em *WebFramework*. O problema ocorre quando a composição de dois ou mais serviços presentes em um *WebFrameworks* para a solução de um problema não cobre todo o conjunto de requisitos. Se o serviço do *WebFramework* é do tipo estático, a carência de requisitos pode ser resolvida com uma interface adicional para ambos os *WebFrameworks*.

Se o problema residir em serviços ativos, será necessário algum *software* do tipo mediador (*mediator*) [Gamma et alli 1995] para minimizar o problema. Neste caso, o *software* deve prover um canal de comunicação entre ambos os serviços ativos do *WebFramework*, de forma a certificar que os dados recebidos ou enviados o foram com sucesso.

***WebFrameworks*: superposição de perspectivas**

Em *frameworks*, este problema ocorre quando classes do *framework* e/ou outro *framework* ou componente a ser reutilizado possuem representações diferentes para o mesmo problema, ou seja, as duas classes foram modeladas com perspectivas diferentes.

Em *WebFrameworks*, este problema relaciona-se a serviços. Logo, ele ocorre quando serviços do *WebFrameworks* possuem a mesma identificação, realizando a mesma

tarefa. No caso de *WebFrameworks*, uma boa alternativa é usar *roles* para a identificação do serviço a ser utilizado. Isto porque um serviço pode ter associado a si um conjunto de *roles* que permite visões diferenciadas para o mesmo serviço. A figura 3.25 exemplifica o que foi dito.

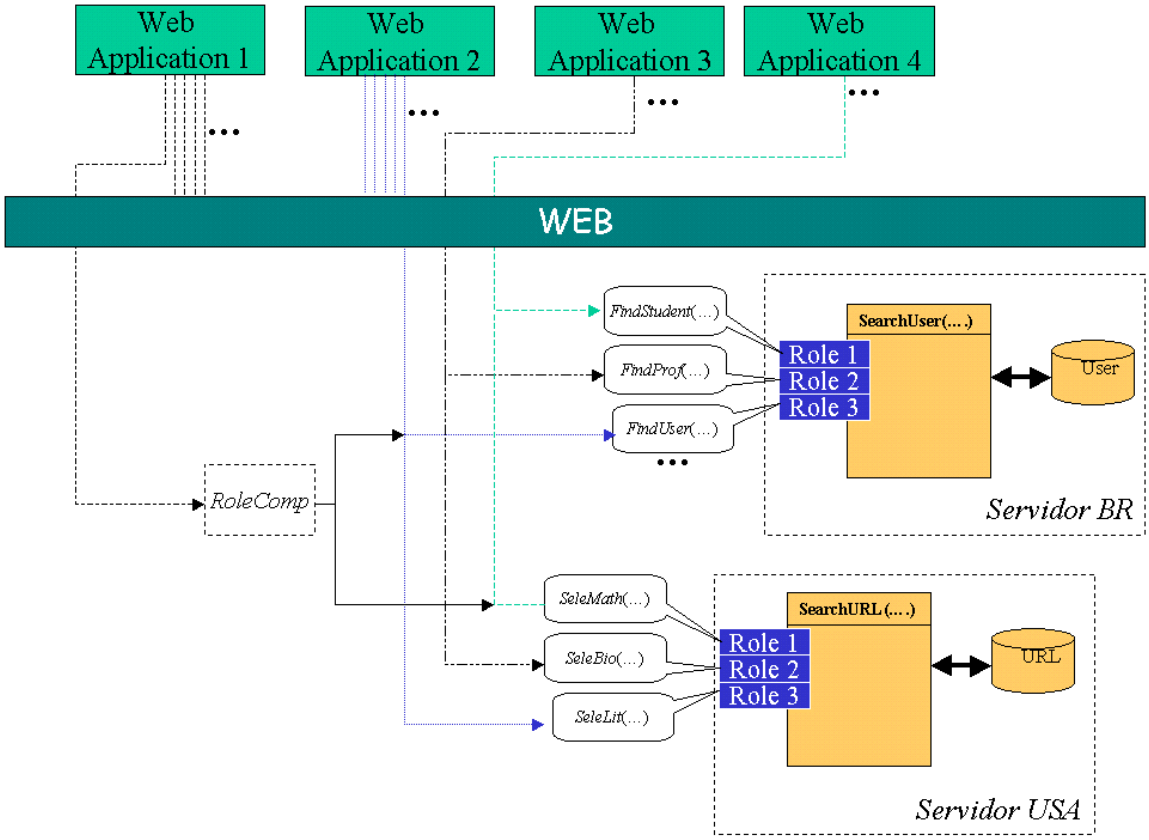

Figura 3.25 - Sobreposição de perspectivas

Na figura 3.25, várias instâncias do *WebFramework* utilizam visões diferenciadas para um mesmo serviço. No exemplo, podemos identificar que o serviço "*SearchUser*" no servidor BR possui um conjunto de vários *roles* que permitem que este serviço atue de forma diferenciada para as diversas aplicações geradas. Estes *roles*, ou no caso perspectivas, são: *FindStudent*, *FindProf* e *FindUser*.

Onde:

Aplicação	Role	Serviço
WebApplication 4	*FindStudent:* utiliza o serviço *SearchUser* para selecionar estudantes em uma base de dados de usuários.	*SearchUser*

WebApplication 3	*FindProf:* utiliza o serviço *SearchUser* para selecionar professores em uma base de dados de usuários.	*SearchUser*
WebApplication 1	*FindUser:* utiliza o serviço *SearchUser* para selecionar usuários em geral em uma base de dados de usuários.	*SearchUser*

Tabela 3.4 – *Roles* e serviços associados

Desta forma, como se pode ver na tabela 3.4, podemos ter múltiplas visões para um mesmo serviço sem que este comprometa o comportamento das aplicações geradas a partir do *WebFramework*.

Agora identificaremos outros tipos de problemas não identificados nos estudos realizados no Capítulo 2 sobre *frameworks* e que estão presentes em *WebFrameworks*. Os problemas particulares da plataforma Web serão agora apresentados e discutidos.

***WebFrameworks*: Assincronismo de tarefas**

Sem dúvida, um dos maiores problemas no desenvolvimento e integração ou composição de serviços em plataforma Web é o seu caráter predominantemente assíncrono. Os serviços uma vez disponibilizados em servidores Web, tornam-se acessíveis a qualquer outro servidor, desde que este conheça a sua interface de entrada. O assincronismo das tarefas não permite um controle muito rígido e confiável para identificarmos se o serviço durante o seu período de execução conseguiu executar plenamente o seu *script*. Problemas de conexão, queda de servidores, dentre outros, são comuns e, quando acontecem, precisam ser identificados e a aplicação que utiliza tais serviços precisaria ser notificada, de alguma forma, que o serviço que uma determinada tarefa executa não conseguiu completar a sua missão. Este problema torna-se muito mais grave na medida em que o término de um serviço serve de entrada de dados para outro. Serviços que geram algum tipo de *input* para outros não podem garantir que os dados foram bem gravados e/ou seu processamento foi bem realizado. Como estamos tratando de sistemas que são basicamente assíncronos, não temos como identificar que um terceiro serviço que utiliza os dados gerados por um outro serviço entre em execução e faça algum processamento com essa informação

errônea. Isto é muito diferente de sistemas gerados em Java ou C++ nos quais temos pleno controle da entrada e saída de módulos na memória para execução.

Hoje o máximo de resposta que um serviço oferece ao seu cliente em caso de queda de conexão é uma mensagem, apresentada pelo *browser* cliente, que tem configurado um tempo limite para *time-out* de conexão com o servidor.

Exemplo de problemas:

Uma aplicação utiliza um serviço, que está em outro servidor, para identificar se um usuário tem acesso a uma determinada parte de um sistema e fazer o seu redirecionamento. Então temos:

Um *script* que atualiza um banco de dados ou uma área de uso público na qual a informação sobre aonde ir estará disponível. O processo de processamento envolve a recepção dos dados pelo servidor *CGI*, que depois serão passados a um *script* que executará e receberá os parâmetros vindos do cliente. Os problemas que podem acontecer são:

- Os dados não chegarem ao servidor *CGI*: neste caso, basta fazer uma nova tentativa;
- A conexão poder ser estabelecida e os dados foram passados, mas ocorreu uma perda da informação no destino: neste caso o *script* que processa a informação poderá retornar um valor incorreto a aplicação cliente;
- Os dados chegaram bem ao servidor *CGI* e o *script* recebeu corretamente os dados, mas um problema no servidor impediu que o *script* pudesse atualizar o banco de dados: neste caso, uma resposta também inválida é passada para a aplicação cliente que não tem como saber dos problemas ocorridos no servidor durante a execução do *script*;

Com base nestes problemas, fica claro que não há uma garantia sobre do retorno da informação por parte do servidor caso algum problema intrínseco ao processamento ocorra. Os problemas relatados à aplicação cliente são, geralmente, por decorrência de *time-out* de conexão.

***WebFrameworks*: Trocas de mensagens entre serviços**

Outra característica importante na plataforma Web é a forma com que os dados são passados aos servidores *CGI*. Este dados são passados por meio de formulário, no qual os métodos para a transferência são *GET* e *POST*. O formulário, uma vez acionado, faz uma conexão com um servidor *CGI* e aciona um determinado *script*

para processar a informação. Se esta informação precisar enviar algum dado para outro *script*, isto deve ser feito também por um formulário. Neste caso, campos escondidos são adicionados na aplicação cliente. Estes dados estão inseridos em um formulário que, quando acionado, passará os dados digitados pelo usuário e que estão escondidos. Como podemos ver, os dados ficam acoplados à aplicação cliente e, caso algum problema ocorra com essa aplicação, todo o processo perde-se, pois os dados estão salvos na memória volátil da aplicação cliente. Hoje as linguagens de *scripts* ainda não implementam muito bem a passagem de dados sem que este esteja embutido em uma página *HTML*, o que torna o processo muito dependente e fragilizado na aplicação que o recebe, pois não existe nenhuma forma de persistência destas informações.

Em realidade, o que se forma, em geral, é uma grande corrente na qual cada página *HTML* possui um conjunto de dados embutidos que são passados como parâmetros aos vários *scripts* para processamento. No momento em que um deste elos se rompe, todo o processo fica comprometido.

Estes problemas mencionados serão melhor tratados no próximo capítulo mediante a utilização de um *Blackboard* distribuído na Web, para permitir que mensagens possam ser deixadas para que serviços possam recebê-las; para que serviços possam obter um *status* de alguma transação executada para saber se esta foi bem concluída ou não e identificar em que passo falhou a execução do serviço e a forma de recuperá-lo.

3.5 Conclusão

Neste capítulo, foram apresentadas várias formas para o desenvolvimento de *frameworks*. A motivação hoje na construção de *frameworks* está voltada para a idéia de se trabalhar com uma granularidade menor do que a abordagem mais tradicional.

Propostas como as de Fayad em [Fayad 1999b] que utiliza o conceito de componentes para a construção de *frameworks* enfocando a reutilização por meio da utilização de arquiteturas de 3 camadas. Propostas como as de Jan Bosch em [Bosch 1998][Bosch 1999b] utilizando fragmentos arquiteturais em que, novamente, a reutilização é enfocada por meio da utilização de *roles* como forma de oferecer maior semântica aos modelos de *frameworks* baseados em classes. Propostas de processos baseados em *roles* em [Riehle 1998] que enfatizam em reutilização e uma rápida prototipação,

mediante o uso de *roles* para o fornecimento de maior semântica aos modelos baseados em classes. Estes enfoques iniciam o questionamento de que um modelo baseado em classes não produz semântica suficiente para que a cooperação entre as classes do *framework* possam obter os melhores resultados no momento de sua instanciação e manutenção.

Estas abordagens apontam para alguns aspectos de grande relevância na construção de *frameworks* que são: utilização de componentes e ou fragmentos em lugar de classes para a formação da estrutura do *framework*, uso de *roles* para proporcionar maior semântica e visões diferenciadas para um mesmo componente ou classe.

Utilizando esse referencial teórico, mudamos o enfoque em que se desenvolve *frameworks* de orientação a objetos para a plataforma Web que possui um conjunto singular de peculiaridades. Na Web, o foco para a construção de um *framework* são os serviços nela disponíveis, a forma como eles podem cooperar entre si e como podem ser compostos para a geração de serviços mais complexos. Desta forma, um novo conceito para *frameworks* na Web foi criado, o *WebFramework*. **Um *WebFramework* é um conjunto de serviços que permite colaboração entre si e composição por meio de *roles* para a geração de aplicações na Web.** As composições dos serviços definirão o perfil de uma aplicação a ser gerada. Um serviço poderá ter vários *roles* permitindo visões diferenciadas a partir de diversas aplicações. Os problemas sobre composição apresentados no Capítulo 2 foram também analisados sob este novo enfoque. Problemas específicos da plataforma Web foram apresentados e discutidos.

Dos problemas levantados, os que causam maior impacto são:

Assincronismo de tarefas: no qual a utilização de um determinado serviço não depende de nenhum sincronismo prévio. Os serviços, uma vez disponíveis, podem ser acionados por um conjunto de aplicações presentes na Web, sem que o serviço em questão tenha noção da necessidade de algum tipo de colaboração necessária para o seu processamento dentro desta aplicação;

Recuperação em caso de falhas: Pelo assincronismo das tarefas presentes na plataforma Web, fica muito difícil identificar se um determinado serviço, uma vez acionado, conseguiu concluir corretamente suas tarefas;

Trocas de mensagens entre serviços: As linguagens de *script* existentes carecem fortemente de mecanismos que asseguram a correta transferência de mensagens entre os serviços. Como os serviços são basicamente assíncronos, uma determinada mensagem deve ficar aguardando que um serviço seja acionado para que ela possa ser

enviada. Em sistemas convencionais, utilizando-se linguagens compiladas, este problema torna-se simples de ser gerenciado, mas quando ele está presente na plataforma Web, o seu grau de dificuldade aumenta consideravelmente. Hoje na Web ainda não existe nenhum mecanismo implementado em linguagens de *scripts* que permita que uma mensagem fique aguardando o seu destinatário para ser entregue.

A conclusão final deste capítulo é que, para um *WebFrameworks* gerar aplicações a partir de um conjunto básico de serviços (que não foram projetados para atuarem em conjunto ou compostos), torna-se fundamental o uso de *roles* como meio de fornecer maior semântica, diferentes visões e como mediador de diferentes interfaces.

No próximo capítulo, veremos que *roles* são equivalente em *design* a *ADV's/ADO's*, que foram criados para especificar de forma clara e formal a separação da interface do usuário da aplicação em que ela está inserida, oferecendo assim um método de *design* sistemático, e independente da aplicação específica. Esta independência, fornecida pelo método, enfatiza *separation of concerns* e reutilização de componentes. Esta teoria está fortemente solidificada em [Cowan and Lucena 1995], [Alencar, Cowan and Lucena 1995a], [Alencar, Cowan and Lucena 1995b], [Alencar, Cowan and Lucena 1995c] e [Alencar, Cowan, Lucena et alli 1995].

Capítulo 4

ADVs/ADOs como uma teoria para suporte à composição de *WebFrameworks*.

Resumo

Este capítulo apresenta a teoria de "*Abstract Design View* – *ADV* e *Abstract Design Object*– *ADO*" que é estendida para a plataforma Web com seus conceitos adaptados. Os conceitos de *ADVService*/*ADOService* são introduzidos, com destaque para a noção de *roles* neste novo contexto. Suas propriedades são apresentadas junto com suas principais características e exemplos. Esta teoria oferece suporte à composição de *WebFrameworks*. No final, propõe-se uma notação gráfica de *design* para permitir o *design* detalhado de aplicações baseadas na Web com a explicitação de *roles* e de serviços, *ADVServiceChart*, baseada na teoria de *ADVChart*. Esta notação gráfica desempenhará o mesmo papel da *UML* na especificação de *frameworks* orientados a objetos.

4.1 Introdução

Uma das grandes dificuldades para a obtenção de reutilização de *design* e implementação é o fato de que tanto o *design* como a implementação acumularem detalhes sobre o ambiente ao qual pertencem. Isto ocasiona um grande acoplamento entre os módulos e, desta forma, dificulta o seu reuso, uma vez que sua interface interna está profundamente comprometida com alguma estrutura de dado.

Vários modelos de arquiteturas e metodologias de programação [Carneiro, Coffin, Cowan e Lucena 1993] [Myers 1991] têm sido propostos com o objetivo de separar a interface do usuário de sua correspondente aplicação. O modelo de *ADV* foi criado, inicialmente, para especificar de forma clara e formal a separação da interface do usuário da aplicação em que ela está inserida, oferecendo um método de *design* sistemático e independente da aplicação específica. Esta independência, fornecida

pelo método, enfatiza o conceito de *separation of concerns* e reuso de componentes [Cowan e Lucena 1995].

4.2 Abstract Design View e Abstract Design Object

Um *ADO* é um construtor de *design* que permite o gerenciamento do estado de uma aplicação e não tem contato direto com o mundo exterior. Um *ADO* possui um estado e uma interface pública que pode ser utilizada para consultas ou para alterar o seu estado. Um *ADV* é um *ADO* mais robusto para suportar o desenvolvimento de visões genéricas de um *ADO* [Alencar, Cowan and Lucena1995]. O que diferencia basicamente um *ADV* de um *ADO* são as restrições impostas à sua execução. Vários tipos de sistemas de *software* foram implementados utilizando-se os conceitos de *ADVs*/*ADOs* tais como: editores gráficos [Cowan et alli 1992]; mecanismos para interconectar módulos em interfaces de usuários [Lucena, Cowan and Potengy 1992]; para dar suporte ao processamento concorrente em editores colaborativos; e para *design*/implementação de sistemas de *Ray-tracer* em ambientes distribuídos. A figura 4.1 mostra a forma de interação entre *ADV* e *ADO*.

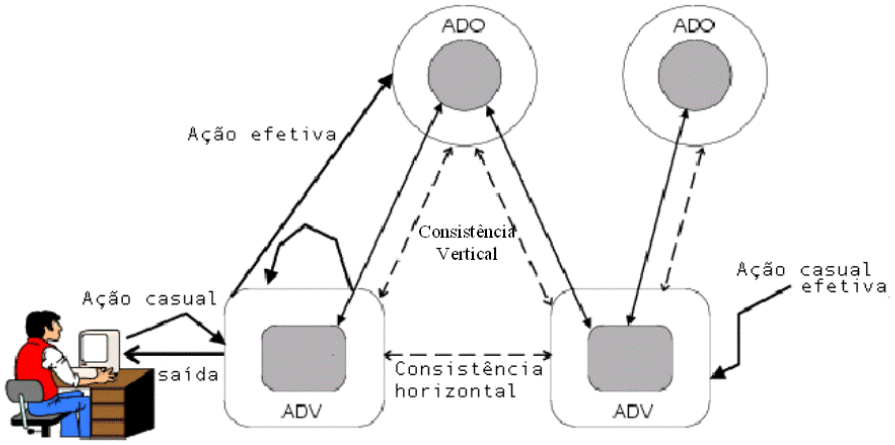

Figura 4.1 - *ADV/ADO* Modelo de Interação

[Alencar, Cowan, Lucena et alli 1995]

Um *ADO* pode ter associado a si um conjunto de visões (*views*) que são implementadas por *ADVs*. Um *ADV* conhece o nome do *ADO* ao qual está conectado, porém um *ADO* não possui nenhum conhecimento sobre os ADVs que estão a ele associados.

Se o estado de um *ADO* é alterado, então o estado do *ADV* a ele mapeado também sofre alterações de forma a ser consistente com o objeto que é observado.

Mecanismos de morfismo são utilizados para permitir que estas trocas de estado sejam mapeadas utilizando-se as interfaces públicas contidas nos *ADO's* [Alencar, Cowan and Lucena 1995c].

Uma visão pode manipular interfaces dos usuários ou interfaces públicas de um *ADO* de forma a alterar a forma pela qual um outro *ADV* o visualiza. Uma *view* pode alterar o estado de um *ADO* a ela associado por meio de algum tipo de evento produzido por um usuário ou de uma ação de outro *ADO*. A figura 4.1 ilustra o que foi dito.

Na figura 4.1, ambos *ADV* e *ADO* podem atuar sobre ações representadas por funções ou eventos de entrada de dados para ocasionar uma alteração no estado de um *ADO* ou acionar algum mecanismo de consulta. As ações podem ser divididas em dois grupos de categorias: casuais e efetivas. O termo casual denota eventos de entrada que atuam diretamente sobre um *ADV* quando este atua como um dispositivo de interface para usuários ou para alguma outra mídia. Ações casuais não podem ser geradas por objetos internos ao sistema. Na figura 4.1, o usuário gera uma ação casual a partir de seu teclado ou *mouse*, tendo o *ADV* atuado como um mecanismo de interface para este usuário. Ações efetivas são ações geradas direta ou indiretamente por uma ação casual. No exemplo acima, quando o usuário digita alguma coisa, o *ADV* recebe esta ação casual e a processa. Como uma saída deste procedimento, uma ação efetiva é gerada pelo *ADO* que está a ele associado produzindo, por exemplo, uma *query* de tal forma que o usuário posteriormente possa receber como *output* do *ADV*, que está atuando com uma interface para este usuário, uma relação de dados referente à sua consulta inicial. Desta forma, podemos verificar que somente ações casuais podem causar alguma alteração no estado de um sistema, pois a existência de uma ação efetiva pressupõe a existência de uma ação casual. Logo uma ação efetiva existe somente se uma ação casual existir.

Ações casuais são originadas de fora do sistema e, portanto, não podem gerar outras ações casuais. A figura 4.2, abaixo, resume as dependências entre ações.

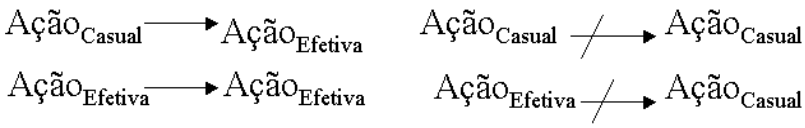

Figura 4.2 - Dependência entre ações

[Alencar, Cowan, Lucena and Nova 1995+]

4.2.1 Comunicação entre ADV's/ADO's

A comunicação entre *ADV* e uma instância de um *ADO* é, essencialmente, uma chamada síncrona que é mapeada entre o *ADV* e seu *ADO* associado, porém comunicação assíncrona também pode ser implementada.

Cada relacionamento entre um *ADV* e um *ADO* é definido como um par de elementos no qual um possui uma correspondente ação dentro de um *ADV* e outro dentro de um *ADO*. Em geral, o corpo de um *ADV* pode ser visto como um elemento de *design* que especifica como um *ADV* interage com seus *ADOs* associados. A figura 4.3 ilustra um *design* de estrutura que pode modelar comunicação assíncrona e síncrona usando *ADVs*.

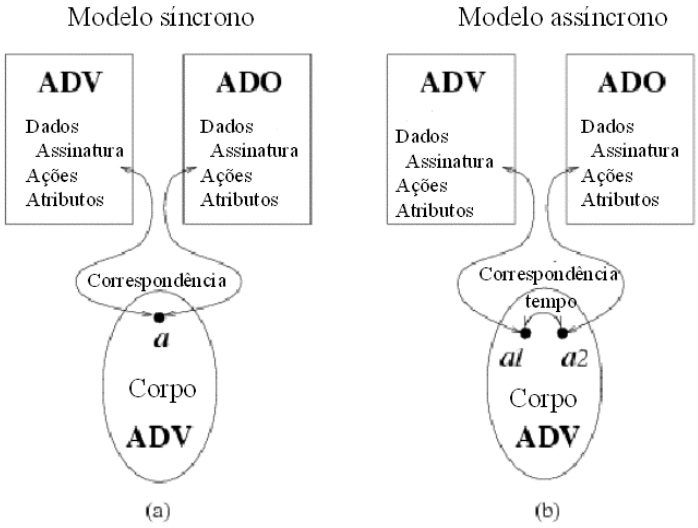

Figura 4.3 - Comunicação entre os modelos

[Alencar, Cowan, Lucena and Nova 1995]

A figura 4.3 (a) mostra um modelo de comunicação síncrona, no qual "***a***" é uma simples ação dentro do corpo de um *ADV* que possui uma ação correspondente dentro do *ADV* e de seu *ADO* associado. Na figura 4.3 (b), temos uma comunicação assíncrona na qual as ações a1 e a2 possuem, respectivamente, ações no seu *ADV* e *ADO*

A seguir, apresentar-se-á uma nova visão de *ADVs*/*ADOs* usando *WebFrameworks* como plataforma. Este referencial teórico que será adaptado pode ser encontrado em [Cowan and Lucena 1995], [Alencar, Cowan and Lucena 1995a], [Alencar, Cowan and Lucena 1995b], [Alencar, Cowan and Lucena 1995c] e [Alencar, Cowan, Lucena et alli 1995].

4.2.2 ADVs / ADOs em WebFrameworks: ADVService e ADOService.

Um serviço dentro da Web é composto por um ou vários *scripts* de forma a realizar uma determinada tarefa a partir do *browser*. Geralmente serviços na Web podem ser vistos como caixas pretas onde os parâmetros passados pelo *browser* são decodificados por um *script* no Servidor WWW da aplicação. Um mesmo *script* pode ser acionado por uma grande variedade de aplicações diferentes e de forma assíncrona.

Um *WebFramework* pode ser visto como um conjunto de serviços que permitem colaboração entre si e composição por meio de *roles*. As composições dos serviços definirão o perfil de uma aplicação a ser gerada. Um serviço poderá ter vários *roles* permitindo visões diferenciadas a partir de diversas aplicações.

O relacionamento entre os vários serviços encontrados na Web serão realizados por meio de *roles* que fazem o papel de uma interface para a colaboração dos serviços. A composição de serviços também segue esta regra, ou seja, quando um serviço necessita de vários outros para cumprir suas obrigações, torna-se necessária a definição de um *role* que componha esses serviços, permitindo que as interfaces envolvidas possam cooperar livremente e sem alterações em sua implementações originais.

Baseado na definição acima de *WebFrameworks*, podemos, então, definir *ADVService* e *ADOService*.

Definição 1: *ADVService*.

Um *ADVService* é um construtor de *design* que permite mapear, visualizar e servir como instrumento de conexão entre serviços disponíveis na Web.

Definição 2: *ADOService*.

Um *ADOService* é um construtor de *design* que permite o gerenciamento do estado de um serviço na Web e não tem contato direto com o mundo exterior. Um *ADOService* possui um estado e um conjunto de variáveis e funções públicas que podem ser utilizados para consultas ou para alterar o seu estado.

4.3 Esquemas para *ADVService* / *ADOService*

Um *ADVService* permite que o *design* de uma aplicação possa executar vários *roles* para um mesmo *ADOService*. Um *ADVService* é um *ADOService* mais robusto para suportar o desenvolvimento de *roles* genéricos de um *ADOService*.

ADVServices e *ADOServices* possuem diferentes papéis em sistemas de *software* e, conseqüentemente, são descritos por diferentes esquemas. Estes esquemas são descrições das propriedades estáticas e dinâmicas dos serviços que estão disponíveis. A estrutura de *ADVService* e *ADOService* está dividida em três seções: declarações, propriedades estáticas e propriedades dinâmicas.

O que diferencia, basicamente, um *ADVservice* de um *ADOService* são as restrições impostas à sua execução. Inicialmente, introduziremos os esquemas abstratos e, posteriormente, um exemplo concreto.

Esquema Abstrato para *ADVService*.

ADVService Name for ADOService ADOService Name

- Declarations
  - Data Signatures - funções
  - Attributes - atributos das propriedades do *role* executado
  - Causal Actions - lista das possíveis entradas de ações
  - Effectual actions - lista das possíveis ações efetivas
  - Nested ADVServices - composição de *ADVServices,* acesso a funções
- Static Properties
  - Constraints - restrições nos valores dos atributos
  - Derived Attributes - atributos derivados
- Dynamic Properties
  - Initialization - atributos para inicialização
  - Interconection - descrição dos processos de comunicação
  - Valuation - a avaliação dos eventos dos atributos
  - Behavior - comportamento das propriedades do *ADVService*

End ADVService Name

Esquema Abstrato para ADOService

ADOService ADOService Name

- Declarations
  - Data Signatures - funções
  - Attributes - atributos das propriedades do *role* executado
  - Causal Actions - lista das possíveis entradas de ações
  - Effectual actions - lista das possíveis ações efetivas

Nested ADOServices - composição de *ADOServices*, acesso a funções

Static Properties

Constraints - restrições nos valores dos atributos

Derived Attributes - atributos derivados

Dynamic Properties

Initialization - atributos para inicialização

Interconection - descrição dos processos de comunicação

Valuation - a avaliação dos eventos dos atributos

Behavior - comportamento das propriedades do *ADOService*

End ADOService Name

4.4 Consistência

A separação entre interface e aplicação torna possível criar diferentes representações visuais para uma simples coleção de *ADOServices* na Web. Por exemplo, a interface do usuário para um serviço que disponibiliza a hora na Web pode ser representada tanto em formato analógico, quanto digital ou em ambos. Outra possibilidade de uso para este serviço é utilizar os seus dados como elementos de entrada para outro serviço que necessite de um valor hora para o seu processamento. Neste caso, um outro *ADVService* pode atuar como uma interface entre o serviço da Web que fornece a hora e o outro sistema que necessita desta informação.

A consistência entre a representação visual (*ADVServices*) e o *ADOService* é garantida como conseqüência de *separation of concerns* introduzida pelo modelo de *ADVService*. A consistência entre os diferentes *ADVServices* é chamada de consistência horizontal, enquanto que a consistência entre o *ADVService* e seu respectivo *ADOService* é chamada de consistência vertical. A figura 4.4 mostra o que foi dito.

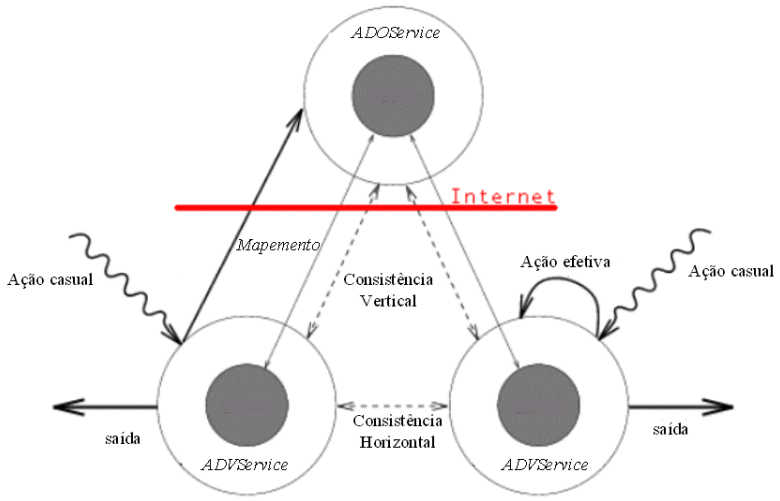

Figura 4.4 - Modelo de interação entre *ADVServices* e *ADOServices*

No exemplo citado, a consistência vertical garante a fidelidade da hora mostrada pelo *ADVService* com os serviços oferecido pelo *ADOService*, enquanto a consistência horizontal garante que todos os *ADVServices* terão a mesma hora ao mesmo tempo.

Nota-se que a hora é acessível aos diferentes *ADVServices* mediante um mapeamento que o *ADVService* realiza em seu correspondente *ADOService*. Desta forma, sempre que ocorre uma mudança de estado, uma troca é realizada nos *ADVServices* conectados ao respectivo *ADOService*. Aqui não ocorre nenhuma interferência ou ação do usuário para a mudança da hora. Esta mudança é realizada pelo sistema operacional da máquina.

4.5 *Roles* em *ADVServices* / *ADOServices*

No Capítulo 3, na sua conclusão, enfatizou-se que, para que *WebFrameworks* possam gerar aplicações a partir de um conjunto básico de serviços o uso de *roles* torna-se funcamental. Desta forma, pode-se obter maior semântica e diferentes visões entre interfaces permitindo a separação da interface do usuário da aplicação em que ela está inserida.

A figura 4.5, a seguir, exemplifica o que foi dito.

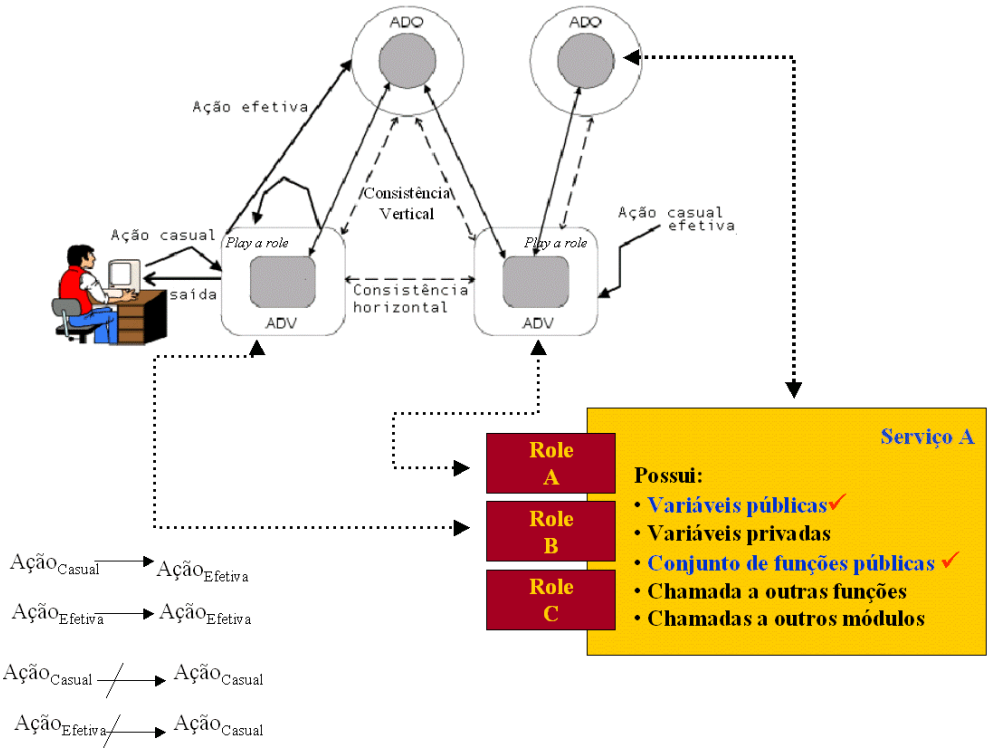

Figura 4.5 - Roles no modelo *ADVServices* / *ADOServices*

Na figura 4.5, o que mostramos é a equivalência entre a notação de *ADVServices* e *ADOServices* com a notação de *roles*. A equivalência dá-se na medida em que *roles* proporcionam *views* entre clientes e classes, possibilitando a reutilização em nível de requisitos. Com a utilização de *ADVServices*/*ADOServices*, podemos expressar a mesma idéia no nível de *design*, se ampliarmos um pouco a notação. Os pontos de equivalência e a notação adicional são apresentados na tabela 4.1, a seguir:

Roles sem a abordagem de *ADVServices/ADOServices*	**Roles baseados na abordagem de *ADVServices/ADOServices***
Role é uma *view* para uma classe.	*Role* é uma *view* para um serviço já existente na Web.
Um *role* é sempre o mesmo para uma determinada classe / cliente	Um *role* **pode ou não** desempenhar uma mesma *view* para um determinado serviço.
Um *role* proporciona somente uma *view* estática entre o cliente e a classe.	*Roles* podem ser estáticos ou dinâmicos.

Tabela 4.1 - comparação entre roles nas abordagens baseadas ou não em *ADVServices/ADOServices*

As características adicionais oferecidas pela notação apresentada são:

- Permitir múltiplas *views* para um determinado serviço;
- Interface entre mídias distintas;
- Reutilização de *design* em ponto grande;

A multiplicidade de *views* permite que um conjunto de *ADVServices* possa fazer uso de um único *ADOService*, sem que seja necessário a alteração do mesmo. Além do mais, a consistência entre observador e observado é garantida.

ADVServices podem servir de interface entre mídias distintas de forma a prover um canal comum entre elas. Esta é uma das propriedades de *ADVServices*/*ADOServices* que será apresentada e exemplificada mais adiante.

A reutilização em ponto grande é garantida na medida em que não se está aproveitando somente classes, que nem sempre implementam totalmente um determinado serviço. Neste caso, reutiliza-se serviços que são, em sua maioria, compostos por um conjunto de *scripts* ou até mesmo classes, e estes serviços, por sua vez, estão disponíveis na plataforma Web.

4.5.1 Dinâmica de Roles em ADVServices/ADOServices

Um *role* proporciona uma determinada visão entre uma aplicação e um determinado serviço (*ADOServices*) presente na Web.

Nesta abordagem, um *role* deixa de ser uma visão estática, ou seja, sempre a mesma entre o observador e o observado, e agora pode incorporar um aspecto dinâmico em sua criação. Esta nova característica permite que diversas visões possam ser criadas em tempo de execução fazendo com que um mesmo *ADOService* possa retornar informações diferenciadas aos seus vários observadores sem invalidar a consistência entre observador e observado. Esta nova característica será exemplificada na propriedade de criação de *ADVServices*/*ADOServices* a seguir. Quando um *role* é utilizado de forma estática entre um *ADVService* e o seu correspondente *ADOService*, dizemos que o *ADVService* está desempenhando um *ERole* (*Role* Estático), quando esta interação for dinâmica, dizemos que está desempenhado um *DRole* (*Role* dinâmico). A figura 4.6, a seguir, exemplifica esse conceito.

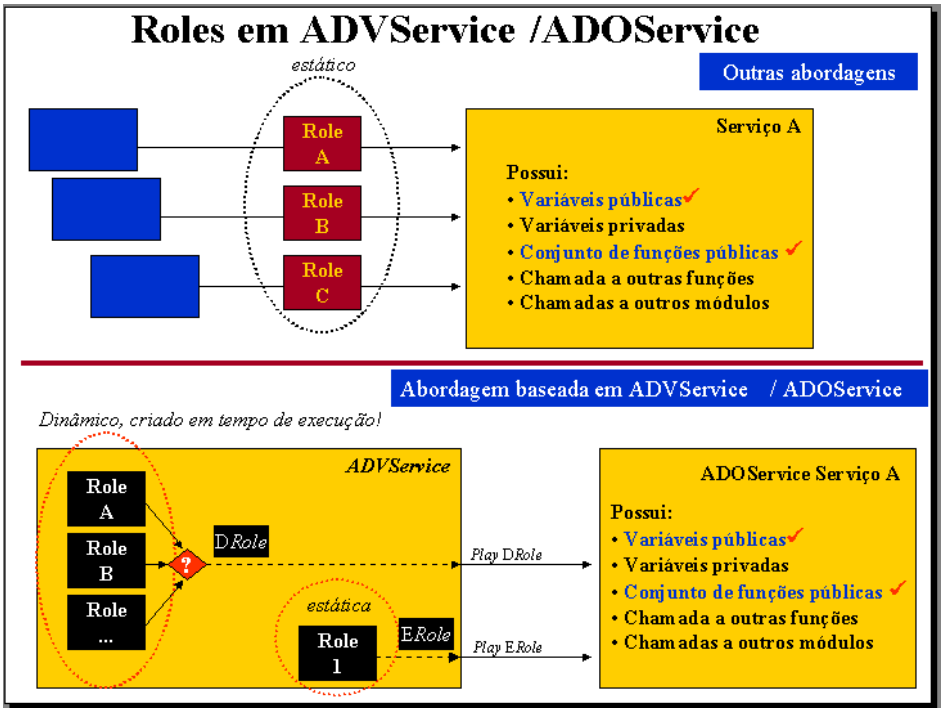

Figura 4.6 - Dinâmica de *roles*

Na figura 4.6, podemos ver que, em uma abordagem não baseada em *ADVServices*/*ADOServices*, um *role* está sempre associado ao observador e ao seu observado, sendo esta uma relação de 1:1 na qual o *role* desempenhado é sempre o mesmo. Já na abordagem baseada em *ADVServices*/*ADOServices*, como podemos ver, um *ADVService* pode criar vários *roles* em tempo de execução e definir que *role* ele necessita desempenhar em um dado momento. Esta dinâmica fornece uma grande flexibilidade para a composição de serviços que mudam periodicamente a sua interface com o mundo exterior.

4.6 Propriedades de *ADVService/ADOService*

As propriedades *ADVServices* / *ADOServices* são:

[1] Consistência vertical

[2] Consistência horizontal

[3] Composição

[4] Comunicação [síncrona/assíncrona]

[5] Criação [estático/dinâmica]

[6] Tipos de Mídia [pipes/filters]

A seguir, explicaremos cada uma das propriedades através de exemplos reais.

Propriedade 1: Consistência vertical

Um *ADVService* deve ser consistente com o seu respectivo *ADOService*.

Um *ADVService* é uma aplicação Web que tem no *browser* a sua interface básica. Este *ADVService* tem acesso, via uma chamada de formulário, a um determinado *ADOService* de tal forma que o retorno desta operação, que é realizada pelo servidor WWW no qual o *ADOService* está inserido, seja sempre o mesmo desde que a chamada no *ADVServe* seja sempre a mesma para todas as suas *views*. Neste caso, em que a chamada é sempre a mesma, dizemos que o *ADVService* executa um *ERole*, ou seja, um **Role estático**. Desta forma, pode-se garantir que uma consistência horizontal é sempre atingida. Vide o exemplo na figura 4.7 a seguir:

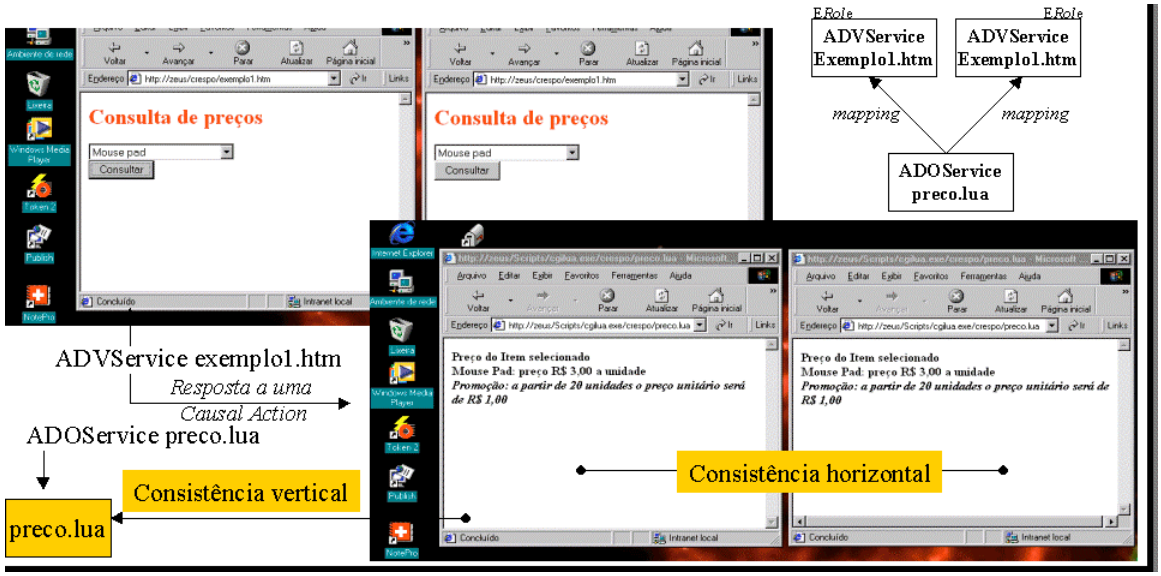

Figura 4.7 - Propriedade 1: Consistência vertical

No exemplo da figura 4.7, temos a seguinte situação: uma aplicação que executa uma consulta de preços. Existe uma lista de itens a serem consultados e, para cada item, um preço com algumas possíveis promoções. Nesta aplicação, o *ADVService* chamado Exemplo1.htm possui um formulário que utiliza um serviço, *ADOService*, chamado preco.lua. Para qualquer consulta de itens, somente este *ADService* é acionado. Desta forma, para todas as consulta a partir deste *ADVService*, uma mesma saída deverá ser mostrada ao usuário sempre que um determinado item for consultado. O *ADVService* desempenha um determinado *role* para cada item consultado. Desta forma, para a consulta de um mesmo item, uma mesma resposta será apresentada ao usuário, como pode ser visto na figura 4.7. Assim sendo, o *ADVService* é consistente com o seu *ADOService* para aquele *role* desempenhado, e todas as consultas que utilizarem este *ADVService* para um mesmo item mostrarão a mesma resposta. Neste

caso, além de uma consistência vertical, temos também uma consistência horizontal, que será explicada adiante.

Propriedade 2: Consistência horizontal

Esta propriedade é garantida sempre que um *ADVService* executa um *ERole*.

Desta forma, todas as *views* do *ADVService* geram a mesma resposta a partir de um mesmo *ADOService*.

***Role* Estático garante consistência horizontal**

Propriedade 3: Composição

É a possibilidade de aninhamento de *ADVServices* ou *ADOServices*. Neste caso, o *ADVService*/*ADOService* principal (que contém os aninhados) tem pleno conhecimento dos *ADVServices*/*ADOServices* aninhados e acesso a suas funções e variáveis públicas. A figura 4.8, a seguir, mostra um exemplo de composição.

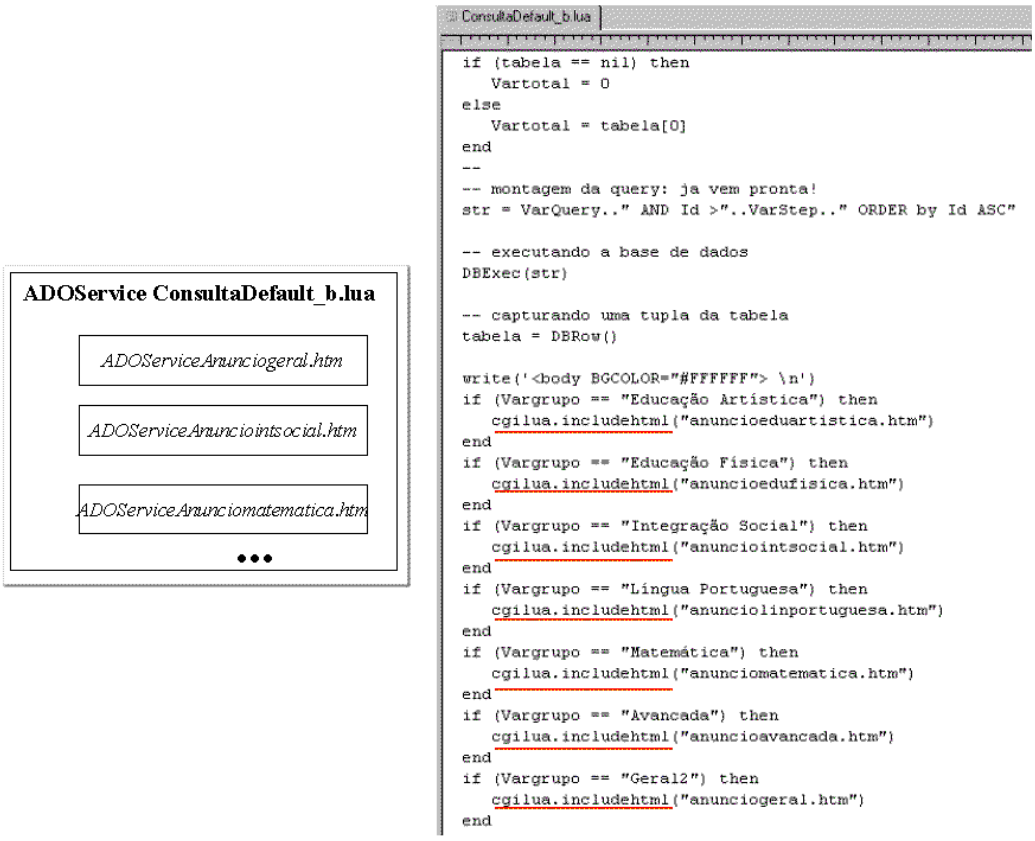

Figura 4.8 - Composição de *ADOServices*

Na figura 4.8, temos o *design* de um *ADOService* que é composto por um conjunto de outros *ADOServices*. Os outros *ADOServices* provêm informações sobre como deverá ser montada a página de resposta a uma consulta do usuário, tendo algumas restrições quanto ao tipo de item consultado, de forma a mostrar diferentes *banners* com

propaganda e mensagens no *bottom* da página. O *ADOService* principal conhece e fornece dados para que os *ADOServices* incluídos possam configurar a sua saída no *browser*. O exemplo apresentado utilizou uma linguagem de *script* chamada *CGLua* [Hester, Borges and Ierusalimschy 1997] [Ierusalimschy, Figueiredo,Celes 1996] para implementar o *ADVService* bem com o seu respectivo *ADOService*.

Propriedade 4: Comunicação

Esta propriedade permite diferenciar a forma de comunicação pela qual o *ADVService* tem acesso ao seu respectivo *ADOService*. A forma poderá ser síncrona ou assíncrona.

Forma síncrona: é a forma mais usual de comunicação entre um *ADVService* e seu respectivo *ADOService*. Nela a resposta é imediata e não há a necessidade de se conhecer antecipadamente nenhum conteúdo de variável para o processamento a ser realizado no *ADOService*.

Forma assíncrona: é a forma de comunicação em que, quando um *ADVService* envia um pedido a um *ADOService*, este, por sua vez, necessita ter conhecimento prévio de algumas variáveis para poder executar o seu processamento. Para que isto possa ser possível, um *Blackboard* [Gamma et alli 1995] é utilizado pelo *ADOService* de forma a obter, em tempo de execução, os dados necessários para o seu processamento. Neste caso, o *ADOService* conhece as funções e variáveis públicas do *Blackboard*.

A figura 4.9 mostra um exemplo do que foi dito.

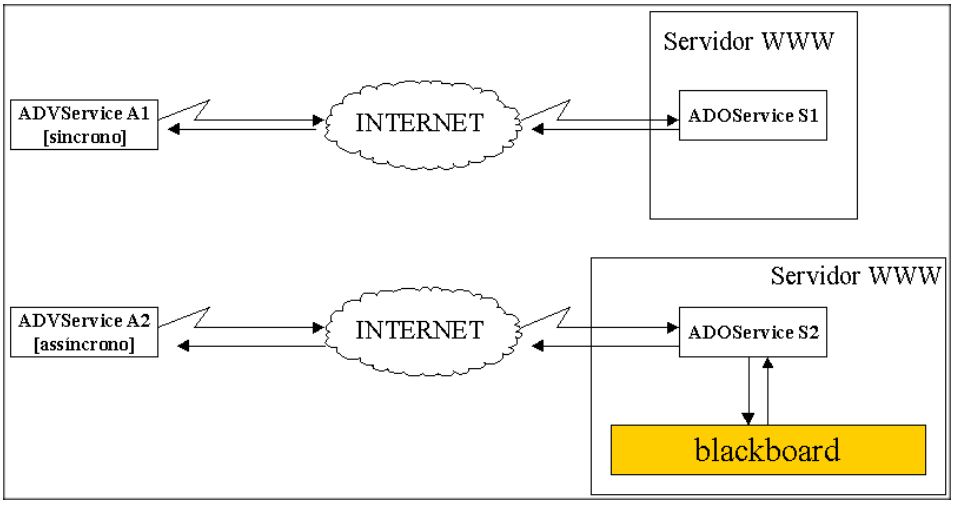

Figura 4.9 - Propriedade de comunicação

No exemplo da figura 4.9, temos duas opções de comunicação sendo realizadas. A primeira é uma comunicação síncrona entre um *ADVService* e seu respectivo *ADOService* de forma que o *ADOService* não necessite de nenhuma outra informação, exceto as enviadas pelo *role* que está desempenhando no momento o *ADVService*,

para o seu processamento. Neste caso, uma resposta imediata, após o *ADOService* realizar o processamento, será enviada ao seu observador, no caso o *ADVService* "*ADVService* A1" e este, por sua vez, exibirá a informação no *browser* do usuário final. No segundo caso, uma comunicação assíncrona é solicitada de tal forma que o *ADOService* procurará por informações extras, que não foram providas pelo *ADVService* A2, para poder executar o seu processamento. Como a persistência de variáveis em *CGI* não é implementada facilmente e sua confiabilidade pode ser questionada, uma estrutura baseada no Design Pattern do tipo *Blackboard*[Gamma et alli 1995], foi construída de forma a implementar a persistência em CGI para plataforma Web. Assim, sempre que se fizer necessário deixar informações para que outros *ADOServices* as utilizem, este *Blackboard* deverá ser utilizado. A figura 4.10 exemplifica o conceito do *Blackboard*.

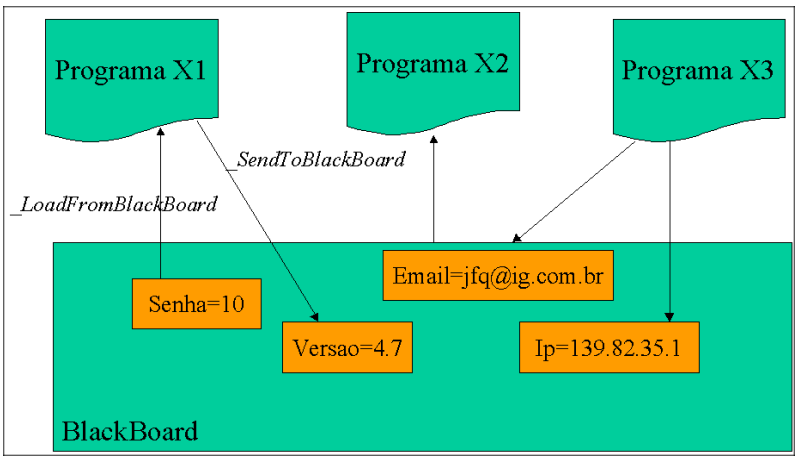

Figura 4.10 - *Blackboard*

A figura 4.10 mostra um exemplo no qual diversos *ADOServices* utilizam o *Blackboard* para buscar informação ou para gravar informação para que outros *ADOServices* possam utilizá-las durante seu processamento. Esta estrutura de *Blackboard* será mais detalhada no próximo capítulo.

Propriedade 5: Criação

Esta propriedade permite criar *ADVServices* com *Roles* Estáticos ou Dinâmicos. Um *ADVService* desempenhando um ***ERole*** permite que a propriedade de consistência horizontal seja sempre verificada. Um ***ERole*** faz com que todos os *ADVService* conectados a um determinado *ADOService* retornem a mesma mensagem ao seu usuário. Já um ***DRole***, ou seja, um *role* dinâmico, permite que vários *ADVServices* conectados a um determinado *ADOService* possam produzir respostas distintas ao seu usuário. Neste caso, a propriedade de consistência horizontal não pode ser garantida.

A figura 4.11 mostra um exemplo desta propriedade.

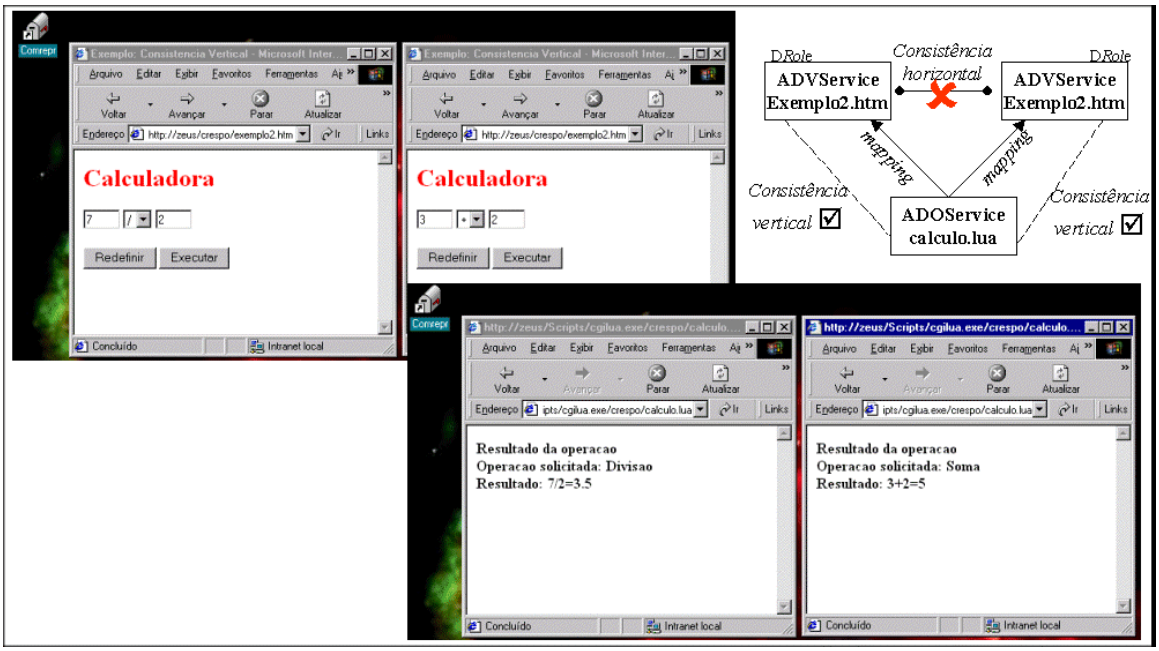

Figura 4.11 - Propriedade de criação

Na figura 4.11, podemos visualizar várias ocorrências de um *ADVService* para um determinado *ADOService*. O *ADVService* em questão implementa uma calculadora que realiza as quatro operações aritméticas básicas (soma, subtração, divisão e multiplicação). Este *ADVService*, assim como todos os outros, utiliza um mesmo *ADOService* para prover tal operação. Neste caso, cada uma das ocorrências pode desempenhar um *role* distinto para um mesmo *ADOService*. O *ADOService* possui uma interface do seguinte tipo:

- Recebe como parâmetros dois valores inteiros e um *string* contendo o tipo de cálculo a ser executado.

Desta forma, podemos ter D*Roles* sendo desempenhados pelos vários *ADVServices* para um mesmo *ADOService*. Então, podemos assegurar que existe uma consistência vertical entre o *ADVService*, desempenhando o seu *role*, e o seu respectivo *ADOService*, que, neste caso, é o mesmo para todos. Da mesma forma que a consistência vertical pode ser garantida, a horizontal não pode, pois cada *browser* fornecerá respostas baseadas em seus valores de entrada e da operação aritmética selecionada pelo usuário.

Propriedade 6: Tipos de mídia

Esta propriedade permite que um *ADVService* possa atuar como um *Design Pattern* tipo *Pipes and Filter* [Buschmann et alli 1996], transformando uma entrada de dados

de um determinado formato em outra de formato diferente que servirá como dispositivo de entrada para outro *ADVService* ou *ADOService*.

A figura 4.12 mostra um exemplo do que foi dito.

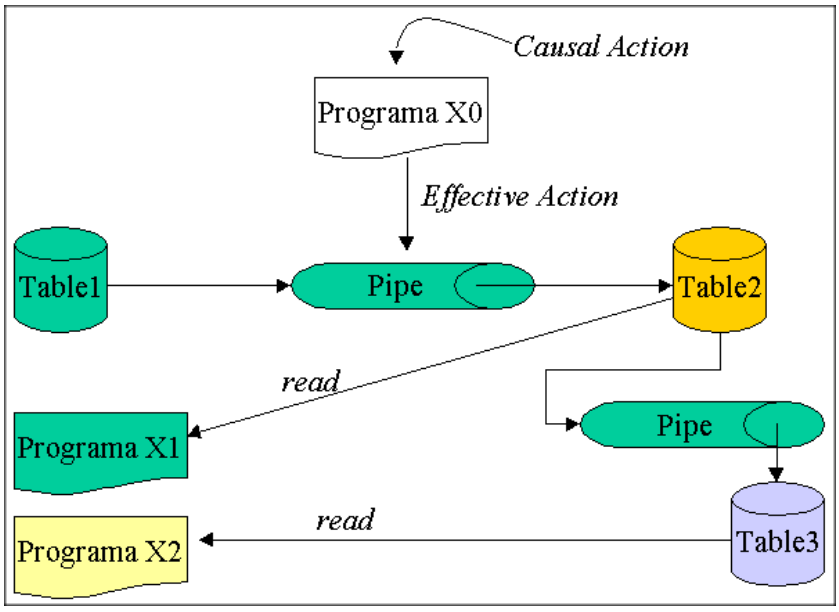

Figura 4.12 - Propriedade: tipos de mídia

A figura 4.12 mostra-nos um exemplo em que, por meio de um *ADVService* inicial, uma ação efetiva é gerada e, a partir desta, uma série de eventos é acionado. O *ADVService* que é representado pelo Programa *X0* executa, inicialmente, uma transformação de uma estrutura de tabela em um banco de dados para uma segunda tabela. Esta será o ponto de entrada e/ou de consulta para um outro *ADVService* chamado Programa *X1* que utiliza a saída gerada pelo *ADOService* conectado ao *ADVService X0*. Este, por sua vez, realiza um processo análogo ao anterior de forma a gerar uma nova saída que servirá aos propósitos de outro *ADVService* chamado Programa *X2*.

Toda essa possibilidade de manipulação de estruturas de banco de dados baseia-se em um *Design Patterns* tipo *Pipes and Filters* [Buschmann et alli 1996] que possibilita a manipulação de dispositivos de diferentes tipos. Ou seja, o *ADVService* está servindo de interface entre mídias distintas, como já havia sido ponderado na descrição inicial da teoria de *ADVs*/*ADOs* que foi adaptada para *ADVServices*/*ADOServices*.

4.7 *ADVServiceChart*: Uma notação gráfica para *ADVServices* / *ADOServices*

Esta seção propõe uma extensão para o trabalho sobre *ADVcharts* [Carneiro, Cowan and Lucena 1993a], [Carneiro, Cowan and Lucena 1993b], [Cowan, Barbosa, Ierusalimschy, Lucena and Oliveira 1992] e [Cowan, Mackie, Pianosi and Smit 1991]. *ADVcharts* oferecem um esquema visual para a especificação interativa de sistemas baseados em *ADVService*s/*ADOService*s. *ADVcharts* são baseados em máquinas de estados finitos e também são uma extensão de *Statecharts* [Harel 1987], *Objectcharts* [Coleman, Hayes and Bear 1992] e Redes de Petri [Reisig 1985].

As interfaces de usuários são usualmente construídas baseadas na composição de comportamento e estrutura. Por exemplo, o comportamento da interface do usuário, quando este inicia uma ação, pode ser visto como a composição de simples *scripts* tanto quanto de complexas rotinas. Assim sendo, a estrutura de uma interface de usuário é composta de um variado número de objetos[3] virtuais. Por exemplo, uma janela de diálogo com o usuário geralmente possui um campo texto no qual apresenta-se a mensagem, um campo botão no qual o usuário seleciona uma determina ação tipo "*ok*" ou "*cancel*", e, em alguns casos, campos tipo "*checkbox* ou *list box*".

ADVServiceChart provê mecanismos para que o aninhamento tanto de comportameto como de estrutura possa ser realizado de forma gráfica [Carneiro, Cowan and Lucena 1993a].

Na próxima seção, a semântica e os diagramas de modelagem serão explicados fazendo-se uso de exemplos.

4.8 Exemplo

O exemplo a seguir, mostra um pequeno serviço na Web que informa a hora local (da máquina do usuário). A informação pode ser exibida dentro de um *browser* em dois formatos, analógico e digital.

A idéia deste exemplo é permitir que uma aplicação utilize um serviço já existente na Web e que se possa obter vários *roles* para a interface do usuário. Neste exemplo, a hora será exibida em formato digital e analógico sem que o serviço tenha de sofrer

[3] A palavra objeto, neste caso, não tem nenhuma relação com objeto dentro da ótica de orientação a objetos.

mudanças. A figura 4.13 mostra o exemplo da interface que foi implementado, onde podemos encontrar de forma esquemática uma representação do sistema a ser desenvolvido na qual pode-se visualizar um *ADOService* e dois *ADVServices* que são *views* do serviço que fornece a hora.

A figura 4.13 mostra a interface final para o usuário.

De forma esquemática, a figura 4.14 mostra-nos como será o modelo de interação de nosso exemplo.

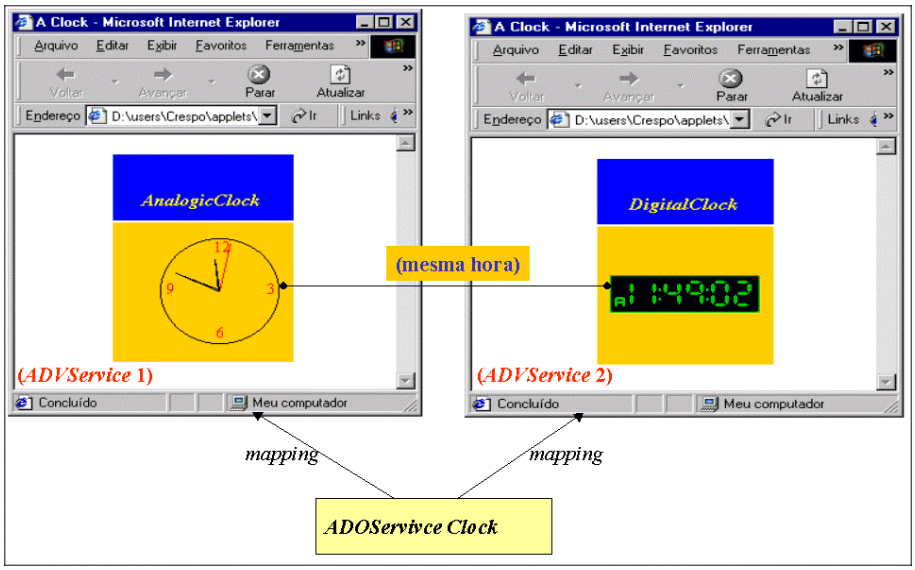

Figura 4.13 - Serviço de hora

Nesta figura, dois *ADVServices* mapeam um *ADOService* que fornece a hora, minutos e segundos aos seus *viewers*. Os *ADServices*, por sua vez, transformam esta informação recebida em saídas distintas para o usuário, mantendo a consistência tanto vertical quanto horizontal, pois, neste caso, um mesmo *ERole* foi desempenhado por ambos os *ADVServices*.

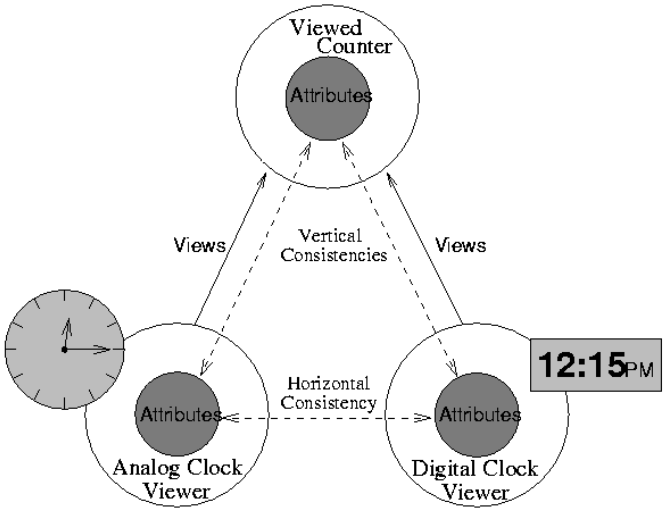

Figura 4.14 - modelo de interação

Na figura 4.13, podemos ver como é apresentada a hora ao usuário. A representação da hora dá-se em dois formatos, analógico e digital, porém ambos os *ADVServices* são consistentes com o mapeamento que fazem sobre o *ADOService* e, por sua vez, são consistentes entre si, na medida em que exibem sempre a mesma hora.

O *Design* desta interface usando a notação de *ADVServiceChart* está representado na figura 4.15.

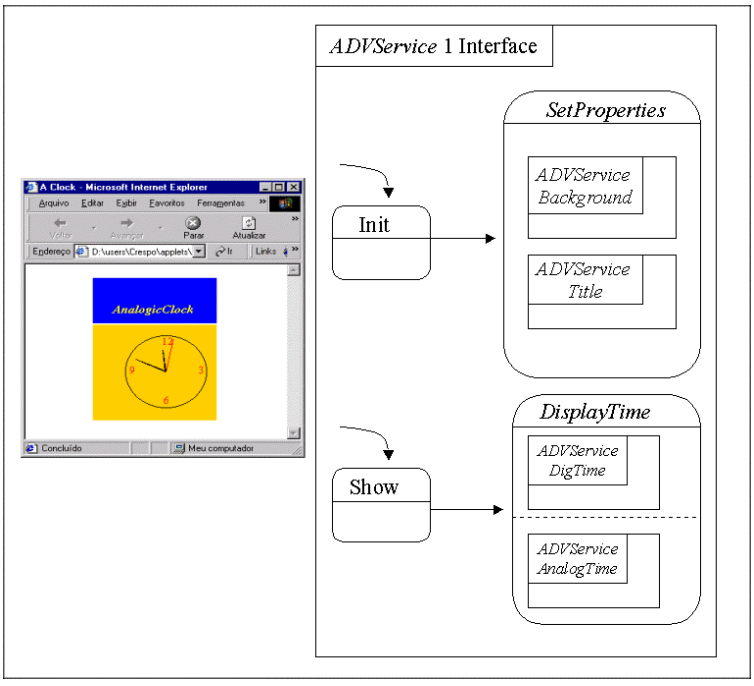

Figura 4.15 - *ADVServiceChart*

Como pode ser visto na figura acima, dois eventos são disparados no momento em que o serviço é requisitado: *Init* que possui dois *AVDService* (*Title* e *Background*).

Title prepara o fundo da região da tela onde será exibido um título em uma determinada cor. Já *Background* prepara o fundo da tela onde será mostrada a hora em um determinado formato.

O evento *Show* chama o evento *DisplayTime* que possui um *design* com um operador *OR* para os *ADVService DigTime* e *Analogtime*, ou seja, somente um dos dois será acionado por intervalo de tempo. Os dois *ADVServices* não podem ser executados ao mesmo tempo. A figura 4.16 mostra onde identificar o operador *OR* no *design* (é uma linha tracejada)

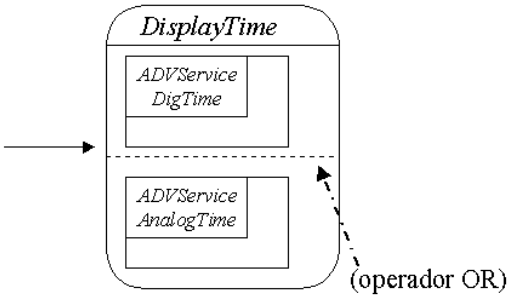

Figura 4.16 - operador *OR* no *ADVServiceChart*

A descrição do *ADOService* chamado *ADOService Clock* é mostrada a seguir utilizando-se a linguagem script *JavaScript* [Flanagan 1998].

```
function clock () //JavaScript [ADOService]
{
//Declarations
//Data Signature [nome]
//Attributes
    Private var now = new Date();
    Private var yr = now.getYear();
    Private var mName = now.getMonth() + 1;
    Private var dName = now.getDay() + 1;
    Private var dayNr = ((now.getDate()<10) ? "0" : "")+ now.getDate();
    Private var ampm = (now.getHours() >= 12) ? " P.M." : " A.M."
    var hours = now.getHours();
//Static Properties
// Constraints
    hours = ((hours > 12) ? hours - 12 : hours);
    var minutes = ((now.getMinutes() < 10) ? ":0" : ":") + now.getMinutes();
    var seconds = ((now.getSeconds() < 10) ? ":0" : ":") + now.getSeconds();
//Derived Attributes
   // String to display current time.
   var TimeValue =(" "
   + hours
   + minutes
```

```
     + seconds
     + " "
    + ampm);
//Dinamic Properties
// Behavior [Displays the current time.]
 document.forms[0].elements[0].value = TimeValue;
 timerID = setTimeout("time()",1000);
 timerRunning = true;
}
```

A descrição dos *ADVServices* não está presente em virtude do código ser muito extenso para a apresentação gráfica da hora tanto em formato digital como analógico.

4.9 Conclusão

Uma das grandes dificuldades para se obter a reutilização do *design* e da implementação é o fato de tanto o *design* como a implementação acumularem detalhes sobre o ambiente ao qual pertencem, ocasionando um grande acoplamento entre os módulos e, desta forma, dificultando o seu reuso por outros módulos, uma vez que sua interface interna está profundamente comprometida com alguma estrutura de dado.

Para que *WebFrameworks* possam gerar aplicações a partir de um conjunto básico de serviços, o uso de *roles* torna-se fundamental.

Este capítulo apresentou uma teoria que expressa *roles* no nível de *design*. *Roles* são equivalentes à notação de *ADVServices/ADOServices*, que foi criada para especificar de forma clara e formal a separação da interface do usuário da aplicação em que ela está inserida, oferecendo um método de *design* sistemático, independente da aplicação específica.

A conclusão final é de que *WebFrameworks* podem ser bem projetados mediante o uso da notação de *ADVServices*/*ADOServices* pelos seguintes fatores:

- A notação de *ADVServices*/*ADOServices* mostrou-se consistente e adicionou novas funcionalidades e ou características ao modelo de *roles* visto no Capítulo 3.
- A dinâmica proporcionada pela notação de *ADVServices*/*ADOServices* permitiu a criação dos conceitos de *roles* dinâmicos e estáticos que permitem que um determinado *ADVService* possa selecionar e criar em tempo de execução o *role* mais adequado para um certo momento e desempenhá-lo com o seu *ADOService* correspondente. Desta forma, um conjunto de visões diferenciadas podem fazer

parte do escopo de um *ADVService*, sendo que, antes desta abordagem, somente se mencionava um *role* para um observador, sendo este *role* previamente criado.

- A possibilidade de um *ADVService* trabalhar na plataforma Web foi também tratada pela notação tanto de forma síncrona como assíncrona com a novidade da persistência de variáveis de *CGI* mediante a utilização de um *Blackboard* associado aos *ADOServices* que pertencem a um servidor *ADOService* WWW de informação.
- Também foi apresentada uma notação gráfica para o *design* de *ADVServices/ADOServices* chamada *ADVServiceChart* baseada na teoria de *ADVCharts*. Esta notação permite que um *design* possa ser construído utilizando-se composição de estruturas e comportamento no momento em que se especificam os mais variados tipos de eventos.
- Permitirem uma forte reutilização de *design*, *separation of concerns*.
- Dinâmica para a criação e desempenho de *roles,* estendendo o seu conceito original, permitindo a criação de *roles* estáticos e dinâmicos (em tempo de execução).
- Para a expressão gráfica do *design* de *WebFrameworks*, *ADVChart* foi transformado em *ADVServiceCharts*, que, no mundo da Web, desempenha o mesmo papel que o da notação *UML* na especificação de *frameworks* orientados a objetos [Fontoura 1999].
- A teoria de *ADVs*/*ADOs* permite expressar o conceito de *roles* no nível de *design* de forma detalhada, caracterizando todas as suas propriedades neste nível, permitindo o *design* detalhado de aplicações concretas.
- A notação de *ADVServices*/*ADOServices* proporcionou a conversão da teoria de *ADVs*/*ADOs* para o mundo da Web.

Capítulo 5

WebCompose: uma linguagem de *script* específica de domínio baseada em *framelets* para a composição de *WebFrameworks*.

Resumo

Este capítulo apresenta uma linguagem específica de domínio (*DSL*), chamada *WebCompose*, que utiliza uma abordagem de *framelets* para a sua construção e é baseada na teoria de *ADVServices* e *ADOService*. É feita uma comparação com outras linguagens que tratam problemas similares. *WebCompose* permite a construção e composição de serviços de *WebFrameworks*.

5.1 Introdução

Nos últimos 15 anos, uma fundamental mudança vem acontecendo na área de desenvolvimento de *software*. Essa mudança é a transição de linguagens de programação convencionais tais como *C* e *C++* para linguagens de *script*: *Perl, Tcl*, *Rexx*, *Visual Basic* e *Unix Shell*, *Python*,entre outras. As linguagens de *scripts* representam vários estilos de programação. Elas assumem que já existe uma coleção de componentes utilizáveis programados em outras linguagens, e que é possível combiná-los em futuras implementações para estender algumas das características dos componentes já disponíveis. Em virtude desta característica, elas, muitas vezes, são conhecidas como *glue language* ou linguagens de integração de sistemas [Ousterhout 1998].

Scripts especificam como os componentes podem ser "*plugados*". Pensar em *script* é pensar em atores interpretando vários papéis [Achermann, Limpe, Schneider and Nierstrasz 2000]. A figura 5.1 ilustra o que foi dito.

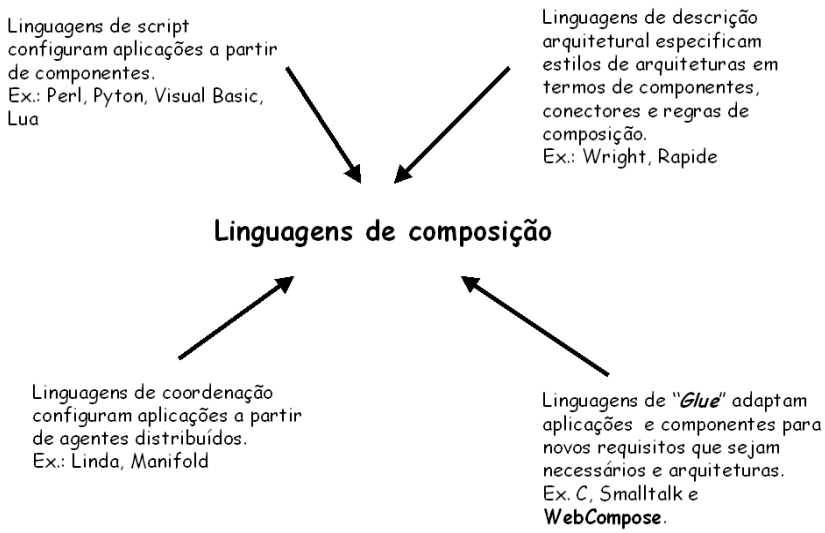

Figura 5.1 - Linguagens de composição

[Achermann, Limpe, Schneider and Nierstrasz 2000]

Com o crescimento da *Internet*, as linguagens de *script* vêm se popularizando. A *Internet* pode ser vista como uma grande ferramenta de *glue*! A linguagem ideal nesta área é aquela que permite conectar mais facilmente os componentes para trabalharem juntos e também para manipulação de *frameworks* [Ousterhout 1998].

Com base nestas considerações, apresentaremos a linguagem *WebCompose*, que é uma linguagem de *script* que proporciona a colaboração e o reuso de serviços já existentes na plataforma Web. *WebCompose* também oferece mecanismos para composição de serviços de forma a gerar outros com mais funcionalidade.

Será também descrito o ambiente em que a *WebCompose* foi desenvolvido, sua arquitetura, comandos e uma comparação com outras abordagens.

5.2 Ambiente de desenvolvimento

Para o desenvolvimento do protótipo foi utilizado o sistema operacional *Windows NT Server* 4.1, usando o servidor *Internet Information System* (**IIS**) da *Microsoft*. A linguagem de *Script* utilizada para a manipulação de formulário é *CGILua* [Hester, Borges and Ierusalimschy 1998a] [Hester, Borges and Ierusalimschy 1998b] e *Lua* [Ierusalimschy, Figueiredo, Celes 1996]. A seguir, descreveremos como funciona a arquitetura *Internet* baseada em *CGI* e a ferramenta *CGILua* juntamente com a linguagem *Lua*.

5.2.1 Introdução à linguagem Lua

Lua é uma linguagem de programação poderosa e leve, projetada para estender aplicações, e freqüentemente usada como uma linguagem de propósito geral. Ela combina programação "*procedural"* (com sintaxe semelhante à de Pascal) com poderosas construções para descrição de dados, baseadas em tabelas associativas e semântica extensível. *Lua* é *tipada* dinamicamente, interpretada a partir de *bytecodes*, e tem gerenciamento automático de memória com coleta de lixo. Essas características a tornam uma linguagem ideal para configuração, automação (*scripting*) e prototipagem rápida.

Um conceito fundamental da linguagem é fornecer meta-mecanismos para a implementação de construções, ao invés de fornecer um número grande de construções diretamente na linguagem. Por exemplo, embora ela não seja uma linguagem puramente orientada a objetos, fornece meta-mecanismos para a implementação de classes e herança, os quais trazem uma economia de conceitos e mantêm a linguagem pequena, ao mesmo tempo que permitem que a semântica seja estendida de maneiras não convencionais. Ter uma semântica estensível é uma das suas características mais marcantes. *Lua* está implementada como uma pequena biblioteca de funções C, escritas em *ANSI C*, que compila sem modificações em todas as plataformas conhecidas. Os objetivos da implementação são simplicidade, eficiência, portabilidade e o baixo impacto na manutenção das aplicações[*Lua* Site] [Ierusalimschy, Figueiredo, Celes 1996].

5.2.2 Introdução sobre CGILua

CGILua é uma ferramenta para o desenvolvimento de páginas *HTML* dinâmicas e para a manipulação de dados por meio de formulários. Ela usa a linguagem interpretada Lua para codificar os comandos, os quais são executados no momento em que a página é exibida.

CGILua é simples, porém com grande poder de processamento. Conseqüentemente, proporciona o trabalho de tarefas complexas com muito pouco esforço. O tipo mais freqüente de aplicações construídas com *CGILua* são sistemas de informações baseados na Web, utilizando acessos a banco de dados, e *Gateways* WWW.

Tecnicamente, *CGILua* é um programa *CGI* que executa sobre um servidor e recebe arquivos para serem interpretados. Ela atua como um intermediário entre os *scripts* submetidos a um servidor *HTTP*, realizando todo o processamento, tal como

decodificar dados de entrada, e também oferece recursos adicionais tais como funções de criptografia, acesso a base de dados, *upload* de arquivos e outros[CGILua Site]. Dentre as principais funcionalidades proporcionadas pelo *CGILua* podemos citar: múltiplos paradigmas de programação de *scripts*, extensibilidade, portabilidade, segurança, simplicidade e flexibilidade[CGILua Site] [Hester 1999].

5.2.1 Arquitetura de Servidores WWW e CGI

A *World Wide Web* (WWW) é um grande sistema distribuído, construído sobre o conjunto de protocolos *TCP/IP*, que obedece a uma organização de elementos baseada essencialmente no paradigma cliente-servidor [Hester 1999].

Quando um servidor HTTP atende a uma requisição por uma página estática, ele apenas traduz a *URL*(*Uniform Request Locator*) para um caminho físico e recupera o arquivo correspondente do sistema de arquivos. A página requisitada é, então, enviada para o cliente junto com informações a respeito do tipo *MINE* do documento, o tamanho do arquivo, a data da última alteração, etc.. Para atender a uma requisição por uma página dinâmica, o servidor precisa identificar qual programa deve ser acionado e para depois repassar os dados recebidos na requisição para o programa encarregado do processamento. O servidor aguarda, então, o término do processamento e envia os resultados do programa de volta para o cliente que fez a requisição. Com o propósito de estabelecer regras de como deve ser realizada a comunicação de dados e a interação entre o servidor *HTTP* e os programas invocados por ele, são estabelecidos padrões de interface, as *Applications Programming Interfaces(APIs*). O padrão mais antigo e mais popular entre todos os servidores *HTTP* é o *CGI*, que proporciona um alto grau de portabilidade para os programas que a utilizam [Hester 1999].

Na interface *CGI*, quando o servidor *HTTP* recebe uma requisição a ser processada por um programa externo, um novo processo é criado para executar o programa *CGI*. Os dados da requisição *HTTP* são repassados para o programa por meio de variáveis do ambiente e da entrada padrão do processo (*standard input*). O programa *CGI* executa, então, o processamento necessário e transfere o resultado para o servidor *HTTP* pela saída padrão (*standard output*). Quando recebe o resultado do processamento, o servidor elabora a resposta *HTTP* ao cliente, repassando integralmente o que é gerado pelo programa *CGI*. No caso do servidor receber requisições simultâneas, tratadas pelo mesmo processo *CGI*, vários processos do

programa são executados simultaneamente, sem nenhuma cooperação ou interferência das múltiplas instâncias, já que cada instância é executada em um processo separado, e o modelo não estabelece nenhuma forma de comunicação entre processos. Os programas *CGI* relacionados a uma mesma aplicação, por exemplo, são executados de forma completamente independente e não compartilham dados entre si. O programador é responsável por implementar algum mecanismo de passagem de dados, se for necessário, para a lógica da aplicação [Hester 1999].

5.3 *WebCompose*: uma linguagem específica de domínio baseada em *framelets* para composição de serviços em *WebFrameworks*.

A linguagem proposta foi desenvolvida utilizando-se o paradigma de *framelets* [Pree 1999a] [Pree 1999b], onde cada comando comporta-se como um pequeno *framework*, com seus pontos de flexibilização definidos e com duas formas de utilização que são: *black-box* e *white-box*.

A linguagem *WebCompose* também pode ser vista como um grande *toolkit* para o ambiente *CGILua*. A linguagem está baseada na teoria de *ADVServices/ADOServices* na medida em que seus comandos permitem satisfazer todas as propriedades da teoria apresentada no Capítulo 4. O fato das propriedades serem satisfeitas pelo conjunto de comandos proposto na *DSL WebCompose* serve como critério para avaliar a completude da linguagem. Em outras palavras, todas as situações de composição previstas na versão *ADVService/ADOService* da teoria de *ADVs/ADOs* estão contempladas na linguagem.

Os comandos da linguagem *WebCompose* fornecem uma série de recursos para a criação de restrições, manipulação de formulários, banco de dados, utilização de *pipes* and *filters* e serviços de *proxy* entre serviços. A *WebCompose* visa facilitar a construção e a manutenção de *sites* complexos, na medida em que novos serviços são necessários, pode-se compô-los com outros já existentes, sem que seja necessário refazer ou mesmo re-implementar os serviços já existentes.

WebCompose tem por objetivo permitir a utilização de serviços já existentes na Web e que não foram necessariamente projetados para atuarem em conjunto ou cooperarem com outros serviços. O usuários potenciais da *DSL WebCompose* são *WebMasters*.

5.3.1 Estrutura da DSL *WebCompose*

A *WebCompose* está estruturada como um *toolkit* para o ambiente *CGILua*. Desta forma, ela pode transformar um servidor *CGI* que utiliza a ferramenta *CGILua* para processar *scripts* de páginas Web em um Servidor *ADOService* WWW que permite a composição e reutilização de serviços presentes na Web. A figura 5.2 apresenta a arquitetura da linguagem em conjunto com todo o ambiente WWW.

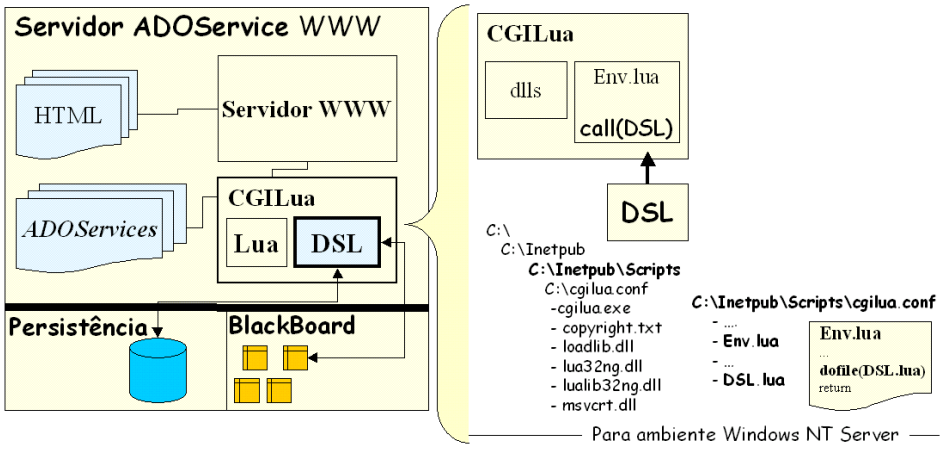

Figura 5.2 - Integração *DSL* e *CGILua*

Como pode ser visto na figura 5.2, a ferramenta *CGILua* atua como uma linguagem de *scripts* para manipulação de formulários via Web. *CGILua* permite também o uso da linguagem *Lua* para a confecção de *scripts*.

A *Webcompose* foi desenvolvida para ser utilizada em conjunto com *CGILua* e *Lua*. Desta forma, o processo de criação de *scripts* direcionados à utilização de serviços na Web fica muito facilitado. As facilidades da linguagem permitem definir vários critérios de restrição, a utilização de diversos *Design Patterns* para permitir composição em *WebFrameworks* e a reutilização de diversos serviços presentes na Web.

A *DSL WebCompose* foi desenvolvida utilizando-se a linguagem *Lua* e permitindo uma extensão da ferramenta *CGILua*. Outra característica importante é a disponibilização de um *Blackboard* no Servidor *ADOService* WWW que permite que dados sejam compartilhados por diversos serviços de forma assíncrona. Este tópico será melhor detalhado mais adiante.

A seguir, será apresentado a sintaxe abstrata da *WebCompose*.

5.3.2 Sintaxe abstrata da *WebCompose*

Na sintaxe abstrata de uma linguagem de programação são estabelecidos os relacionamentos hierárquicos entre as frases e subfrases semanticamente relevantes que compõem a linguagem.

<SCRIPT>	::= <CMD>
<CMD>	::= <CMD> <CMD> | <CONSTRAINT> | <EVALUATE> <DBQUERY> | <DBPIPE> | <DBDELETETABLE> | <PROXY> | <SLICE> | <SENDTOBLACKBOARD> | <LOADFROMBLACKBOARD> | <GARBAGEBLACKBOARD> | Λ
<CONSTRAINT>	::= CONSTRAINT <NUM_SEQ> <EXP> <VAR_RET>
<EVALUATE>	::= EVALUATE <NUM_SEQ> <DESVIO> <DESVIO>
<DBQUERY>	::= DBQUERY <ID_BD> <QUERY> <DESVIO> <CONS><NUM_SEQ> <DESVIO><DESVIO>
<DBPIPE>	::= DBPIPE <ID_BD> <ID_TABLE> <CAMPO> <ID_TABLE><CAMPO> | DBPIPE <ID_BD> <ID_TABLE> <ID_SCRIPT>
<DBDELETETABLE>	::= DBDELETETABLE <ID_BD> <ID_TABLE>
<PROXY>	::= PROXY <VAR><ID_MET><ID_SERVER>
<SLICE>	::= SLICE <URL> <STRING> <STRING>
<SENDTOBLACKBOARD>	::= SENDTOBLACKBOARD <ID_VAR> <ID_SCRIPT> <TIMEOUT>
<LOADFROMBLACKBOARD>	::= LOADFROMBLACKBOARD <ID_VAR> <TIMEOUT>
<GARBAGEBLACKBOARD>	::= GARBAGEBLACKBOARD
<DESVIO>	::= - | ID_SCRIPT
<CONS>	::= #CONS
<CAMPO>	::= <CAMPO> <CAMPO> | <ID_CAMPO><ID_TIPO> | Λ
<ID_SCRIPT>	::= <ID>.LUA
<VAR>	::= <VAR> <VAR> | ID_VAR | Λ

Tabela 5.1 - BNF da *DSL WebCompose*

A tabela 5.1 descreve a sintaxe abstrata do conjunto restrito da linguagem, onde são apresentados os relacionamentos hierárquicos de frases e subfrases. Esta especificação foi escrita em notação BNF (*Backus Naur Form*), onde é estabelecido um conjunto de regras de produção. Cada regra de produção está separada pelo meta-símbolo '::=',

que pode ser lido como '*consiste de*' ou '*é definido como*'. O lado esquerdo consiste de um símbolo não terminal entre '<' e '>'. Cada não terminal é o nome de um tipo de construtor ou categoria sintática da linguagem sendo especificada. O lado direito da regra consiste de uma ou mais especificações alternativas, separadas pelo meta-símbolo '|' (lido como '*ou*'). Cada uma destas alternativas é uma seqüência de símbolos não terminais e/ou terminais, onde um símbolo terminal é um *token* (caracter ou grupo indivisível de caracteres) da linguagem.

A seguir será apresentado os comandos da *WebCompose* com maiores detalhes.

5.3.3 Comandos da *WebCompose*

Os comandos da linguagem atuam em conjunto com os comandos do *CGILua* e *Lua* de forma a dar maior poder de processamento a quem estiver criando um *script*. A relação dos comandos da *WebCompose* são:

Nome do Comando	Descrição
Constraint	Define restrições por meio de expressões aritméticas.
Evaluate	Avalia uma *Constraint* definida.
DBQuery	Executa de uma *query* configurando o laço principal de execução. Pode ter a sua execução condicionada por alguma *Constraint*.
DBPipe	Realiza a transformação de tabelas dentro de um banco de dados definindo campos e redefinir os tipos.
DBDeleteTable	Exclui uma determinada tabela de um banco de dados.
Proxy	Define a partir de uma única interface de dados múltiplas saídas para outras interface de serviços.
Slice	Extrai um pedaço de uma página *HTML*.
SendToBlackBoard	Envia dados para o *BlackBoard*.
LoadFromBlackBoard	Recupera dados do *BlackBoard*.
GarbageBlackBoard	Aciona o coletor de lixo que atua no *BlackBoard*.

Tabela 5.2 – Resumo dos comandos da WebCompose

A seguir, serão descritos com mais detalhes os comandos da linguagem *WebCompose*.

Comando: **Constraint**

Sintaxe**:**

Constraint(< número da seqüência [1a]>,<expressão>[2a],<váriável de retorno lógica>)

Este comando permite que se possa definir um conjunto de restrições para a execução de algum trecho ou comandos da linguagem *WebCompose*, bem como para comando da linguagem presente no *CGILua* e Lua.

Propriedade de *ADVService/ADOService* satisfeita:

- **Comunicação**: A utilização do comando *Constraint* permite que *scripts* que estejam na mesma seção possam estabelecer uma comunicação síncrona por meio da restrição de execução de alguns processos. Quando uma composição em um *script* é realizada, pode ser necessário algum tipo de comunicação síncrona entre o *ADVService* principal e o que foi encapsulado. A utilização de *Constaints* viabiliza este tipo de comunicação.

Os vários *hot-spots* do comando são descritos, a seguir, na tabela 5.3:

Hot-spot	**Descrição**
hot-spot[1a]: número de seqüência	Define um número que está associado a uma restrição, logo, pode-se definir um conjunto de restrições que poderão ser avaliadas no decorrer da elaboração do *script*. Este comando não avalia, inicialmente, a restrição proposta. Estas restrições são empilhadas para posterior uso.
hot-spot[2a]: expressão	Este é um dos pontos de flexibilização do comando, atuando de *forma black-box*, ou seja, parametrizado. Neste parâmetro, exige-se uma expressão contendo operadores aritméticos e ou lógicos e utilizando qualquer variável local, pública ou proveniente de qualquer operação tipo *CGI*. As variáveis provenientes de operações *CGI* possuem a seguinte sintaxe: cgi.<nome da variável do formulário do cliente>.
variável de retorno lógica	Esta variável é atualizada sempre que uma determinada *Constraint* for avaliada. O conteúdo da variável é lógico (tipo *true* ou *false*) e permite que ela seja testada no decorrer do *script* após ter sua *constraint* avaliada.

Tabela 5.3 – *hot-spots* do domando *Constraint*

Exemplo:

Constraint(1,"valor_fin>10",”RetVal")

```
-- Trecho do script para autenticação de usuários em um site, utiliza uma restrição quanto ao tamanho
-- do campo de senha  se o tamanho for maior que 7 então gera  uma exceção de segurança.
$debug
DBOpen("DSN=kstudio") --abrindo a base da dados

. . .

Constraint(1, "strlen(cgi.senha)>7","Var_ExcSenha")

. . .

return
```

Para que uma dada restrição seja ativada, o comando *Evaluate* deve ser utilizado.

Comando: **Evaluate**

Sintaxe**:**

Evaluate(<número da seqüência >, [scriptYes|-][1], [scriptNo|-] [2])

Este comando permite acionar uma determinada restrição. Após a utilização do *Evaluate*, a variável lógica definida no comando *Constraint* fica disponível para que possa ser utilizada dentro de qualquer parte do *script*. Este comando pode estar condicionado ao resultado da avaliação da expressão presente na restrição.

Propriedade de ADVService/ADOService satisfeita:

- **Comunicação**: Assim como o *Constraint*, o comando *Evaluate* possibilita a comunicação síncrona entre *scripts*. O comando *Constraint* define expressões para serem avaliadas, e o *Evaluate* é o responsável por estabelecer a conexão síncrona entre os *scripts* de forma a determinar a execução ou não de alguns trechos de *script*.

A seguir, os vários *hot-spots* do comando *Evaluate* são descritos na tabela 5.4.

Sendo que:

Hot-spot	**Descrição**
Número da seqüência	Determina que restrição avaliar.
hot-spot [1] e [2]	Permite determinar desvios da execução do *script* dependendo do resultado da avaliação da expressão contida no comando *Constraint*. Estes *hot-spot* podem ou não ser utilizados. Se forem

	utilizados, tanto o *hot-spot*[1] como o *hot-spot*[2], ou ambos, um desvio produzir-se-á no decorrer do *script*. Caso nenhum deles tenha sido definido, então a execução do *script* ocorrerá de forma normal, passando a executar a próxima linha existente. Quando não for necessário utilizar algum dos *hot-spots*, deve-se entrar com o valor: "-".

Tabela 5.4 – *hot-spots* do comando *Evaluate*

Exemplos:

Constraint(1,"valor_fin>10",”RetVal")

Evaluate(1, "-","-")

Neste caso, está sendo feita somente a avaliação do resultado da restrição número 1 e deixando disponível para todo o *script* a variável RetVal com um valor lógico (*true* ou *false*), dependendo do resultado da expressão.

```
-- Trecho do script para autenticação de usuários em um site, utiliza uma restrição quanto ao tamanho
-- do campo de senha. Se o tamanho for maior do que 7 dígitos então gera  uma exceção de segurança.
$debug
--abrindo a base da dados
DBOpen("DSN=kstudio")

. . .

Constraint(1, "strlen(cgi.senha)>7","Var_ExcSenha")

. . .

Evaluate(1, "-", "ErrorPass.lua")
```

Neste exemplo, se o resultado do *Evaluate* for falso, então o *script* sofrerá um desvio para o *script ErrorPass.lua,* que tratará o problema de senha inválida.

Comando: **DBQuery**

Sintaxe**:**

DBQuery(<DSNName[1]>,<query[2]>,<LoopControl[3]>,[#*Cons*],<*seq*>,<TrueEscape>, <FalseEscape> [4])

Este comando executa uma determinada *query* em um banco de dados que esteja no Servidor *ADOService* WWW. O *DBQuery* possui um conjunto de *hot-spots* que atua de forma *black-box* e *white-box*. Também é possível condicionar a sua execução a uma determinada restrição.

Propriedades de ADVService/ADOService satisfeitas:

O comando *DBQuery* satisfaz as seguintes propriedades:

- **Consistência vertical:** verifica-se sempre que o comando da linguagem conseguir responder ao cliente de forma satisfatória.

A consistência vertical em caso de aplicações na Web é um processo que envolve a participação direta do usuário, na medida em que ele dá o aval sobre o que foi retornado pelo serviço requisitado. A figura 5.3 exemplifica o processo.

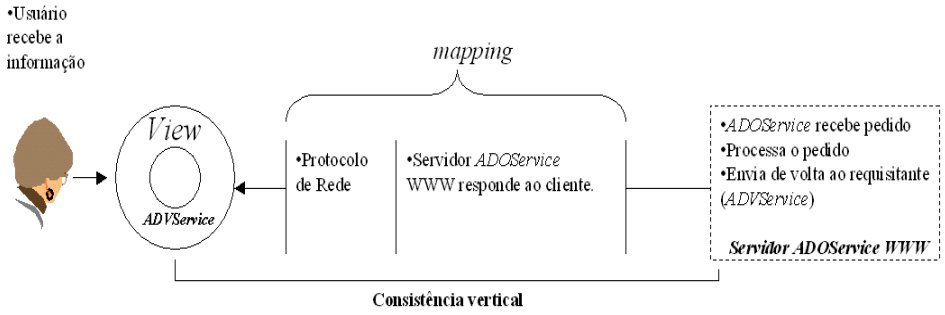

Figura 5.3 - Consistência vertical

Suponha que o serviço realiza corretamente a sua tarefa. Neste caso, do momento em que o usuário requisita o serviço até o seu retorno, alguns passos fazem parte do processo de consistência vertical:

- Envio do pedido ao *ADOService;*
- Processamento do pedido;
- A resposta dada pelo servidor *ADOService* WWW ao cliente que requisitou o pedido;
- Protocolo de Rede *TCP/IP* (*Internet*).

- **Criação:** a propriedade de criação trata da maneira pela qual um *role* desempenha o seu papel, que pode ser estático ou dinâmico. O comando *DBQuery* oferece todos os requisitos para atuar de forma estática e dinâmica. Quando o comando é utilizado para um conjunto de *scripts* nos quais os seus parâmetros são todos fixos e o *hot-spot*[3] não é utilizado, ele faz uso da propriedade de criação desempenhando um papel estático, pois para todas as ocorrências da aplicação com o comando configurado sempre da mesma forma, a execução é desempenhada sempre da mesma forma. No caso da necessidade de uma utilização mais flexível, ou seja, na qual os parâmetros podem mudar dependendo de algum estado de variáveis que estejam na memória ou que estejam no

Blackboard, ele desempenha um papel dinâmico. Este papel faz com que resultados diferentes possam ser vistos em diversas ocorrências de aplicações nas quais o comando é utilizado. Outra situação em que o comando é dinâmico é a necessidade de se utilizar *o hot-spot*[3] para controle do laço de operação principal do próprio *DBQuery*. Este controle é muito particular e está muito relacionado ao tipo de tabela em que atuará, podendo ter um conjunto variado de comportamento no decorrer de seu processamento. Apesar da particularidade do uso do *hot-spot*[3] ser identificado, ou seja, um *script* externo é utilizado para o processamento (e isto é fixo), podendo dar a impressão do comando estar desempenhando um papel estático, a dinâmica da operação está, justamente, no conteúdo da tabela pertencente a um determinado banco de dados. Este fato já ilustra a característica dinâmica dos bancos de dados, no qual não se pode prever com antecedência o conteúdo e a quantidade de registros existentes. Com isso, as várias aplicações que utilizam este comando na *forma white-box*, ou seja, usando o *hot-spot*[3], poderão ter respostas distintas que não configuram um erro, mas sim o fato que a tabela foi atualizada ou teve registro excluídos.

- **Composição:** verifica-se sempre que o comando necessita utilizar o *hot-spot*[3], para poder desempenhar corretamente o seu papel. Para isso, é chamando um *script* externo. A figura 5.4 mostra o esquema deste *hot-spot*.

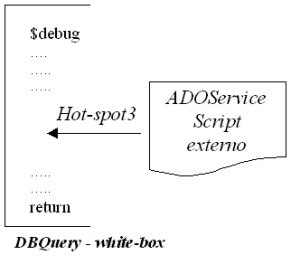

Figura 5.4 – *DBQuery* com uso *white-box*: esquema interno

Na figura 5.4, podemos ver que, quando se utiliza o *hot-spot*[3], o comando fica com características *white-box*, deixando todo o poder de processamento do laço principal a cargo do usuário. A tabela 5.5, a seguir, descreve os *hot-spots* do comando:

Hot-spot	**Descrição**
hot-spot[1]: <DNSName>	Define o nome do banco de dados que está relacionado ao *drive ODBC* do servidor.
hot-spot[2]: <query>	Define uma determinada *query* para ser submetida ao banco de dados.

hot-spot[3]: <LoopControl>	Quando este *hot-spot* é utilizado, o comando fica com características *white-box*. Este *hot-spot* permite que o usuário detenha o controle do laço de execução sobre as chamadas aos vários registros resultantes da execução da *query*.
hot-spot[4]: [*#Cons*], <*seq*>, <TrueEscape>, <FalseEscape>	Se presente, a *string* #Cons indica que a execução da *query* fica condicionada a uma determinada restrição. Uma vez definida essa necessidade, torna-se obrigatório informar qual o número(seq) da restrição que deve ser analisada. As opções <*TrueEscape*> e <*FalseEscape*> são opcionais e permitem desviar a ação da *query* dependendo do resultado da restrição. No último caso, a *query* nem seria executada no banco de dados.

Tabela 5.5 – *hot-spots* do comando *DBQuery*

Na figura 5.5, podemos ver um pequeno exemplo sem o uso da *DSL*

```
--open data base
DBOpen("DSN=Viewpoints")
VarId = cgi.ClassId
str=" SELECT * FROM ViewpointClass WHERE ID ="..tonumber(VarId)
-- executando a base da dados
DBExec(str)
-- capturando uma tupla da tabela
tabela = DBRow()
if (tabela == nil) then
  write("Nenhum resultado para a sua query.")
  return
end
while (tabela ~= nil) do
  ...
  write(tabela[0])
  ...
  write(tabela[1])
  write(tabela[2])
  write(tabela[3])
  ...
  tabela = DBRow()
end
-- fechando a base de dados
DBClose()
return
```

Figura 5.5 - Pequeno exemplo

A figura 5.5 mostra um exemplo que manipula um *script* que consulta um banco de dados e processa o resultado da consulta. Abaixo, vemos um exemplo em que podemos comparar a utilização do comando da *WebCompose*.

Exemplos Usando a *DSL*	**query**=‘SELECT * FROM ViewpointClass WHERE ID =‘..tonumber(VarId) DBQuery(‘*Viewpoints*‘,*query*, ‘-‘ , ‘*#Cons*‘,‘*1*‘,‘*ImprimeURL.lua*‘,‘*MenError.lua*‘)
	query=‘SELECT * FROM ViewpointClass WHERE ID =‘..tonumber(VarId) DBQuery(‘*Viewpoints*‘,*query*, ‘*prog1.lua*‘ , ‘*#Cons*‘,‘*1*‘,‘*ImprimeURL.lua*‘,‘-‘)

O comando *DBQuery* pode ser visto como um *Design Pattern* tipo *Strategy* [Gamma etti all 1995]. A figura 5.6, mostra um esquema do que foi mencionado.

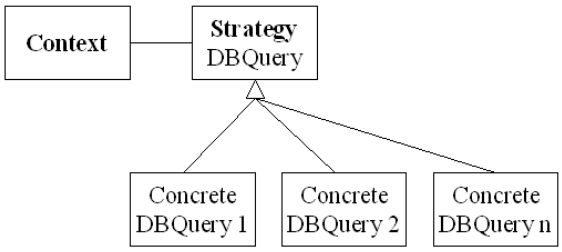

Figura 5.6 - *Design* do comando *DBQuery*

Comando: **DBPipe**

Sintaxe**:**

Versão *black-box*

DBPipe(<DSNName>[1],<#TableName>[2],<Lista de campos>[3],<#TableResult>[4], <Lista de campos(tipo)>[5])

Versão *white-box*

DBPipe(<DSNName>[1],<#TableName>[2],<NameOfADOService script>[1a])

Este comando cria uma série de *pipes* em tabelas de banco de dados com possibilidade de executar filtros, definir que campos dos registros devem ser lidos e gravados em registros de outra tabela. Também pode-se fazer conversão de tipos de dados na geração da nova tabela.

Propriedades de *ADVService/ADOService* satisfeitas:

O comando *DBPipe* satisfaz as seguintes propriedades:

- **Consistência vertical:** verifica-se sempre que o comando da *WebCompose* conseguir responder ao cliente de forma satisfatória.
- **Criação:** verifica-se, pois o *role* desempenhado por este comando da linguagem pode variar em tempo de execução ou ser estático. Para ser estático, basta utilizar as opções *default* do próprio comando. O *DBPipe*, por definição, desempenha sempre um papel dinâmico. Esta afirmação é verdadeira na medida em que os objetos manipulados são tabelas pertencentes a banco de dados, e estas estão sempre em constante modificação, seja por atualização, por exclusão ou por inclusão de registros. Desta forma, o produto final criado será uma conseqüência

direta do estado das tabelas manipuladas, podendo variar sempre que alguma das tabelas envolvidas no processo for modificada. Assim, informações diferenciadas serão enviadas a todas as ocorrências de aplicações que estiverem utilizando este comando.

- **Composição**: verifica-se sempre que o comando necessita utilizar o *hot-spot*[1a]. Neste caso, usa-se uma abordagem *white-box* para a utilização do comando. Quando este *hot-pot*[1a] é utilizado, diferentemente da abordagem do comando *DBQuery,* a composição torna-se evidente, pois o *script* utilizado é a essência do comando que, neste caso, define um "*hook"* que delega toda a execução para um *script* externo que é anexado ao seu próprio código fonte.
- **Tipos de mídia:** verifica-se, pois o *DBPipe* atua como um canal de dados e realiza um filtro dos dados que entram e, se necessário, os converte para outro tipo. Quando o *DBPipe* é utilizado para servir como um "*pipe"*, ele oferece as seguintes funcionalidades: um canal para que os dados possam trafegar, a possibilidade de receber como entrada uma tabela, de filtrar os dados da tabela recebida como entrada, a conversão de tipos e a criação de outra tabela com um formato definido pelo uso de filtros. Desta forma, diferentes tipos de mídia estão sendo manipulados e transformados.

A estrutura do *DBPipe* pode ser vista na figura 5.7, a seguir:

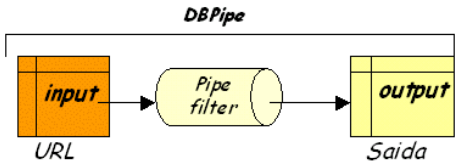

Figura 5.7– *DBPipe*

A tabela 5.6 descreve os *hot-spots* do comando *DBPipe*.

Hot-spot	**Descrição**
hot-spot[1a]: <NameOfADOService script>	Permite que o usuário defina todo o *script* para a realização da tarefa.
hot-spot[1] <DSNName>	Define o nome do banco de dados que está relacionado ao *drive ODBC* do servidor.
hot-spot[2] <#TableName>	Permite que o usuário defina o nome da tabela inicial a ser lida.
hot-spot[3]	Permite que o usuário defina os campos e tipos a

<Lista de campos(tipo)>	serem lidos do registro a ser acionado.
hot-spot[4] <#TableResult>	Permite que o usuário defina o nome da tabela resultante.
hot-spot[5] <Lista de campos(tipo)>	Permite que o usuário forneça os novos campos e tipos a serem gerados na nova tabela.

Tabela 5.6 – *hot-spots* do comando *DBPipe*

A figura 5.8 mostra um exemplo da utilização do *DBPipe*.

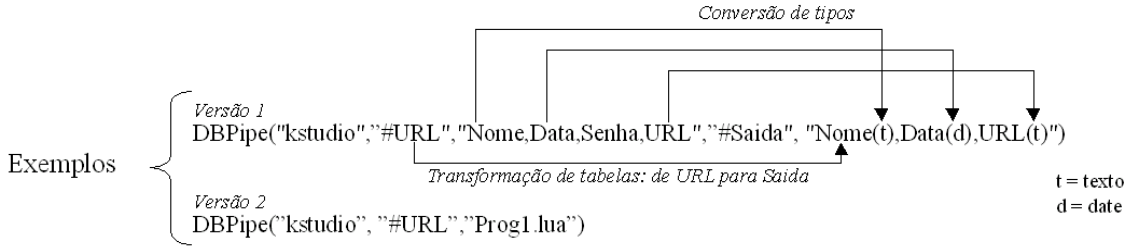

Figura 5.8 - Exemplo do uso da sintaxe do *DBPipe*

Na figura 5.8, podemos ver duas versões da utilização do comando *DBPipe*. Será descrito somente a primeira versão do comando que está presente na figura 5.8.

Na primeira versão, o comando tem um uso tipo *black-box*, ou seja, ele é utilizado de forma parametrizada. Neste exemplo, é feita uma conversão de uma tabela chamada *URL* presente no banco de dados chamado *kstudio* para outra chamada Saída. Os campos lidos das *tuplas* da tabela *URL* são respectivamente: Nome, que é um campo tipo texto; Data, que é do tipo texto; Senha, que é do tipo texto; e *URL*, que também é do tipo texto. Faz-se uma conversão, ou seja, a tabela passa por um *pipe*, sofrendo transformações para a geração da tabela Saída com os seguintes campos e tipos: Nome, que permanece com o mesmo tipo, que é texto; Data, que agora possui um novo tipo, tipo data; e *URL*, que permanece com o mesmo tipo inicial que é texto. A figura 5.9 mostra um exemplo.

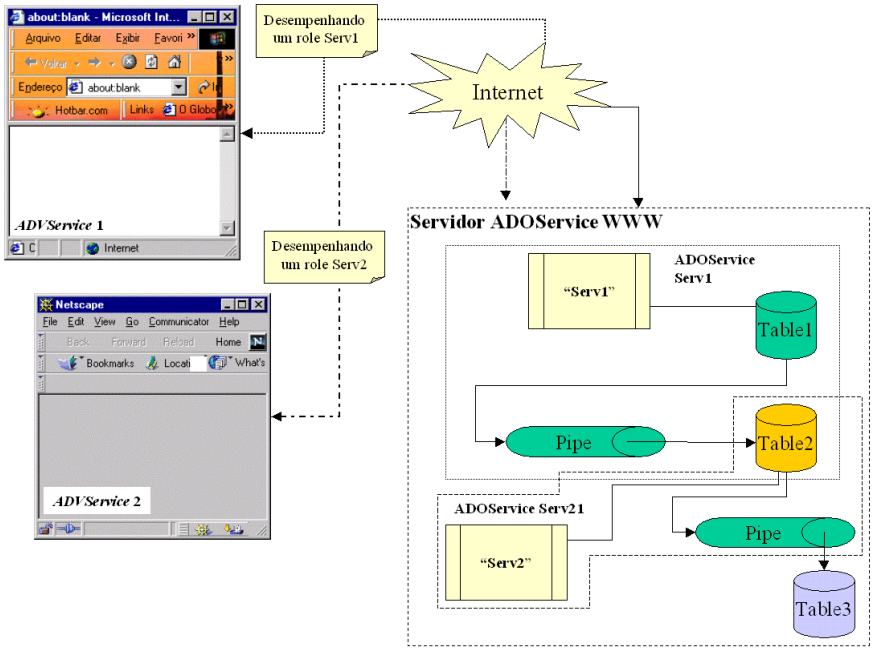

Figura 5.9 - Exemplo de uso do *DBPipe*

Neste exemplo, uma aplicação desempenhando um *role* de nome *Serv1* utiliza uma determinada tabela (*table1*) e gera uma segunda tabela chamada *table2* pelo comando *DBPipe*. A segunda tabela é objeto de consulta por outra aplicação, a qual desempenha um *role* (*Serv2*) que faz uso da tabela de nome *table2*.

A seguir, será descrito o comando *DBDeleteTable*.

Comando: **DBDeleteTable**

Sintaxe**:**

DBDeleteTable(<DSNName>[1],<#TableName>[2])

Permite a exclusão de uma tabela criada pelo comando *DBPipe*.

Propriedade de *ADVService/ADOService* satisfeita:

Não se aplica no caso deste comando.

Os *hot-spots* estão descritos na tabela 5.7, a seguir.

Hot-spot	**Descrição**
hot-spot[1] <DSNName>	Define o nome do banco de dados que está relacionado ao *drive* ODBC do servidor.
hot-spot[2] <#TableName>	Define o nome da tabela a ser excluída.

Tabela 5.7 – *hot-spots* do comando *DBDeleteTable*

Comando: **Proxy**

Sintaxe**:**

Proxy(< Lista de variáveis >[1],<NomedoMétodo>[2],<Servidor>[3])

Este comando permite que se receba uma coleção de dados oriundos de algum formulário em *HTML*, transformando-os para que possam servir de dados de entrada para outros serviços disponíveis na Web, e adequando os dados recebidos para as respectivas interfaces dos serviços selecionados.

Propriedades de *ADVService/ADOService* satisfeitas:

- **Consistência vertical:** verifica-se sempre que o comando da *WebCompose* conseguir responder ao cliente de forma satisfatória. Neste comando, a consistência vertical fica clara na execução do mesmo. A base de sua funcionalidade está em operações de envio de formulários pela Web, para que sejam processados por serviços já existentes. Neste caso, a consistência vertical fica explicitada na resposta que o usuário recebe da aplicação que possui o serviço requisitado.
- **Tipos de mídia:** verifica-se pelo fato do comando *proxy* estar servindo de conexão entre mídias com interfaces distintas. Quando um formulário é utilizado para se ter acesso a um serviço já existente, torna-se necessário adequar as interfaces dos dois lados da operação. Pode ser necessário que os dados recebidos sejam transformados para que sirvam de entrada para o *script* que processa um determinado serviço. Desta forma, a utilização de mídias distintas verifica-se e torna-se visível ao usuário no recebimento da resposta que o servidor externo emite ao cliente que enviou o pedido.

A tabela 5.8 descreve os *hot-spots* do comando *Proxy*.

Hot-spot	**Descrição**
hot-spot[1] < Lista de variáveis >	Permite que o usuário defina uma lista de dados a ser fornecida como parâmetro ao *script* de destino que a processará
hot-spot[2] <NomedoMetodo>	Permite que o usuário defina o método que se utilizará para a passagem dos parâmetros que são: *Get* ou *Post*.
hot-spot[3] <Servidor>	Permite que o usuário defina o servidor que fornece o serviço.

Tabela 5.8 – *hot-spots* do comando *Proxy*

As figuras 5.10 e 5.11, a seguir, exemplificam a utilização do comando *Proxy* em uma aplicação real de comércio eletrônico chamada VBookM@rket.

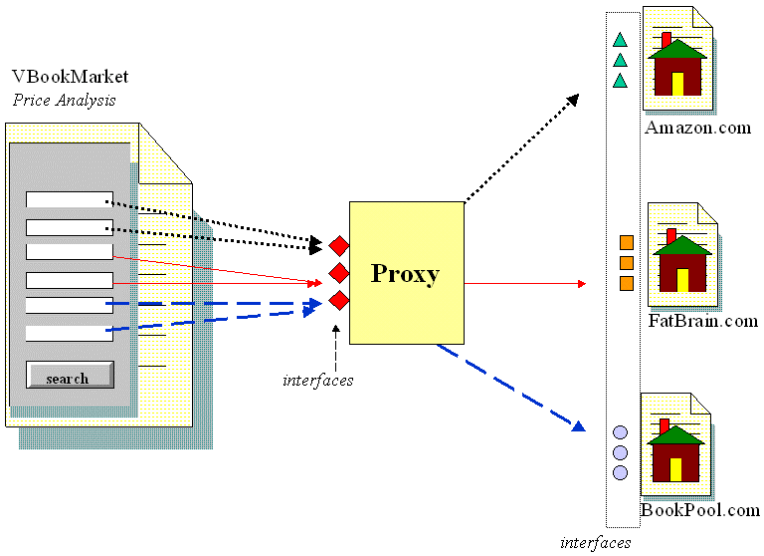

Figura 5.10 – Proxy

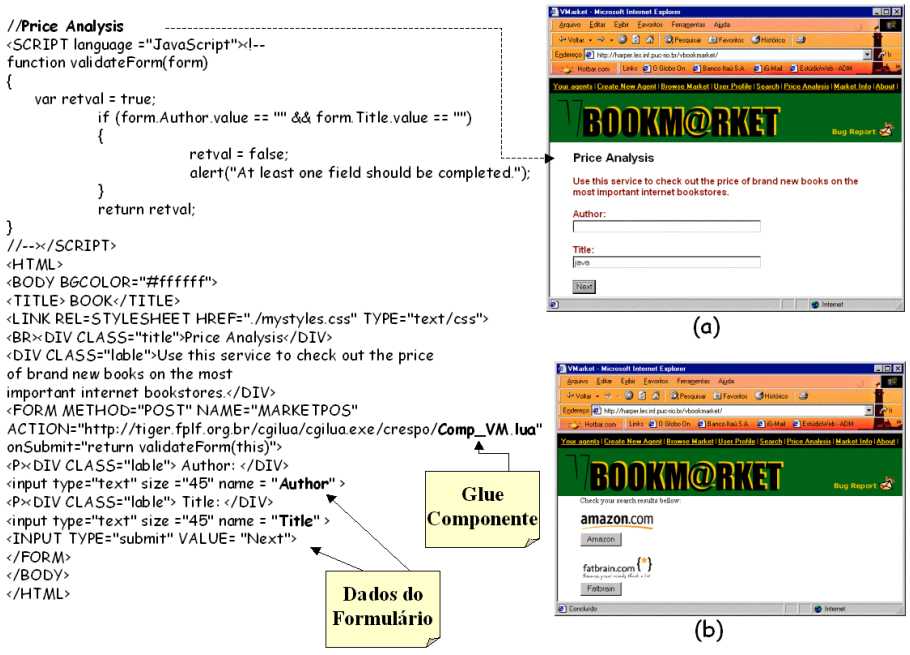

Figura 5.11 - Código fonte da aplicação *HTML* que usa o comando Proxy

Na figura 5.11, podemos ver o código em *HTML* de uma aplicação que utiliza o comando *proxy* para, por meio de seu formulário, disparar uma série de pesquisas de preços de livros em várias livrarias virtuais tais como *Amazon*, *FatBrain* e *BookPool*. O código que aparece do lado esquerdo da figura 5.11 é a forma textual da respectiva interface do usuário para a figura 5.11 (a). Quando o usuário preenche o formulário e

pressiona *next*, o formulário é enviado para um *script* chamado *Comp_VM.lua*. Este *script* usa o comando *Proxy* para poder enviar ao usuário em questão a tela representada pela figura 5.11 (b). Nesta fase, a aplicação, ou o *ADOService*, já está preparada para desempenhar um *role* específico para cada livraria. O comando *proxy* aqui recebe como única entrada um formulário para ser utilizado como entrada para os diferentes tipos de interfaces pertencentes as livrarias virtuais mencionadas.

O comando *proxy* possui um *design* baseado no *Desing Pattern Proxy* [Buschmann ett alli 1996][Gamma et alli 1995]. A figura 5.12 ilustra o que foi dito.

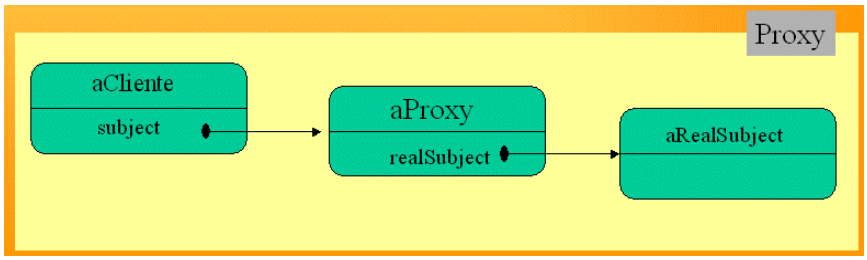

Figura 5.12 - Design Pattern Proxy

A seguir, será descrita a estrutura que permite a persistência de dados, *scripts* e variáveis de *CGI*. Esta estrutura é uma implementação do *Design Pattern Blackboard* [Buschmann et alli 1996]. Inicialmente, será descrita a estrutura criada e, depois, todos os comandos que fazem o acesso ao *Blackboard*.

Comando: **Slice**

Sintaxe**:**

Slice(< URL >[1],<String inicial>[2],<String final>[3])

Este comando permite extrair uma fatia de uma página *HTML*. O *slice* estabelece uma conexão tipo *socket* com o servidor definido na passagem do parâmetro *URL*, interceptando o retorno da página. Desta forma, o *slice* transfere para o cliente que requisitou o serviço somente o trecho delimitado pelo parâmetros *String* inicial e *String* final.

Propriedades de *ADOService/ADOService* satisfeitas:

- **Consistência vertical:** verifica-se sempre que o comando da linguagem conseguir responder ao cliente de forma satisfatória. Este comando exibe a propriedade vertical no momento em que o trecho de página *HTML* é selecionado de um determinado servidor e apresentado ao cliente.

- **Consistência horizontal:** verifica-se sempre que os mesmos parâmetros forem utilizados em mais de uma ocorrência do *browser* com esse *ADVService* especificado. Quando se utiliza o comando *slice*, todas as ocorrências de aplicações que estejam utilizando o mesmo comando devem apresentar a mesma resposta e, assim, a propriedade horizontal verifica-se. Porém, se o *slice* for realizado em páginas dinâmicas, ou seja, que são o produto de um *CGI*, então a propriedade horizontal pode não ser verificada. Como um exemplo desta possibilidade, basta que seja feito um *slice* em páginas que apresentam plantões de notícias. Estas páginas são constantemente atualizadas e, desta forma, geram resultados sempre diferentes em cada momento no tempo.
- **Tipos de mídia:** verifica-se pelo fato do comando *slice* estar servindo de conexão entre mídias com interface distintas. O comando *slice* possui a peculiaridade de poder atuar como um comando *Proxy*, se dentro do parâmetro *URL* forem também especificadas as passagens de parâmetros. Desta maneira, o comando *slice* adaptaria duas mídias distintas para processamento. Um exemplo de como seria o parâmetro *URL* para que o *slice* se transforme em um *Proxy* é:

  URL:=http://www1.fatbrain.com/shop/quicksearch.cl?SearchFunction=key&vm=c&qtext=java&Submit.x=26&Submit.y=11

  Neste caso, a *URL* possui os seguintes parâmetros:

  Parâmetros: SearchFunction=key&vm=c&qtext=java&Submit.x=26&Submit.y=1

  Desta forma, pode-se simular o comando *Proxy* com características de corte.

A seguir, estão descritos os parâmetros do comando *slice*.

Hot-spot	**Descrição**
hot-spot[1] < URL >	Permite que o usuário defina o endereço *HTTP* onde está a página. Pode-se definir, também, o endereço seguido de parâmetros *CGI*.
hot-spot[2] <String inicial>	Permite que o usuário defina a partir de onde o *slice* iniciará a mostrar o resultado da consulta ao cliente que requisitou o serviço. Caso a *string* não exista, a exibição iniciará no início da página *HTML*.
hot-spot[3] < String final >	Permite que o usuário defina até onde o *slice* mostrará. Caso a *string* não exista, a exibição será até o final da página *HTML*.

Tabela 5.9 – *hot-spots* do comando *Slice*

A seguir, mostraremos um exemplo da utilização do comando *slice* A precisão do resultado depende de como foram estabelecidas as opções de *string* inicial e final. É importante ressaltar que o "*parsing*" é feito dentro do código *HTML* recebido.

As figuras 5.13 e 5.14, a seguir, ilustrarão o nosso exemplo.

O comando utilizado *slice* será assim definido para o exemplo:

O formato do comando é:

slice "http://www.les.inf.puc-rio.br/~crespo/publicações.htm" "REVISTAS-->" "<!-- CONFERENCIAS"

O comando *slice* é sensível a caracteres maiúsculos e minúsculos.

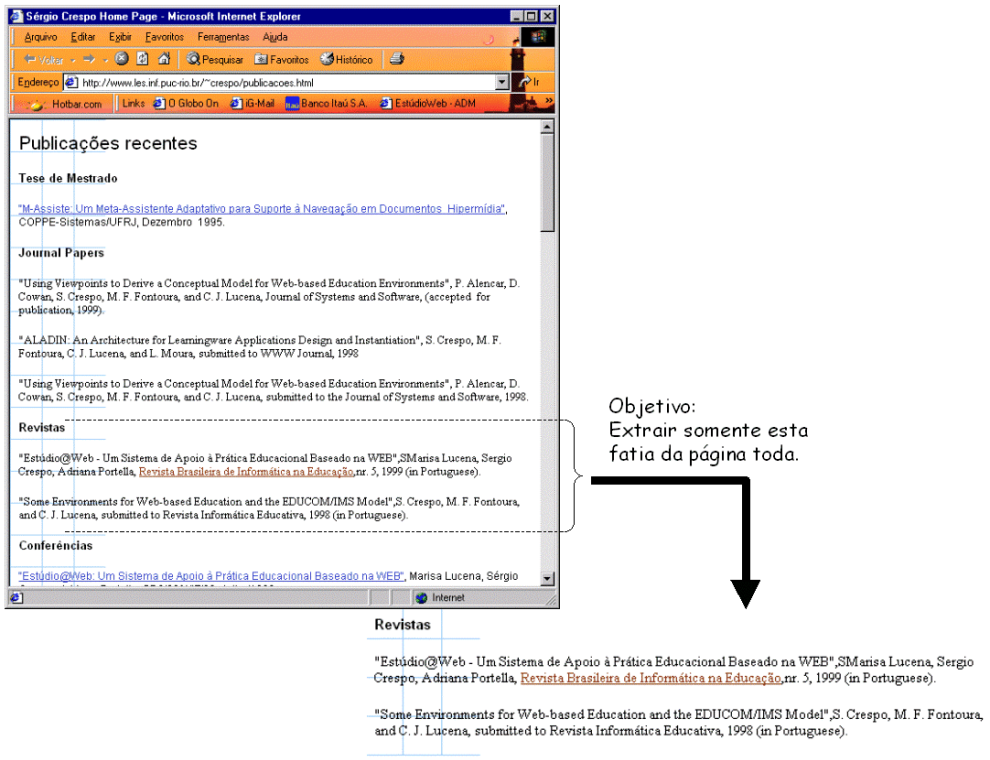

Figura 5.13 - Exemplo do comando *slice*

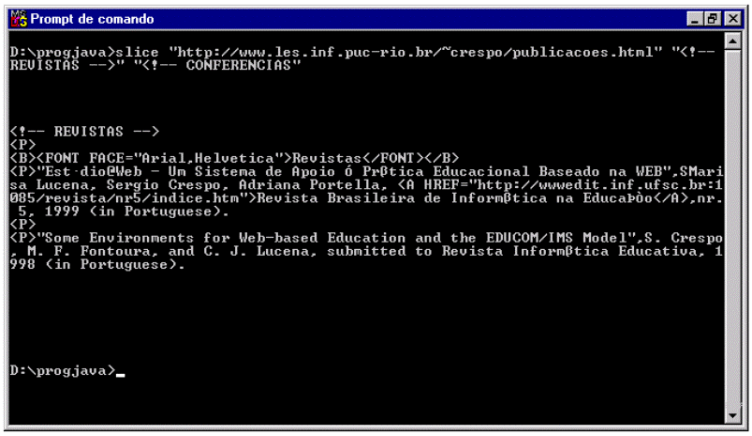

Figura 5.14 - Resultado do *slice*

O *slice* busca todo o conteúdo da página, armazenado-a em um *buffer* temporário, no qual as operações necessárias para a extração da *string* são executadas.

Comando: **SendToBlackBoard**

Sintaxe**:**

SendToBlackBoard (< VarName >[1],<ScriptName>[2],<Timeout>[3])

Este comando permite gravar ou atualizar uma determinada informação na estrutura de *Blackboard*.

Propriedade de *ADOService/ADOService* satisfeita:

- **Comunicação**: Este comando permite que se verifique toda comunicação assíncrona entre *scripts* para o seu correto processamento. Quando uma aplicação necessita obter dados que não são gerados por ela, torna-se necessário utilizar uma estrutura de *Blackboard* que permite que dados sejam disponibilizados para posterior utilização, ou seja, o caráter assíncrono do processamento fica evidenciado. Logo, uma forma de garantir que os dados possam chegar a *scripts* que necessitem desta informação é pela utilização do comando *SendToBlackBoard*.

Os parâmetros para a utilização do comando são:

Hot-Spot	**Descrição**
Hot-spot[1] <VarName>	Nome da variável que contém a informação. Esta variável pode ter qualquer tipo de formato e tamanho. Pode-se armazenar dados ou até *scripts* para serem executados por outros programas.
hot-spot[2]<ScriptName>	O nome do *script* que está gravando dados no *Blackboard*.
hot-spot[3] <Timeout>	Um valor em segundos. Este valor é utilizado pelo *Blackboard* no caso do dado requerido estar sendo utilizado. Caso o tempo se exceda, uma mensagem é enviada ao cliente.

Tabela 5.10 – *hot-spots* do comando SendToBlackBoard

Comando: **LoadFromBlackBoard**

Sintaxe**:**

LoadFromBlackBoard (< VarName >[1], < Timeout >[2])

Este comando permite buscar uma determinada informação na estrutura de *Blackboard*.

Propriedade de *ADOService/ADOService* satisfeita:

Comunicação: Este comando, tal qual o comando *SendToBlackBoard*, permite que se verifique a comunicação assíncrona entre *scripts* para o seu correto processamento. Quando uma aplicação necessita obter dados que não são gerados por ela, torna-se necessário utilizar uma estrutura de *Blackboard* como forma de área comum entre *scripts*.

Quando se torna necessário obter uma informação que pode ser uma variável ou mesmo um *script* completo, o comando *LoadFromBlackBoard* torna-se necessário.

Os parâmetros para a utilização do comando são:

Hot-Spot	**Descrição**
Hot-spot[1] <VarName>	Nome da variável que contém a informação. Esta variável pode ter qualquer tipo de formato e tamanho. Pode-se recuperar dados ou até *scripts* para serem executados por outros programas.
hot-spot[2] <Timeout>	Um valor em segundos. Este valor é utilizado pelo *Blackboard* no caso do dado requerido estar sendo utilizado. Caso o tempo se exceda, uma mensagem é enviada ao cliente.

Tabela 5.11 – *hot-spots* do comando LoadFromBlackBoard

As propriedades de ADVService/ADOService apresentadas no Capítulo 4 são:
Consistência vertical, Consistência horizontal, Composição, Comunicação, Criação e Tipos de Mídia. Entre os comandos apresentados, todas essas propriedades são satisfeitas. Usamos esta constatação como um indicador de que pelo menos as propriedades de *design* satisfeitas por *ADVService/ADOService* podem ser implementadas pela *WebCompose*. Isto significa que um *design* de *framework* para a plataforma Web pode tirar partido da grande variedade de situações de composição

contempladas por *ADVService/ADOService* e que este *design* pode ser diretamente implementado pela linguagem e pelo ambiente proposto.

A seguir, podemos ver um resumo dos comandos e as propriedades satisfeitas.

Comandos	**Propriedades satisfeitas**
Constraint	Comunicação
Evaluate	Comunicação
DBQuery	Consistência vertical Criação Composição
DBPipe	Consistência vertical Criação Composição Tipos de mídia
DBDeleteTable	Nenhuma propriedade se aplica
Proxy	Consistência vertical Tipos de mídia
Slice	Consistência vertical Consistência horizontal Tipos de mídia
SendToBlackBoard	Comunicação
LoadFromBlackBoard	Comunicação
GarbageBlackBoard	Nenhuma propriedade se aplica

Tabela 5.12 – Comandos da *DSL WebCompose* e suas propriedades

Como pode ser visto na tabela acima, com a utilização simultânea de alguns comandos da *DSL* torna-se possível aplicar todo o conjunto de propriedades dos *ADVServices/ADOServices*. A seguir, descreveremos como é feito assincronamente o compartilhamento de variáveis por *scripts* diferentes.

5.4 Persistência no Servidor *ADOService WWW*

O Servidor *ADOService* WWW utiliza uma estrutura baseada no *Design Patterns Blackboard* [Buschmann et alli 1996] para manter a persistência dos dados e *scripts* que necessitam ser compartilhados por outros *ADOServices*.

A idéia da arquitetura de *Blackboard* é ter uma coleção de serviços independentes que trabalhem de forma cooperativa sobre uma estrutura de dados comum. Cada serviço é configurado para resolver parte de um problema e essa coleção de *scripts* atuando em conjunto resolve uma determinada tarefa. O *Blackboard* é uma central de armazenamento de dados. Elementos que pertencem ao espaço de solução e controle de dados estão armazenados nesta estrutura [Buschmann et alli 1996].

Seguindo este enfoque, toda a persistência de dados e *scripts* necessários para que uma tarefa possa ser concluída residirá no *Blackboard*. A seguir, será descrita a estrutura de como o *Blackboard* esta implementado na *WebCompose*.

5.4.1 Arquitetura da estrutura de persistência do servidor *ADOService* WWW

Um das características marcantes em sistemas baseados na Web é o fato de serem predominantemente assíncronos. Ou seja, sistemas que realizam algum tipo de processamento de dados, via requisição de formulários, são, em sua grande maioria, assíncronos, na medida em que não se pode saber quem está e quantas aplicações estão usando um determinado *script* para prover um serviço. Com o enfoque de *WebFrameworks*, torna-se necessário de obter persistência de informação para que serviços compostos, e que dependam da correta execução de outros serviços, para realizar suas tarefas satisfatoriamente possam ser executados. Os servidores WWW não oferecem nenhum tipo de persistência para variáveis que são enviadas via formulários. As linguagens de *scripts*, por sua vez, também têm a mesma carência.

Para enfrentar este problema, uma estrutura de *Blackboard* foi implementada para permitir que qualquer tipo de informação possa ser enviada ou recuperada sem que nenhum sincronismo de serviços seja exigido. A idéia é básica, quando um serviço necessitar de informações extra para que possa desempenhar corretamente o seu papel, ele utilize o *Blackboard* para obter essas informações.

Toda a forma de comunicação com o *Blackboard* é feita por dois comando da *WebCompose*: *SendToBlackBoard* e *LoadFromBlackBoard.*

A figura 5.15, a seguir, mostra uma visão geral do *Blackboard*.

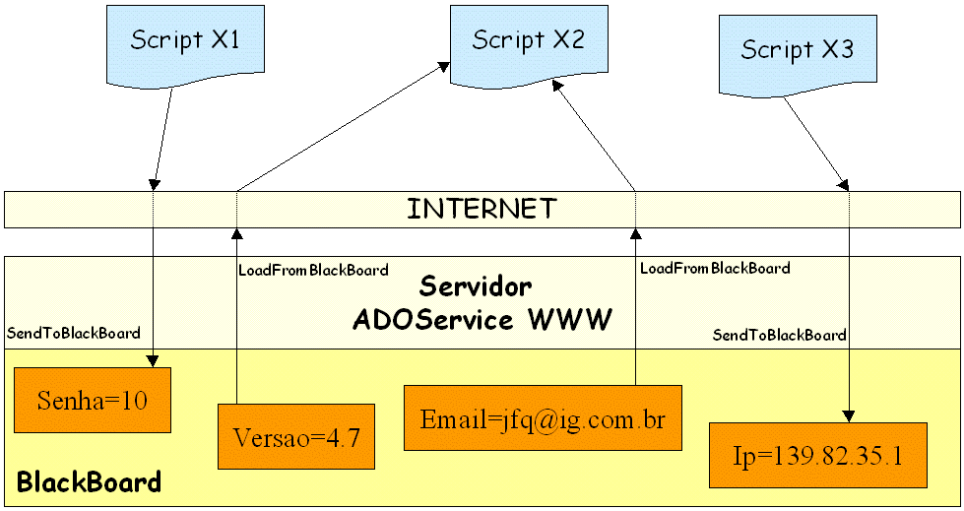

Figura 5.15 – *BlackBoard*

A figura 5.15 mostra uma série de *scripts* que necessitam buscar dados assincronamente para poderem executar seu processamento. Outras aplicações, por sua vez, deixam dados para serem processados por outros *scripts*. Cada informação armazenada no *Blackboard* possui a estrutura descrita na tabela a seguir:

Atributos	**Descrição**
Nome	Nome da informação a ser armazenada. Esta informação poderá ser um dado ou um *script*.
Data de criação/atualização	Data em que foi armazenada ou atualizada no *Blackboard*.
Valor	Conteúdo da informação armazenada.
Quem criou/atualizou	Lista dos *scripts* que fazem a atualização da informação.
Status	Se estiver com valor *true*, significa que o conteúdo da informação é confiável, ou seja, que foi gravado sem nenhuma anormalidade. Se estiver com valor *false*, significa que ocorreu algum problema no transcorrer da operação de criação e ou atualização. Neste caso, uma mensagem de erro é gerada pelo comando responsável pela leitura da informação desejada no *Blackboard*.

Tabela 5.13 – Atributos de informação no *BlackBoard*

O processo de concorrência ao *Blackboard* foi resolvido da seguinte forma.

Quando um *script* deseja guardar uma informação no *Blackboard*, os seguintes procedimentos que podem ser vistos na figura 5.16. são realizados pelo comando *SendToBlackBoard*.

```
//recebe os dados
Recebe(Name,Dado,Data,Owner,Timeout,CalledBy)
flag:=0
tempo:= time()
//testa para saber se o dado esta loked e ou se excedeu o
// tempo limite de espera para o dado ficar disponível
Faça enquanto (tempo <Timeout) ou locked(Name) do
  tempo = calctime(tempo)
  se (locked(Name) ) então
    flag:=1
  senão
    flag :=0
    sair
  fim
fim
Se (flag==1) então
  Retornar(TimeOutMessage(1))
end
// dado disponível, troca o status para off e inicia o processo de atualização
set status:=off
AtualizaDado(Name,Dado,Data,Owner,Timeout,CalledBy
// libera o dado unlock
Unlock(Name)
// muda status para on - processo atualizado sem problema
Status:=on
Retornar
```

Figura 5.16 - Algoritmo *SendToBlackBoard*

O processo de atribuição de uma informação no *Blackboard*, como visto na figura 5.15, procura, inicialmente, certificar-se de que a informação não se encontra bloqueada. Se esta não estiver disponível, então ele espera um tempo determinado para que não seja enviada uma mensagem padrão do *browser* de que a conexão foi cancelada por *timeout*.

Se a informação não estiver bloqueada, inicia-se, então, o processo de atualização. Como um procedimento de segurança, uma variável chamada *status* é atualizada com um valor lógico *false* até que o processo de atualização esteja terminado. Após o término da atualização, a variável *status* volta a ter um valor lógico *true*, terminando o processo de atualização de informação no *Blackboard*.

Sempre que uma nova inclusão e ou atualização é realizada, ela possui a seguinte chave de acesso: *Nome* + *Data*. A figura 5.17 mostra um exemplo do que foi dito. Desta forma, pode-se encontrar no *Blackboard* uma redundância de informação com várias datas diferentes. Esta redundância é, posteriormente, tratada por um módulo coletor de lixo que será comentado mais adiante.

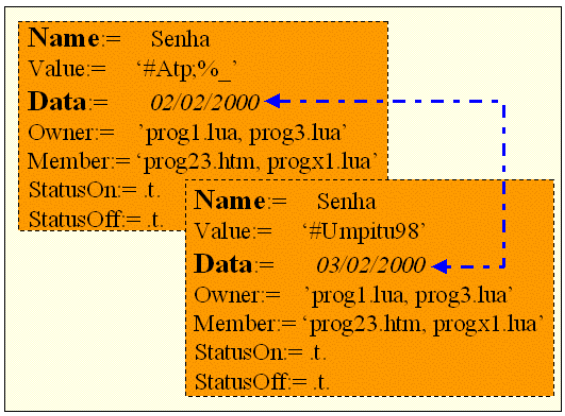

Figura 5.17 - Dados no *Blackboard*

Quando um *script* deseja buscar uma informação que está presente no *Blackboard*, um processo análogo é disparado. Inicialmente, verifica-se se a informação não está bloqueada. Caso esteja, então fica-se em um laço de tempo predeterminado (que é informado pelo usuário). Se o tempo expirou e a informação ainda está bloqueada, manda-se, então, uma mensagem de erro ao cliente.

Caso a informação esteja disponível, ela é retornada ao *script* que solicitou o serviço ao *Blackboard*. **Quando uma informação é solicitada, devolve-se a informação mais atual disponível no *Blackboard***.

A seguir, será descrita a maneira pela qual a redundância de informação é manuseada no *Blackboard*.

5.4.2 Coletor de lixo do *Blackboard*

A eliminação de informação redundante no *Blackboard* é feita segundo o seguinte procedimento: excluem-se todas as informações com o mesmo nome de datas mais antigas. Desta forma, somente uma única ocorrência da informação estará disponível. Assim sendo, obtém-se uma redundância mínima no *Blackboard*, pois como não se pode prever se uma determinada aplicação fará uso de uma dada informação. Esta, uma vez gerada, permanecerá no *Blackboard* com somente uma única ocorrência. A figura 5.18 exemplifica o processo descrito.

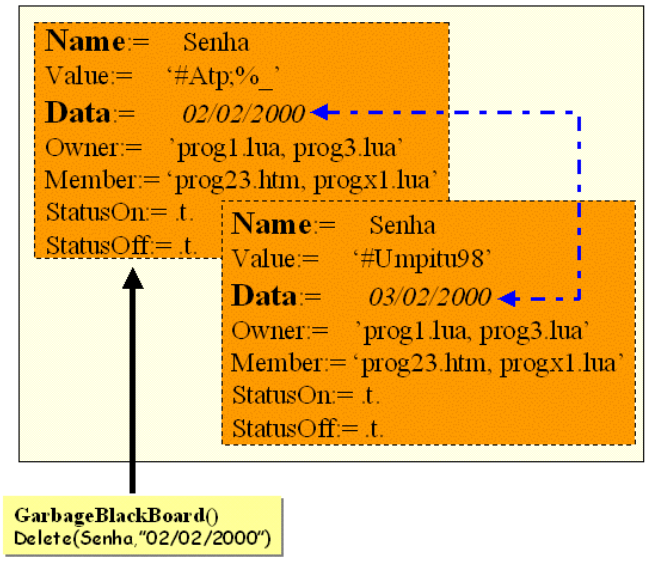

Figura 5.18 - Coletor de lixo do *Blackboard*

Um dos pontos fortes do *Blackboard* é a opção de deixar a persistência no Servidor *ADOService* WWW ao invés de deixá-lo na aplicação cliente. Este fato evita acidentes de perda de informação por motivos de queda de energia elétrica, perda da conexão da *Internet* ou fechamento acidental da aplicação cliente. Em contrapartida, tem-se um processo mais lento em virtude da necessidade de se acessar o servidor quando se precisa manipular dados que sejam necessários para que um determinado serviço possa executar o seu processamento corretamente.

A seguir, apresentaremos algumas abordagens semelhantes e compararemos com a *WebCompose* com estas outras *DSLs*.

5.5 Comparação com outras abordagens

Algumas abordagens sobre o tema de combinação de serviços na Web procuram utilizar algum tipo de *DSL* para tentar compartilhar recursos existentes. Esta seção apresentará três linguagens de domínio para este problema: *WebL* [Kistler, Marais 1998], Mawl [Atkins, Ball, Bruns and Cox 1999] e Piccola [Achermann, Limpe, Schneider and Nierstrasz 2000], [Lumpe, Achermann and Nierstrasz 2000] e [Schneider, Nierstrasz 1999].

5.5.1 *WebL- A Programming Language for the Web.*

WebL foi inspirada no trabalho de Cardelli e Davies [Cardelli, Davies 1999] e tem como propósito realizar experiências com combinação de serviços para tarefas

relacionadas à Web. O protótipo da linguagem foi implementado em linguagem Java. *WebL* oferece dois tipos básicos de serviços que são: *getpage* e *postpage*.

A função *getpage* busca *URLs* do servidor WWW com protocolo *HTTP GET*, de forma análoga a função *postpage* utiliza o protocolo *HTTP POST* para trazer a *URL* solicitada.

A sintaxe dos serviços da *Webl* são:

getpage(string, [.param1=val1,param2=val2,...])

postpage(string,[.param1=val1, param2=val2,....],[.header=val1, header2=val2.])

Baseada nos serviços acima, uma seqüência de operações pode ser executada, como poderá ser visto a seguir.

- Execução seqüencial: S?T

page:=getpage("http://www.altvista.com") ?
getpage("http://www.altavista.yellowpages.com.au")

O exemplo acima tenta, inicialmente conectar o *site* da *www.altavista.com* e, caso uma falha aconteça, ele tenta uma segunda opção que é o *www.altavista.yellowpage.com.au* na Austrália. A execução do comando é de forma seqüencial.

- Execução concorrente: S|T

page:=getpage("http://www.altvista.com") |
getpage("http://www.altavista.yellowpages.com.au")

O exemplo de concorrência permite conectar os *sites* da *www.altavista.com* e *www.altavista.yellowpage.com.au* simultaneamente.

- *Time-out* timeout(t,S)

page:=timeout(10000, getpage("http://www.altvista.com") ?
getpage("http://www.altavista.yellowpages.com.au"))

O exemplo acima permite uma tentativa de conexão delimitada por um intervalo de tempo em segundo. Caso a primeira tentativa falhe, o segundo *site* tentará ser conectado.

- Repetição: repeat(S)

page:=repeat(getpage("http://www.altvista.com") ?
getpage("http://www.altavista.yellowpages.com.au"))

O exemplo realiza um *loop* de tentativas de duas *URL*s distintas.

- Falha: stall()

page:=repeat(getpage (“http://www.altvista.com”) ?timeout(10000,stall())

O exemplo permite realizar um *loop* de tentativas de duas *URL*s distintas até que uma falha aconteça. Também permite trazer o resultado de uma busca na *Amazon.com*. Uma comparação da utilização da *WebL* com a *WebCompose* pode ser vista na figura 5.19.

A figura 5.19 (a) mostra o código em *WebL* para uma busca seguida de um corte no resultado da consulta. Na figura 5.19 (b), não é necessário o usuário digitar uma série de comandos que não sejam *HTML*. Todas as informações que serão utilizadas para a consulta via o comando *Slice* devem estar dentro de um formulário em forma *HTML*, que será enviado ao Servidor *ADOService* WWW, que processará o pedido utilizando o *Slice* para estabelecer a conexão e realizar o corte superior e inferior baseado nas informações que constam nos campos em *hidden* que são preenchidos via *JavaScript*. O código em *JavaScipt* não foi apresentado, uma vez que não faz parte da análise. Os campos que delimitam o corte no *Slice* são: *StringInicial* e *StringFinal*.

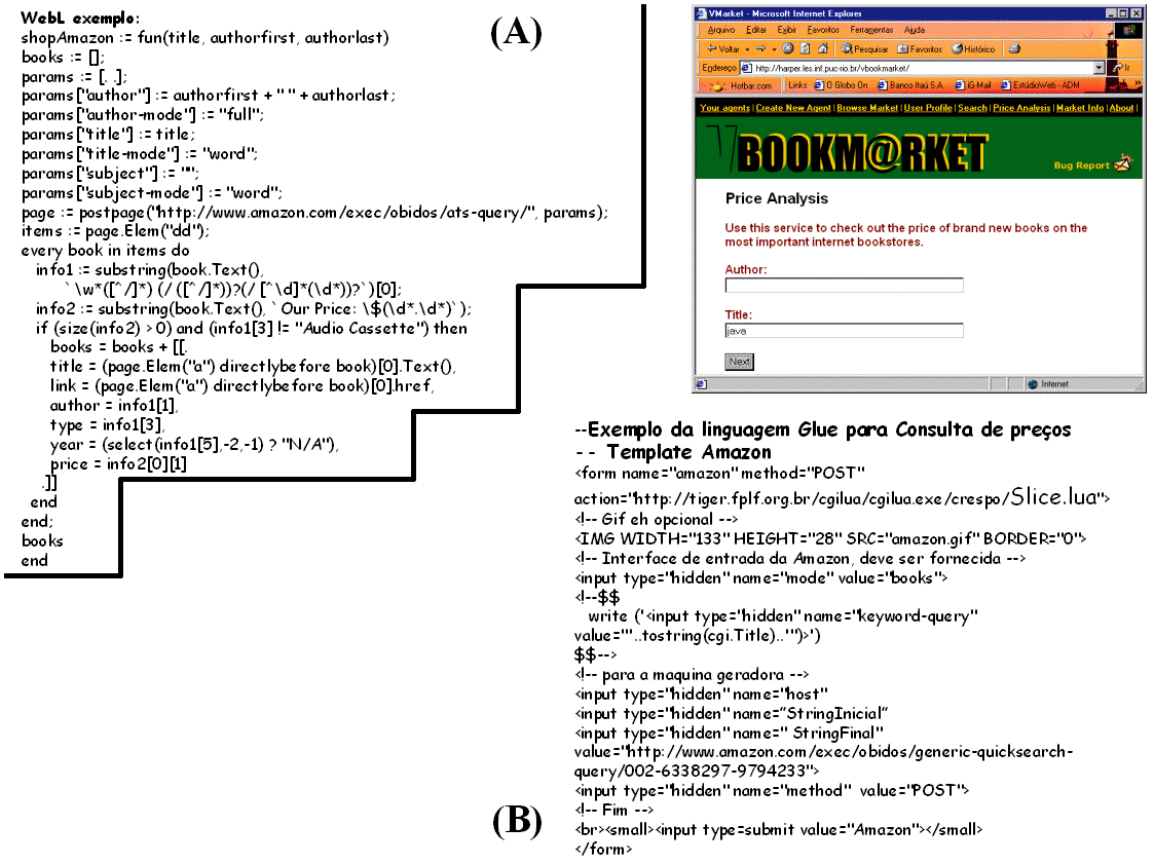

Figura 5.19 - *WebL* x *WebCompose*

A premissa da linguagem *WebCompose* é ser a mais transparente possível para o usuário que, no caso, poderia ser o *WebMaster*. Todos os comandos da linguagem

WebCompose funcionam como *CGI*. Outra alternativa para o exemplo da figura 5.18 seria utilizar, ao invés do comando *Slice,* o comando *Proxy*. O *Proxy* é mais eficiente, porém não permite a realização do corte na página que está retornando. O *Slice* é mais direcionado a tirar uma fatia de uma página *HTML* para uso geral. A implementação hoje existente no ambiente VBookM@rket utiliza o comando *Proxy*. As figuras 5.20 e 5.21 mostram a utilização do *Slice* para extrair trecho de informação do *site* http://www.oglobo.com.br da seção de plantão de esportes. Neste exemplo da figura 5.20, foram realizados 2 cortes:

URL: http://www.oglobo.com.br/futebol/default.htm1

String Inicial: <!--- Final da tag JX ---> String Final: " "

No exemplo menor os cortes foram:

String Inicial: <!--- Final da tag JX ---> String Final: Copa do Brasil

Comando dentro de *WebCompose*: slice(URL,StringInicial,StringFinal)

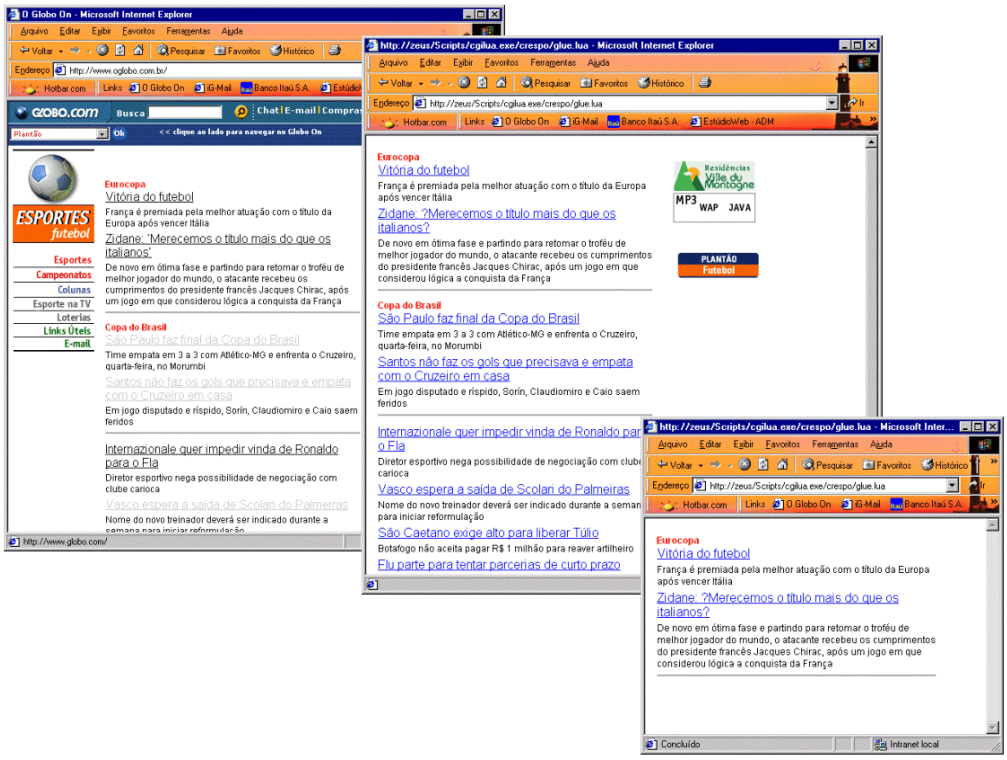

Figura 5.20 - trechos de informação do OGlobo On

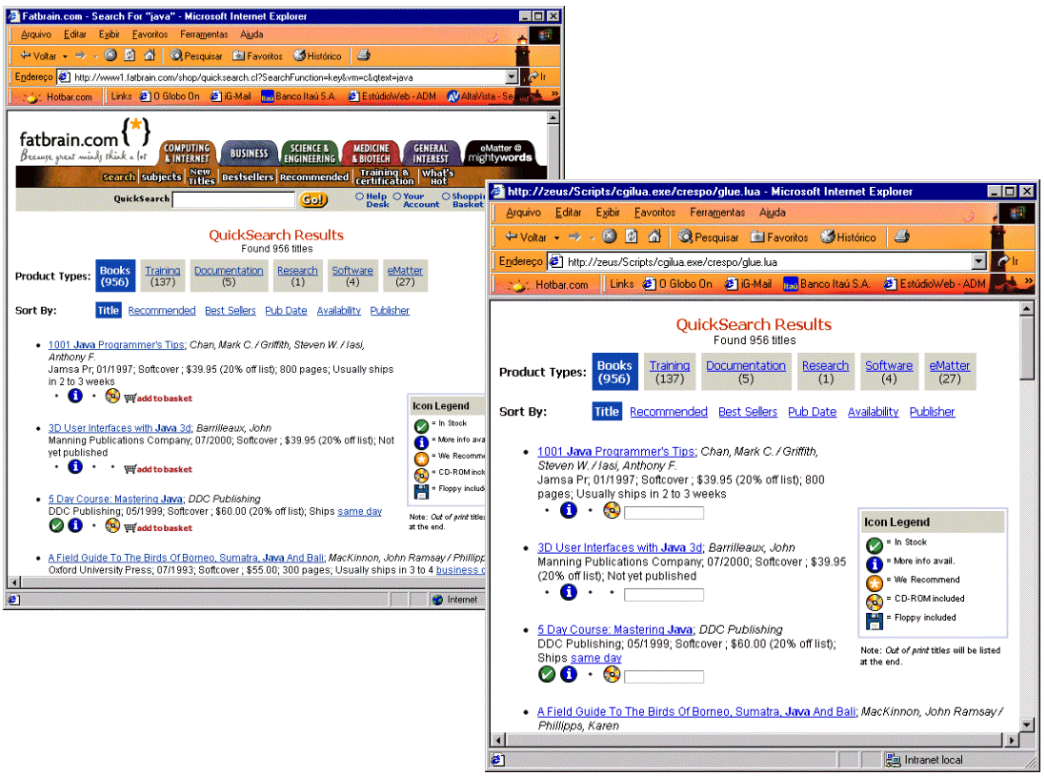

Figura 5.21 - Trecho da consulta a *FatBrain* sobre "Java"

5.5.2 *Mawl: A Domain-specific Language for Form-Based Services.*

Mawl foi desenvolvida em 1995 em virtude das necessidades dos programadores na criação de serviços baseados em formulários *HTML*, usando programas em *CGI* e o protocolo *HTTP*.

Mawl possui uma independência do protocolo *HTTP* e *CGIs*. Um serviço em *Mawl* consiste de uma ou mais seções. Uma seção estabelece o fluxo de controle de um serviço e permite a atualização de variáveis. *Mawl* possui o conceito de *template*, que é uma porção estática ou dinâmica de uma interface do usuário que pode ser parametrizada para exibir informações personalizadas. *Mawl* possui uma linguagem própria chamada *MHTML* que é uma extensão de *HTML* e permite adicionar código *Mawl* dentro de páginas em formato *HTML*.

Mawl está no mesmo nível de ferramentas tipo *CGIs* tais como: *Perl, Phyton, CGILua* e outras. A composição de serviços em *Mawl* acontece no momento em que *MHTML* é utilizado para especificar comportamentos específicos dentro de páginas *HTML*. *Mawl* atua como um *Front-End* entre o cliente e o servidor WWW que contém o serviço a ser executado, fornecendo mais flexibilidade para a confecção do código a ser executado.

5.5.2 PICCOLA – a Small Composition Language.

Piccola é uma linguagem de composição baseada em objetos e que utiliza a teoria de *πλcalculus e polyadic πλcalculus* [Schneider, Lumpe 1997]. Ela utiliza o paradigma de programação funcional. Mais precisamente, o principal elemento da linguagem *Piccola* é uma *expressão de ordem* que representa o conceito unificado de *πλagents e πλforms*. *Piccola* possui a sua sintaxe semelhante às linguagens *Python* [Guelich,Birznieks]0 e *Haskell* [Haskell home page].

Piccola possui um protótipo feito em *Java* e um *Java Gateway* para interconexão com componentes externos. Desta forma, é possível encapsular objetos *Java* dentro de *scripts Piccola*.

Quando um *script Piccola* é executado, providencia-se um contexto inicial para acessos básicos de serviços de I/O, em particular para carregar outros *scripts* feitos em *Piccola*. Além do mais, *scripts* feitos em *Piccola* podem ser anexados em aplicações *stand-alone* feitas em *Java*, ou *applets* ou *servelets*.

O *Gateway Piccola* atua em conjunto com o *servlet* WWW, permitindo interpretar todas as requisições e, desta forma, oferecer mecanismos de composição.

Piccola utiliza também o conceito de *Blackboard*, porém não de forma genérica, para um servidor WWW. Para cada requisição de um *Blackboard*, cria-se uma instância particular para o módulo que o requisitou.

A figura 5.22 mostra um exemplo da utilização de uma estrutura de *Blackboard* em *Piccola* e em *WebCompose*.

(a)

```
Load Blackboard;
function newStorageCell( X) =
  let
     value cell = newBlackboard()
  in
     cell.Write( <val = X.init> );
     <
        function Update( Args ) = cell.Remove; cell.Write( <val = Args> ),
        function Read() = cell.Read()
     >
end
```

(b)

```
---------- Para gravar/atualizar dados no BlackBoard----------
Senha = cgi.senha
status = SendToBlackBoard("Senha", "AtuSenha.lua", 15)


---------- Para ler dados do BlackBoard----------
SenhaAux = LoadFromBlackBoard("Senha", 30)
```

Figura 5.22 - Utilização de um *Blackboard* em *Piccola* e *WebCompose*

Na figura 5.22 (a), podemos ver o exemplo de como se utiliza uma estrutura de *Blackboard* em *Piccola*. Neste caso, para uma determinada instância de X criada pela função *newStorageCell(),* uma particular estrutura de *Blackboard* é disponibilizada para aquele contexto em particular.

O correspondente código em *WebCompose* é apresentado na figura 5.22 (b) onde o *Blackboard* fica disponível para todos os *scripts* que utilizam o Servidor *ADOService* WWW.

5.6 Conclusão

Neste capítulo, apresentamos linguagem de domínio específico *WebCompose*, uma linguagem de *script* que oferece recursos para a composição de serviços já existentes na plataforma Web.

WebCompose foi desenvolvida baseada na teoria de *ADVs/ADOs*, que foi introduzida no Capítulo 4, e *framelets* [Pree 1999a] [Pree 1999b] para a formação da maioria de seus comandos. Junto com a apresentação da linguagem, mostramos que o conjunto de comandos de *WebCompose* permite, pelo menos, a satisfação do conjunto de propriedades de *ADVService/ADOService* que seguem as propriedades de *ADVs/ADOs*.

A linguagem utiliza a ferramenta *CGILua* e *Lua* como linguagem hospedeira, de forma a poder executar tanto os seus comandos quanto os residentes em *CGILua* e Lua, dando mais flexibilidade para a criação dos *scripts* para a composição e ou utilização de serviços. Cada comando da linguagem pode ser especializado para uma particular utilização e, neste caso, duas formas de utilização são oferecidas: *black-box* e *white-box*.

Pela utilização da *WebCompose*, um servidor WWW padrão usando *CGILua* como ferramenta de *script* adquire um novo conjunto de funcionalidades para todos os *sites* residentes neste servidor e possibilidades de vários tipos de "corte e colagem" de outros *sites* na *Internet*.

A esse novo servidor utilizando *WebCompose*, chamamos de Servidor *ADOService* WWW. Com este enfoque, pode-se criar, agora, *WebFrameworks* a partir de um conjunto básico de serviços.

Diversos *Design Patterns* foram utilizados e disponibilizados em formato de comandos tais como *Pipes and Filter* pelo comando *DBPipe*, *Proxy* pelo comando

Proxy (que internamente utiliza um *Strategy*), *Blackboard* para a persistência de dados em servidores *ADOService* WWW e suas diversas aplicações.

Comparando-se *WebCompose* com outras abordagens, podemos identificar a sua grande simplicidade e facilidade de uso por especialistas na gerência de *sites* na Web e que não são necessariamente especialistas em programação *CGI*.

Capítulo 6

Estudo de caso

Resumo

Este capítulo apresenta um estudo de caso no qual será utilizada a *DSL WebCompose* para a criação de um portal que é a base de um *WebFramework*. Este estudo de caso cobre todos os problemas de composição apresentados no Capítulo 3. O *design* do portal será realizado utilizando-se *ADVServiceChart*.

6.1 Introdução

Neste estudo de caso, utilizamos um conjunto básico de serviços já disponíveis na Web para a criação de um portal utilizando-se a linguagem *WebCompose*, descrita no Capítulo 5, que permitirá a composição e utilização destes serviços. A partir deste conjunto básico de serviços, define-se um *WebFramework* que permite a geração de várias aplicações.

Os serviços utilizados pertencem aos seguinte portais:

Hosts	**Serviços utilizados**
IG http://www.ig.cm.br	Turbo *chat*
Bloomberg http://www.bloomberg.com	Seção: /br/ resumo de mercado
O Estadão http://www.estadao.com.br	Seção: economia/finanças
O Estadão	Seção: economia/financas/cotacoes/
iDeal http://139.82.24.86:8080/iDeal/	Serviço de *email*
VBroker http://139.82.24.89:8081/servlet/clipper	Serviço de *clipping*
EstúdioWeb http://www.estudioweb.com.br	Informações na área de matemática

TecComm Server http://www.teccomm.les.inf.puc-rio.br	Sistema de listas *Lyris*
PortalWare http://zeus.teccomm.les.inf.puc-rio.br	*Applet Time*
PortalWare	*Applet Banner*
Altavista http://www.altavista.com	Máquina de busca
Yahoo http://www.yahoo.com	Máquina de busca
Lycos http://www.lycos.com	Máquina de busca

Tabela 6.1 - *Host* e Serviços

Estes serviços básicos são o *kernel* do *WebFramework*. A figura, a seguir, exemplifica o que foi dito.

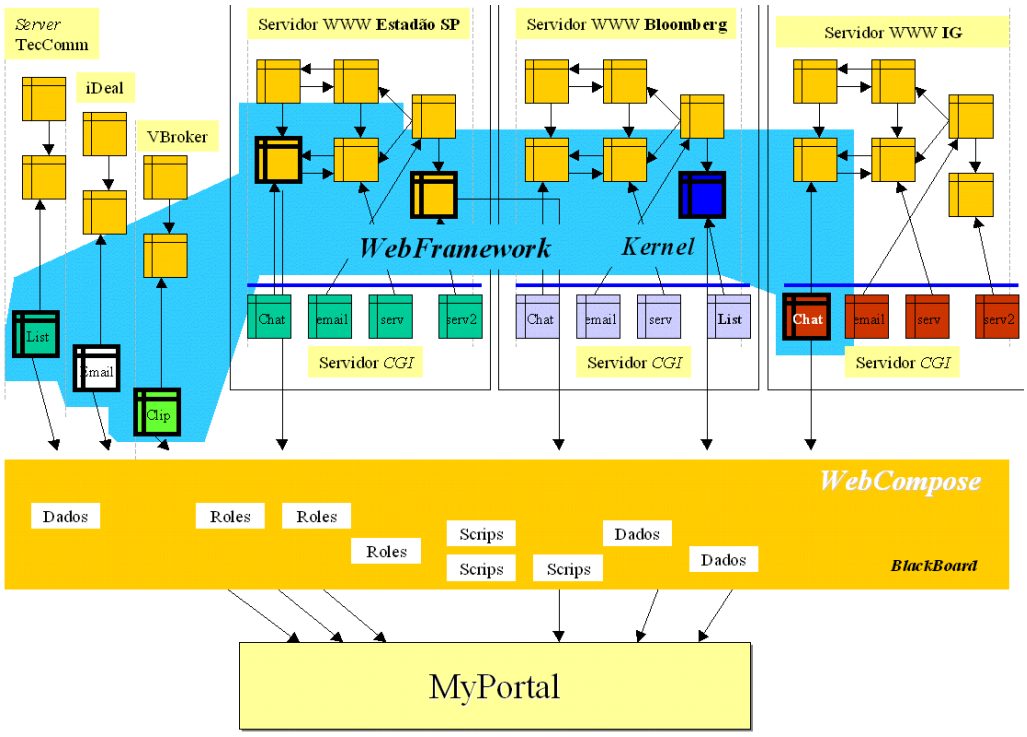

Figura 6.1 - *Kernel* do *WebFramework* para *MyPortal*

Como ilustra a figura 6.1, o *kernel* do *WebFramework* é composto por serviços e até mesmo por páginas que pertencem a vários portais. A aplicação chamada *MyPortal* é decorrente da composição destes serviços mediante o uso da linguagem *WebCompose*.

6.2 *Design* de *MyPortal* usando *ADVServiceCharts*

Apresentar-se-á o *design* da seguinte forma: primeiro utilizando-se a notação de *ADVServiceChart,* e depois uma imagem da tela do respectivo protótipo.

A figura 6.2 apresenta o *design* inicial da aplicação *MyPortal*.

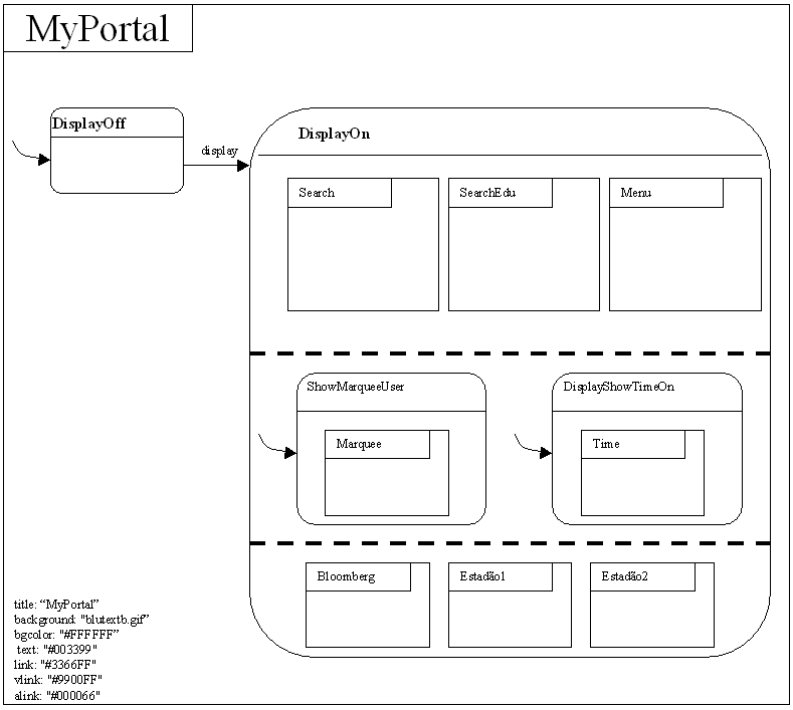

Figura 6.2 - *Design* principal de *MyPortal*

A figura 6.2 apresenta o *design* principal da aplicação *MyPortal*. O objetivo dos vários *designs* apresentados a seguir, é fornecer ao desenvolvedor uma visão geral de como a aplicação deverá ser executada. Também será apresentada a representação, a nível de programação, dos vários *ADVServiceChart* construídos.

No *design* da figura 6.2, podemos ver que o *ADVServiceChart MyPortal* possui dois estados, onde o *DisplayOff* é o estado inicial a partir do qual produz-se um estado de ativação chamado *DisplayOn*. Dentro do estado *DisplayOn*, as linhas tracejadas representam concorrência, ou seja, os processos de "*Search*", "*SearchEdu*" e *Menu* ocorrem de forma seqüencial dentro deste segmento e não seqüencial em relação aos demais que são: "*ShowMarqueeUser*" e "*DisplayShowTimeOn*" que ocorrem também de forma seqüencial dentro de seu segmento. Os *ADVServices* no terceiro segmento são: "*Bloomberg*", "*Estadão1*" e "*Estadão2*", que ocorrem também de forma seqüencial. Os *ADVServices* e os estados nos três segmentos do estado *DisplayOn* ocorrem de forma concorrente, ou seja, não existe seqüencialidade entre eles. A seguir

podemos ver na figura 6.3 como a página inicial de *MyPortal* apresenta-se a partir do *design* estabelecido.

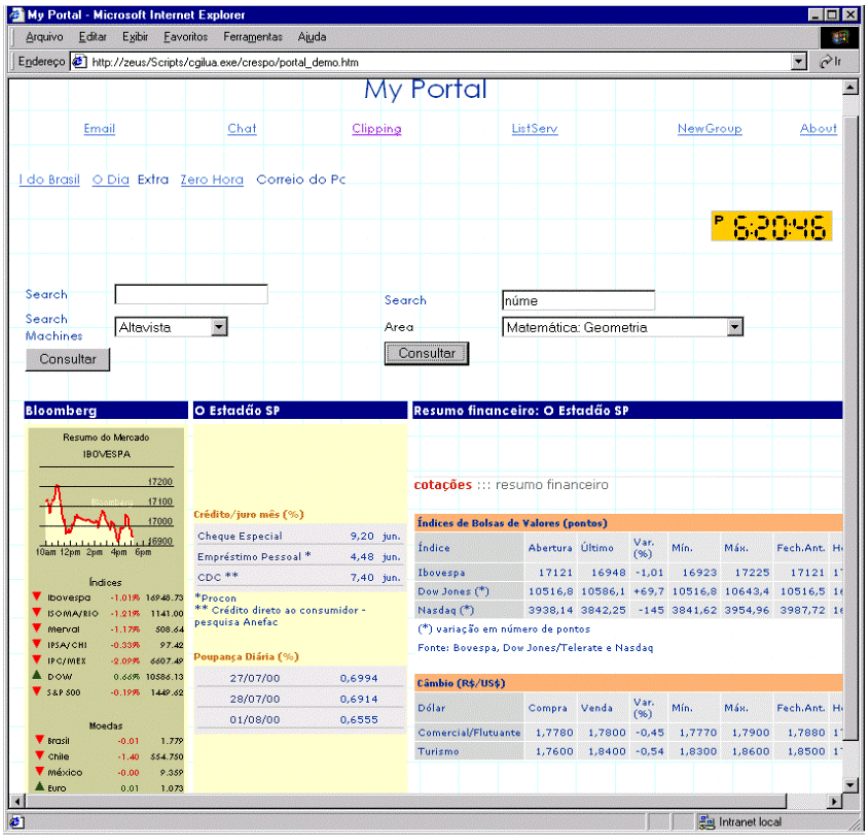

Figura 6.3 – *MyPortal*

A figura 6.3 apresenta o portal *MyPortal* com toda sua interface principal já carregada. A seguir, detalhar-se-á cada um dos *ADVServiceCharts* restantes, assim como o correspondente código em *WebCompose*.

6.3 Detalhamento dos vários subsistemas do portal *MyPortal*

Os vários subsistemas do *MyPortal* são os respectivos *ADVServiceChart* utilizados para a sua criação que serão expandidos nesta seção. A figura 6.4 apresenta o *design* dos *menus* do portal que representam os seus vários subsistemas.

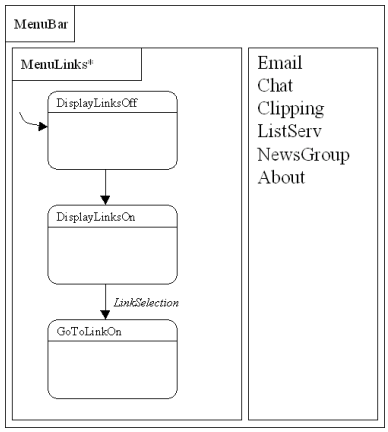

Figura 6.4 - *ADVService* do menu do protótipo

6.3.1 *ADVServiceChart*: *Search*

Este *ADVServiceChat* faz com que uma consulta possa ser realizada por meio de diversas máquinas de busca utilizando-se uma estrutura de *proxy* para isso.

A figura 6.5 apresenta o *design* deste subsistema, que realiza dois procedimentos distintos:

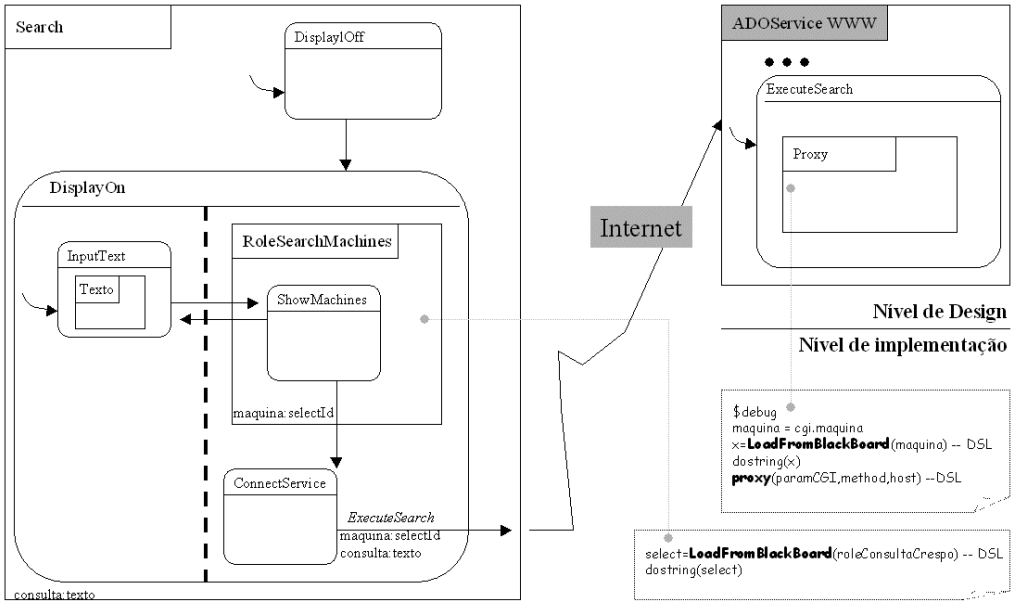

Figura 6.5 - *ADVServiceChart*: *Search*

O primeiro é realizado pelo *ADVServiceChart* chamado *RolesearchMachines* que busca no *Blackboard* um *role* que será desempenhado pelo *ADVServiceChart* de nome *Proxy*, que contém a relação das máquinas de busca que o usuário utiliza para realizar consultas na Web.

O segundo procedimento é realizado pelo *ADVServiceChart Proxy*. Aqui também obtem-se um outro *role* com as configurações das interfaces das máquina de busca já selecionadas pelo usuário. A seguir, relacionaremos esta parte do *design* aos problemas de composição em *WebFrameworks*.

Problema de composição observado: Este é um dos problemas relacionados no Capítulo 3 sobre composição em *WebFrameworks* entitulado "**sobreposição de perspectivas**", e é descrito:

> Em *WebFrameworks*, este problema relaciona-se a serviços. Logo, (ele) ocorre quando os serviços do *WebFrameworks* possuem a mesma identificação e realizam a mesma tarefa. No caso de *WebFrameworks*, uma abordagem utilizando-se *roles* para a identificação do serviço a ser utilizado apresenta uma boa alternativa, porque um serviço pode ter associado a si um conjunto de *roles* que permite visões diferenciadas para o mesmo serviço.

Nota-se que, para qualquer máquina de busca selecionada pelo usuário, o *ADVServiceChart* utilizado é sempre o mesmo: "*Proxy*". Desta forma, ocorre uma sobreposição de perspectivas para um mesmo serviço. Isto quer dizer que um mesmo serviço está sendo utilizado por meio de diferentes *roles*.

No caso da implementação do *ADVServiceChart Search*, ocorre uma composição do primeiro *role* com o segundo para formar a interface necessária para a consulta ser bem aceita pela máquina de busca selecionada. A figura 6.6 mostra o resultado de uma consulta no protótipo.

A consulta realizada foi da palavra "*java*" e a máquina de busca selecionada foi a *Altavista*[4]. Nota-se que o controle da aplicação, após ser disparada a consulta, foi passado diretamente ao *site Altavista*.

[4] http://www.altavista.com

Figura 6.6 – Resultado de uma consulta no *MyPortal*

6.3.2 *ADVServiceChart: SearchEdu*

Este *ADVServiceChat* permite que uma consulta possa ser realizada utilizando-se o ambiente EstúdioWeb[5]. A consulta a ser feita concentra-se somente nos assuntos referentes a categoria matemática e subcategorias: geometria e sistemas de numeração.

Para que esta consulta possa ser feita com êxito, é necessário adaptar duas mídias distintas e realizar um filtro de forma a termos como entrada uma tabela com o seguinte formato: *<id*, *url*, *data*, *descrição*, *obs>* e criar uma tabela auxiliar com o formato *<url* e *descrição>*. Após a tabela resultante estar criada e com os dados inseridos, uma *query* é submetida ao banco de dados de forma a retornar as *tuplas* referentes a uma consulta na tabela auxiliar.

A figura 6.7 mostra o *design* do *ADVServiceChart SearchEdu*.

[5] http://www.estudioweb.com.br

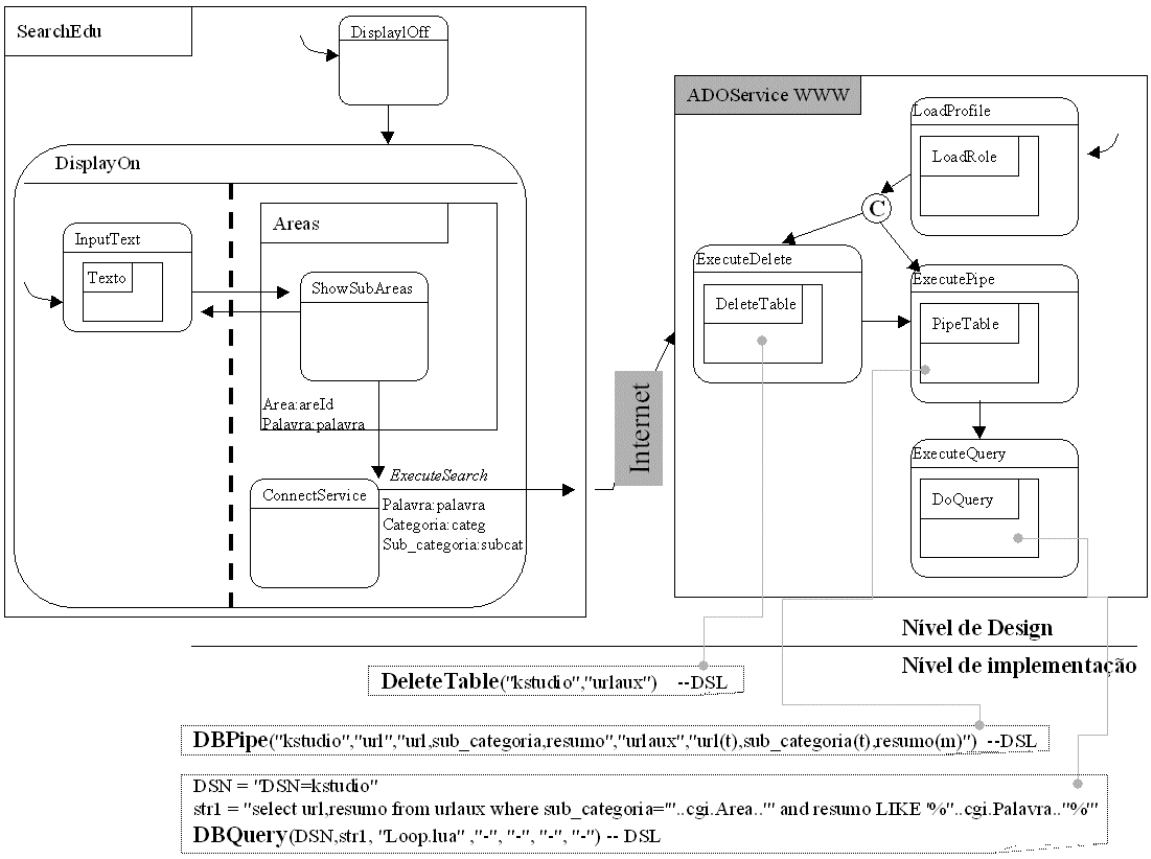

Figura 6.7 - *ADVServiceChart* de *SearchEdu*

O *design* de *DisplayOn* no *ADVService SearchEdu* é muito semelhante ao *Search* anterior, logo daremos ênfase à explicação do *design* dentro do *ADOService* WWW.

Quando o usuário escolhe a subcategoria e digita uma palavra para consulta, o controle passa para o *ADOService* de forma que vários processos possam ser executados. Nesta fase, o *role* a ser desempenhado é obtido do *Blackboard* e uma condição é estabelecida para a execução ou não da exclusão da tabela a ser criada. Caso a tabela exista, esta seria previamente excluída para ser, então, processado o *ADVService PipeTable*. O *PipeTable* transforma e filtra os dados provenientes de uma tabela inicial para uma outra tabela auxiliar com os dados requisitados da anterior. Desta forma, uma aplicação que necessitava da tabela resultante para poder executar seus *scripts* pode agora funcionar corretamente. O *PipeTable* atuou, então, como um *ADVService* pois possibilitou a conexão entre duas mídias distintas.

Após criada a tabela auxiliar uma visão particular dela é apresentada ao usuário por meio do *ADVService DoQuery*.

Problema de composição observado: Este é um dos problemas relacionados no Capítulo 3 sobre composição em *WebFrameworks* intitulado "**sistemas legados**", é discutido:

> A composição com sistema legado na plataforma Web remete-nos ao problemas de como um sistema que necessita de novos requisitos pode incorporá-lo em seu ambiente por meio de um sistema legado, estando este localizado em seu servidor ou remoto.

Neste caso, a composição aconteceu pela utilização de uma parte do banco de dados do sistema legado. A figura 6.8 mostra um exemplo da aplicação executando a consulta.

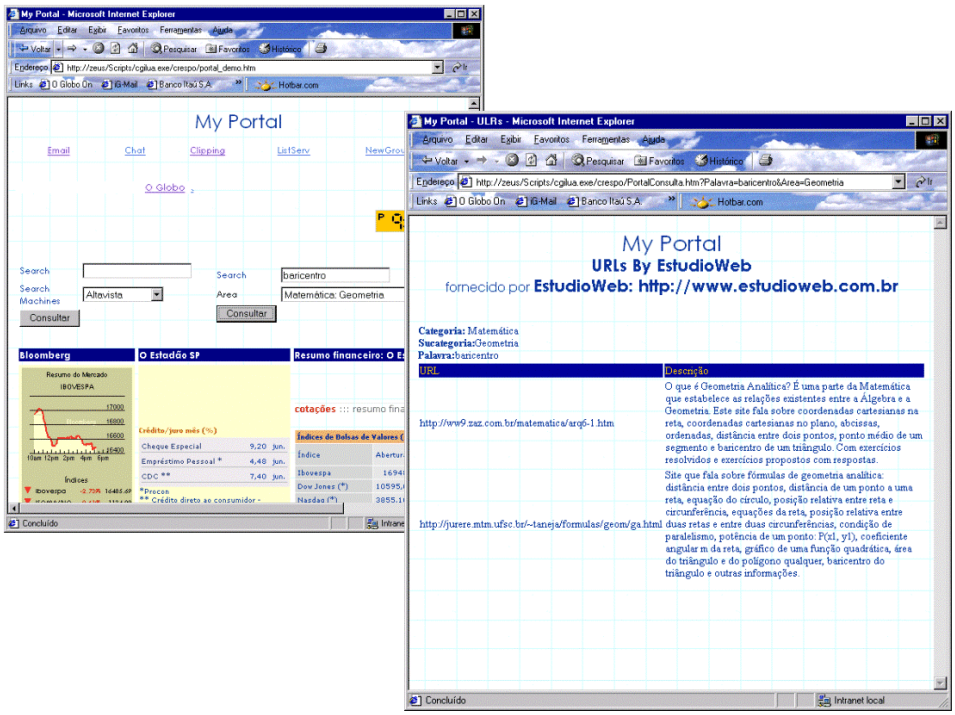

Figura 6.8 - Resultado da consulta via EstúdioWeb

A consulta executada foi da palavra "*baricentro*" e a subcategoria selecionada foi "*geometria*".

6.3.3 *ADVServiceChart*: *Time*

Este *ADVServiceChat* exibe uma hora em vários formatos(12 e 24 horas) e possibilita que, por meio de um simples cálculo, a hora possa ser exibida em diversos fusos horários. Este serviço é oferecido por um *applet java* e, para que possamos ter maior flexibilidade, o código de escrita do *applet* ficou residente no *Blackboard*.

Problema de composição observado: Este é um dos problemas relacionados no Capítulo 3 sobre composição em *WebFrameworks* intitulado "**Problema do controle do loop de execução**".

> Este problema, tal qual descrito no Capítulo 3, ocorre para *WebFrameworks* ativos, ou seja, para aqueles que possuem algum componente ou serviço ativo, tais com *applets java*. Este serviço caracteriza-se pela sua auto-execução e conexão com o seu servidor. Neste caso, não há necessidade de nenhum estímulo externo para a realizar a tarefa.

No caso do serviços de hora, todo o calculo é realizado pelo *ADVService* que chama o serviço evitando que o controle de execução fique sendo compartilhado e, desta forma, evitando o problema acima descrito. A figura 6.9 apresenta o *design* do mencionado *ADVService*.

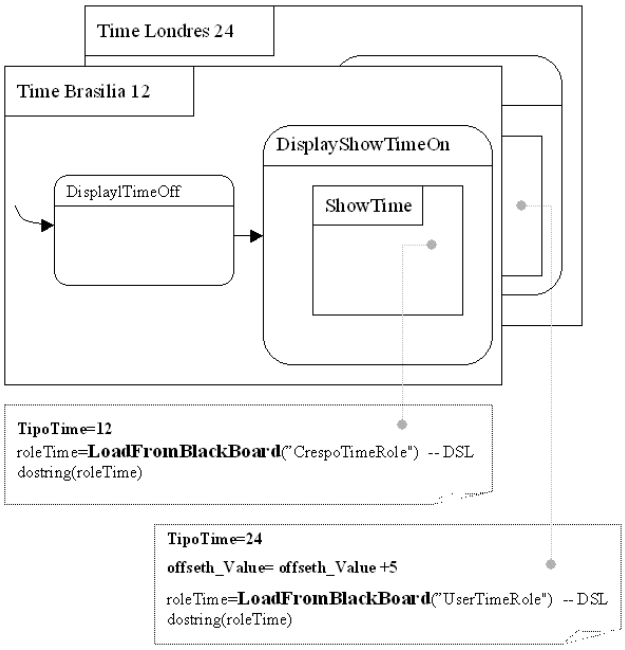

Figura 6.9 - *ADVServiceChart Time*

Para o envio dos dados do formulário de origem, utiliza-se um *ADVService* tipo *proxy*. Ele não será descrito, por ser semelhante tanto do ponto de vista de *design* quanto de implementação de código, ao do exemplo da figura 6.5.

A seguir, descrever-se-á o *ADVServiceChart* referente ao *email*.

6.3.4 *ADVServiceChart*: *Email*

Este *ADVServiceChat* permite que uma mensagem possa ser enviada por meio de um sistema legado. O formulário de origem deve repassar os seus dados para um servidor WWW que possua este serviço. Desta maneira, deve-se adaptar a interface do formulário de origem para que os dados enviados possam ser bem recebidos pela interface do *script* que executa o serviço de envio de mensagens. A figura 6.10 apresenta o *design* deste *ADVServiceChart*.

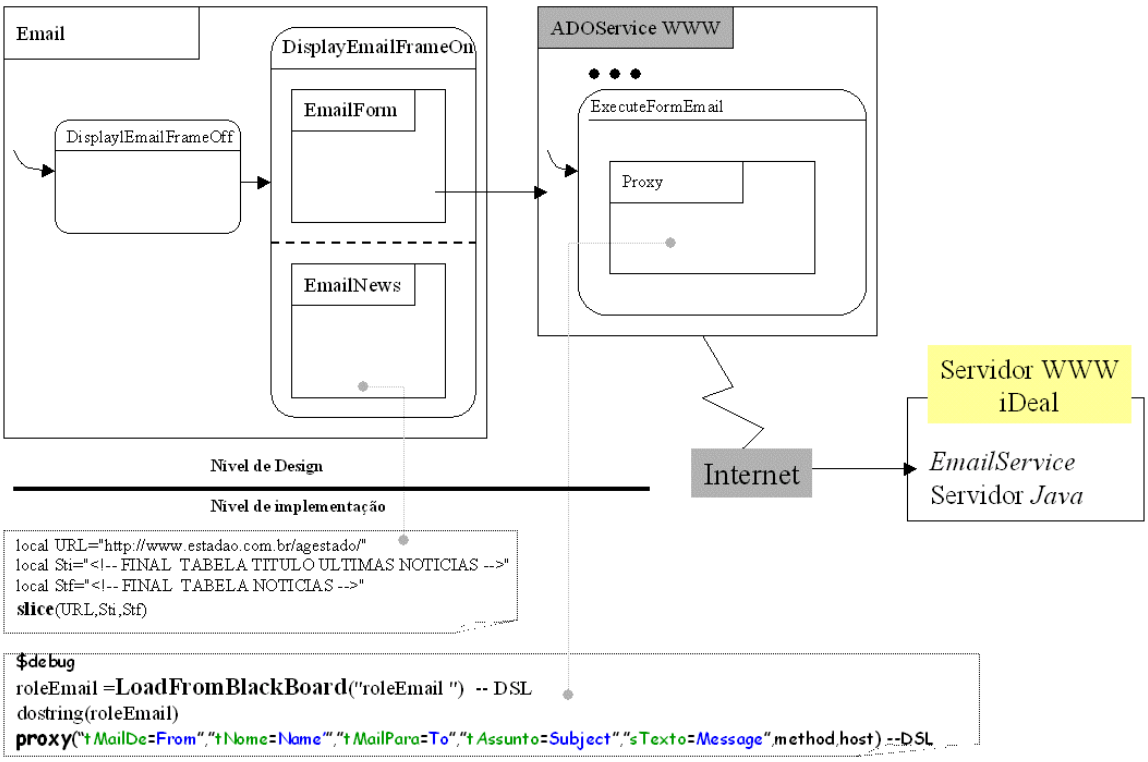

Figura 6.10 - *ADVServiceChart - Email*

Na figura 6.10, o *ADVServiceChart email* apresenta, no seu estado *DisplayEmailFrameOn,* a montagem de um *frame* dentro do *browser,* onde apresenta-se o formulário para o envio de *email* e, no outro, utiliza-se um outro *ADVServiceChart* chamado *EmailNews*. Este *ADVServiceChart* utiliza, basicamente, em sua implementação o comando *slice* da *WebCompose* como pode ser visto na figura acima.

O *slice* estabelece uma conexão *HTTP* com o servidor que detém a informação a ser recortada de sua página *HTML*. Quando o servidor responde a um pedido feito pelo *slice*, este intercepta a resposta, armazenando-a em seguida, para realizar o corte

necessário e adaptar os *links* relativos para fixos. Depois destas alterações, a página é enviada para o *browser* do cliente. A figura 6.11 mostra o funcionamento do *slice*.

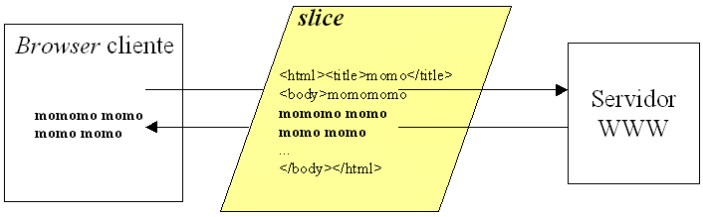

Figura 6.11 – *Slice*

Como pode ser visto na figura 6.11, o *slice* retira somente um trecho selecionado da página que o servidor *HTTP* envia ao *browser* que a requisitou.

Os seguintes *ADVServices* utilizam o *slice* em sua implementação: *Bloomberg*, *Estadão1* e *Estadão2*. Tanto o *design* como a implementação são semelhantes ao *ADVService EmailNews*. A figura 6.12, a seguir, apresenta a interface do *design* no protótipo.

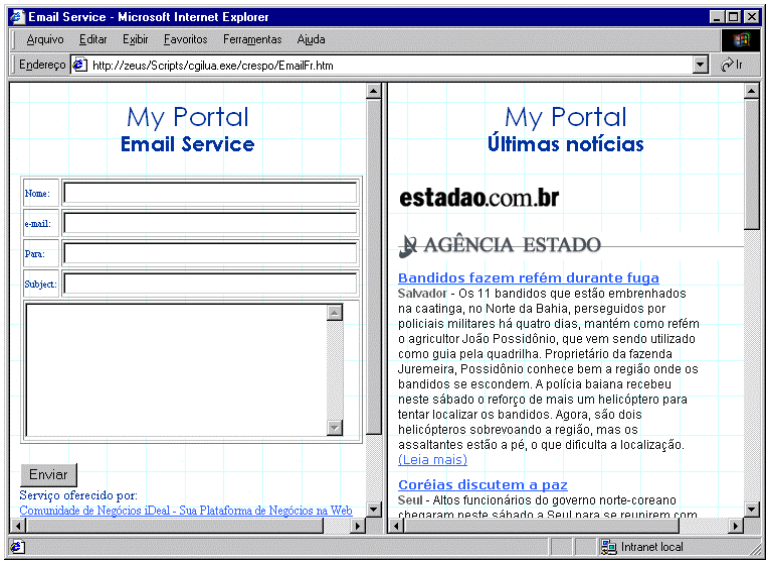

Figura 6.12 - *MyPortal - Email*

A seguir, vamos apresentaremos o *design* do serviço de *Chat* utilizado no estudo de caso.

6.3.5 *ADVServiceChart: Chat*

Este *ADVServiceChat* permite a realização de uma seção de *chat*. Um sistema legado, o *turbo chat* da IG[6], é utilizado para o serviço de *chat*.

[6] http://www.ig.com.br

A figura 6.13 apresenta o *design* deste módulo do portal *MyPortal*.

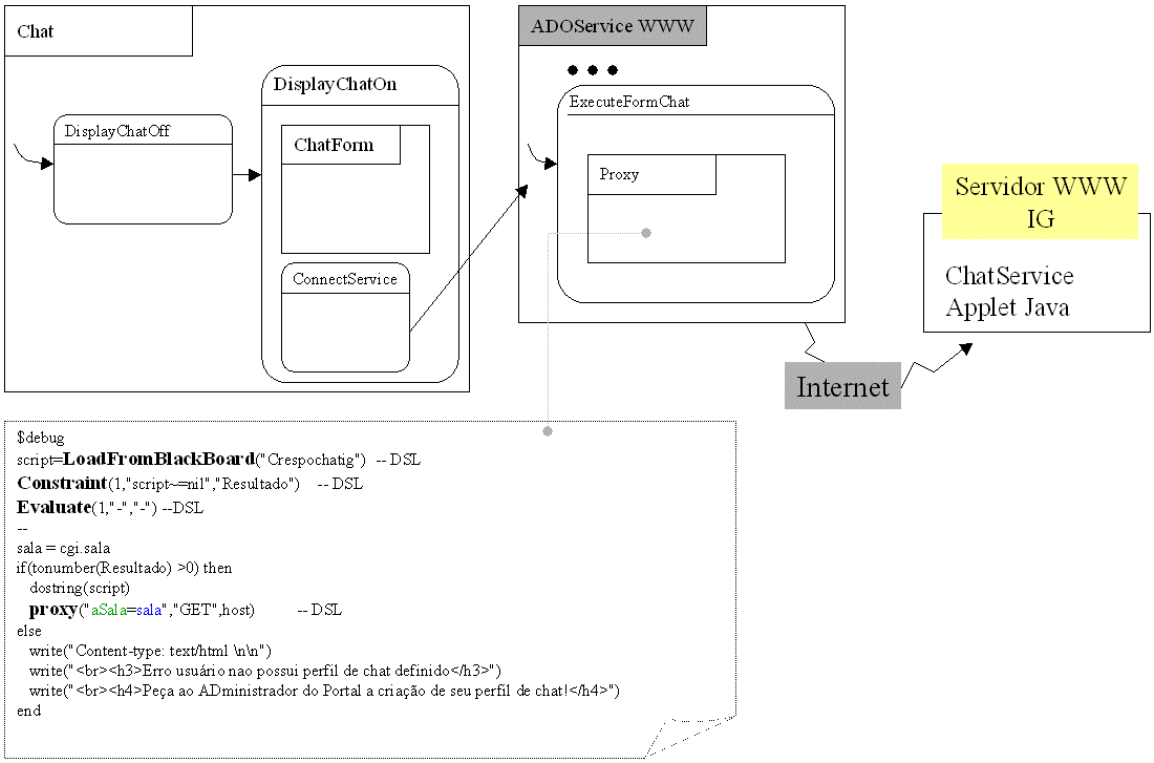

Figura 6.13 - *ADVServiceChart – Chat*

O *design* da figura 6.13 utiliza um *ADVService Proxy* para enviar os dados a respeito da sala e da interface do *applet java* que implementa o serviço de *chat*.

Quando o serviço de *Proxy* é utilizado, o *role* a ser desempenhado já foi obtido do *Blackboard*.

A estrutura de *Blackboard* mostra-se bastante satisfatória para esta forma de *scripts* se comunicarem.

A figura 6.14 exemplifica o que foi dito.

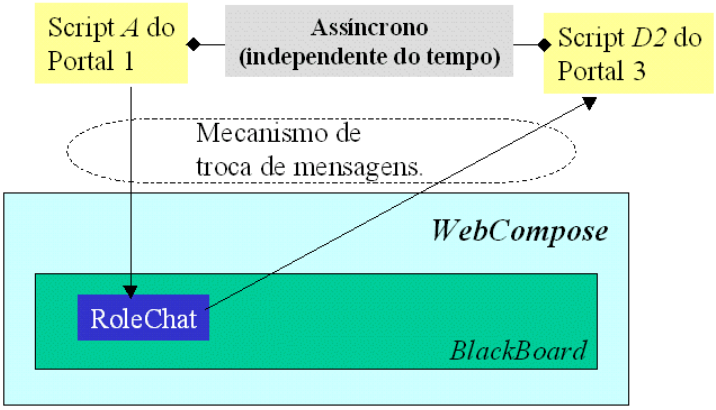

Figura 6.14 – *Blackboard*

Como pode ser visto na figura 6.14, todo o processo de troca de mensagens de forma assíncrona entre os diversos *scripts* residentes no *ADOService* WWW é apoiado pela *DSL WebCompose*.

Os demais *ADVServiceCharts Clipping* e listas utilizam procedimentos semelhantes a este e usam um *ADVService* tipo *Proxy* para poderem estabelecer contato e enviar seus dados para os respectivos serviços de *clipping* e listas.

6.4 Problemas de composição observados no estudo de caso

Neste estudo de caso, os seguintes problemas de composição em *WebFrameworks* puderam ser observados.

ADVServiceCharts	**Problema**
Search	Sobreposição de perspectivas, sistemas legados, troca de mensagens e assincronismo.
SearchEdu	Sistemas legados.
Time	Controle do *loop* de execução.
Email	Sistemas legados.
Chat	Sistemas legados, assincronismo e troca de mensagens.
Bloomberg, Estadão1 e Estadão2	Carência de requisitos.

Tabela 6.2 - Problemas de composição observados no estudo de caso

6.5 Detalhes da implementação do estudo de caso

Nesta seção, apresentaremos o código em *WebCompose* dos diversos subsistemas que compõem *MyPortal*.

6.5.1 – Interface principal de *MyPortal*

A interface principal de *MyPortal* é composta basicamente, de código *HTML* mesclado com códigos da *WebCompose*. Dar-se-á ênfase nas partes referentes a *DSL*. Apresentaremos a seguir, em *Webcompose*, o *design* da figura 6.2.

Portal_Demo.htm

```
<html>
<head>
</head>
<body background="blutextb.gif" bgcolor="#FFFFFF" text="#003399" link="#3366FF"
<!--mstheme--><font face="Century Gothic, Arial, Helvetica">
<br></center>
```

...

```
...
<!-- ========= BANNER - MARQUEE ============================-->
<marquee bgcolor="#FFFFFF" scrollamount="3" width="40%">
<!-- ----------------Buscando um role para desempenhar -------------------- -->
<!--$$
roleBannerCrespo=LoadFromBlackBoard("BannerCrespo")  -- DSL
print(roleBannerCrespo)
$$-->
<!-- --------------------------- role já executado ----------------------- -->
</marquee>
<!--========== FIM ==========================================-->

<!-- === HORA CERTA de acordo com o role do usuario=========================-->
<!--$$
TipoTime=12                          -- Definindo o comportamento do applet
roleTime=LoadFromBlackBoard("hora")  -- DSL
dostring(roleTime)
$$-->
<!--========== FIM ==========================================-->

...

...
<!--FORMULARIO PARA BUSCA NA WEB VIA OUTRAS MAQUINAS DE BUSCA-->
<!--=========================================================-->
<form method="POST" action="busca.lua">

...
<!--$$
select=LoadFromBlackBoard(roleConsultaCrespo) -- DSL
dostring(select)
$$-->
<!--FIM DO FORMULARIO PARA BUSCA NA WEB VIA OUTRAS MAQUINAS DE BUSCA-->
<!--======================== FIM ================================-->
<td  width="31%"  bgcolor="#FFFFCC"><!--mstheme--><font  face="Century  Gothic,  Arial,
Helvetica">
               <table border=0>
               <tr><td>
<!--$$
local URL="http://www.bloomberg.com/br/"
local Sti="<!---------------START OF TEASER----------->"
local Stf="Preços da moeda local em relação ao dólar"
slice(URL,Sti,Stf) -- DSL
$$-->
</TD></TR></table></table>
<!--mstheme--></font></td>
<td  width="35%"  bgcolor="#FFFFCC"><!--mstheme--><font  face="Century  Gothic,  Arial,
Helvetica">
<!--$$
local URL="http://www.estadao.com.br/economia/financas/"
local Sti="<!-- INICIO TABELA CONTEUDO CREDITO -->"
local Stf="<!-- FINAL TABELA CONTEUDO POUPANCA -->"
slice(URL,Sti,Stf) --DSL
$$-->

       <!--mstheme--></font></td>
       <td>
<!--$$
local URL="http://www.estadao.com.br/economia/financas/cotacoes/resumo.htm"
local Sti="<!-- INICIO TABELA SUPERIOR -->"
local Stf="Fonte: Ag&ecirc;ncia Estado/Broadcast"
slice(URL,Sti,Stf) --DSL
$$-->
       </td>
  </tr>
</table>
<!--mstheme--><font face="Century Gothic, Arial, Helvetica"><!--mstheme--></font>
</body>
</html>
```

---------- Fim de Portal_Demo.htm----------

6.5.2 – Interface do *ADVServiceChart Search*

A interface deste subsistema, representada na figura 6.5, em *WebCompose:*

--- *ADVServiceChart RoleSearchMachines*

```
select=LoadFromBlackBoard(roleConsultaCrespo) -- DSL
dostring(select)
```

--- *ADVServiceChart Proxy*

```
$debug
maquina = cgi.maquina
x=LoadFromBlackBoard(maquina) -- DSL
dostring(x)
proxy(paramCGI,method,host) --DSL
```

Este *script* esta sendo executado no *ADOService* WWW, que é o servidor que possui *CGILua + WebCompose.*

6.5.3 – Interface do *ADVServiceChart SearchEdu*

A interface deste subsistema, representada na figura 6.7, em *WebCompose:*

--- *ADVServiceChart ExecuteDelete*

```
DeleteTable("kstudio","urlaux")    --DSL
```

--- *ADVServiceChart ExecutePipe*

```
DBPipe("kstudio","url","url,sub_categoria,resumo","urlaux","url(t),sub_categoria(t),re
sumo(m)")  --DSL
```

--- *ADVServiceChart ExecutePipe*

```
DSN = "DSN=kstudio"
str1 = "select url,resumo from urlaux where sub_categoria='"..cgi.Area.."' and resumo
LIKE '%"..cgi.Palavra.."%'"
DBQuery(DSN,str1, "Loop.lua" ,"-", "-", "-", "-") -- DSL
```

6.5.4 – Interface do *ADVServiceChart Time*

A interface deste subsistema, representada na figura 6.9, em *WebCompose:*

--- *ADVServiceChart ShowTime*

```
TipoTime=12
roleTime=LoadFromBlackBoard(”CrespoTimeRole")  -- DSL
dostring(roleTime)
```

Para a implementação do mesmo ADVServiceChart mostrando a hora em Londres, o código seria:

```
TipoTime=24
offseth_Value= offseth_Value +5
roleTime=LoadFromBlackBoard(”UserTimeRole")  -- DSL
dostring(roleTime)
```

6.5.5 – Interface do *ADVServiceChart Email*

A interface deste sub-sistema, representada na figura 6.10, em *WebCompose:*

--- *ADVServiceChart EmailNews*

```
local URL="http://www.estadao.com.br/agestado/"
local Sti="<!-- FINAL  TABELA TITULO ULTIMAS NOTICIAS -->"
local Stf="<!-- FINAL  TABELA NOTICIAS -->"
slice(URL,Sti,Stf)
```

--- *ADVServiceChart Proxy*

```
$debug
roleEmail =LoadFromBlackBoard("roleEmail ")  -- DSL
dostring(roleEmail)
proxy(“tMailDe=From”,”tNome=Name’”,”tMailPara=To”,”tAssunto=Subject”,”sTexto=Message”,
method,host) --DSL
```

Este *script* está sendo executado no *ADOService* WWW.

6.5.5 – Interface do *ADVServiceChart Chat*

A interface deste subsistema, representada na figura 6.13, em *WebCompose:*

--- *ADVServiceChart ExecuteFomrChat*

```
$debug
script=LoadFromBlackBoard("Crespochatig")  -- DSL
Constraint(1,"script~=nil","Resultado")    -- DSL
Evaluate(1,"-","-") --DSL
--
sala = cgi.sala
if(tonumber(Resultado) >0) then
   dostring(script)
   proxy("aSala=sala","GET",host)          -- DSL
else
   write("Content-type: text/html \n\n")
   write("<br><h3>Erro usuário nao possui perfil de chat definido</h3>")
   write("<br><h4>Peça ao ADministrador do Portal a criação de seu perfil de
chat!</h4>")
end
```

Este *script* está sendo executado no *ADOService* WWW.

6.6 Conclusão

Este capítulo descreveu o estudo de caso para a criação de um protótipo de um portal horizontal utilizando-se a *DSL WebCompose*, que procura cobrir todos os aspectos relacionados à composição de *WebFrameworks* relacionados no Capítulo 3 da tese.

Os *ADVServiceChart* foram utilizados para a representação do *design* do portal *MyPortal* e mostrou-se muito eficiente na medida em que o mapeamento entre o *design* e a implementação foi de um para um. Em todos os *ADVServicesCharts* apresentados, o correspondente código em *WebCompose* estava presente para ilustrar o mapeamento entre o nível de *design* e o de implementação.

Outra constatação que pode ser observada, é a simplicidade com a qual se realiza a passagem do *design* para a implementação usando a notação *ADVServiceChart* e a *DSL WebCompose*.

A *WebCompose* mostrou-se suficientemente robusta para poder implementar um grande conjunto de funcionalidades do portal *MyPortal*, permitindo que o portal seja rapidamente criado sem que nenhum dos serviços utilizados tenha sido implementado pelo criador do portal.

A utilização do conceito de *roles*, mostrou-se evidente em várias partes, e sua implementação foi beneficiada pelo uso da estrutura de *Blackboard* que a *WebCompose* implementa. Esta estrutura permitiu também que *scripts* pudessem deixar informações sobre *roles*, identificação de máquinas de busca, etc. para que

outros *scripts* pudessem, de forma assíncrona, obter as informações para poderem desempenhar papéis variados em suas tarefas.

O conceito de *WebFrameworks* pode ser facilmente implementado pela utilização de um conjunto básico de serviços por meio da *DSL WebCompose*, que permite com que vários *scripts* que antes eram executados através de dezenas de linhas de código fonte, sejam, em sua maioria, criados com apenas uma única linha. Comandos como *slice* são impossíveis de ser implementados em linguagens de *script* no ambiente da Web. Comandos como *DBPipe* e *DBQuery,* que são executados com uma única linha, em outras linguagens de *script* demandam dezenas de linhas cuja a correta execução não pode ser assegurada.

Hoje a *WebCompose* é parte integrante dos projetos realizados no *TecComm*[7]/PUC-Rio. O projeto *PortalWare* e o próprio *site* do grupo *TecComm* utilizam a *WebCompose* em alguns de seus módulos.

[7] http://www.teccomm.les.inf.puc-rio.br

Capítulo 7

Conclusões e Trabalhos Futuros

Resumo

Este capítulo apresenta as conclusões da tese bem como suas contribuições e os trabalhos futuros dela decorrentes.

7.1 Introdução

Esta tese apresentou os problemas sobre composição de *frameworks* em plataforma Web inicialmente discutido por Mattsson [Mattsson 2000] e Wolfgang Pree[Pree 1999a] e [Pree 1999b] em *frameworks* orientados a objetos.

Como referencial teórico para abordar o problema, utilizou-se *ADVs/ADOs* [Cowan and Lucena 1995], [Alencar, Cowan and Lucena 1995a], [Alencar, Cowan and Lucena 1995b], [Alencar, Cowan and Lucena 1995c] e [Alencar, Cowan, Lucena et alli 1995], que enfatiza reutilização e *"separation of concerns*", e define um conjunto de propriedades para a construção de um *design* flexível. A teoria serviu como base para a criação de uma *DSL* chamada *WebCompose* que permite a composição de *WebFrameworks* para a geração de novas aplicações em plataforma Web.

Um estudo de caso foi implementado, de forma a consolidar a *DSL* e validar a sua utilização no mundo real. O estudo de caso, utilizou todos os comandos da *DSL* e abordou todos os problemas relacionados a composição em *WebFrameworks*. Também foi possível observar, a facilidade proporcionada pela *WebCompose* para a produção de aplicações complexas na Web, a partir de serviços já disponíveis. Para a implementação do estudo de caso, não foram necessárias mais do que 50 linhas de código fontes, número esse muito baixo se comparado com alguma outra linguagem utilizada para implementar tais facilidades.

7.2 Contribuições

- Adaptação dos problemas de composição em *frameworks* orientados a objetos para a plataforma Web. Desta forma, uma nova definição para *frameworks* em plataforma Web foi criada, chamada *WebFrameworks*. Este novo conceito, produz um grande impacto na forma pelo qual as aplicações podem ser geradas a partir de

um conjunto básico de serviços presentes na Web. Esta nova maneira de produzir aplicações na Web, simplifica e facilita o trabalho de manutenção e expansão de aplicações já existentes.

- Identificação de novos problemas de composição em *WebFrameworks*. Estes problemas, em sua maioria, são originários de uma característica básica da Web que é a forma assíncrona com que as aplicações executam sua tarefas. Desta forma, aplicações que necessitavam conhecer algum estado antes de executarem propriamente, precisam implementar de forma não flexível, seus próprios mecanismos de comunicação e transferência de mensagens entre *scripts*. Esta tarefa é muito onerosa e nem sempre abrangente o suficiente para todas as aplicações que utilizam um determinado servidor WWW.
- Extensão da teoria de *ADVs/ADOs* como referencial teórico para os problemas de composição em *WebFrameworks*. Desta extensão novos termos foram definidos, bem como novas propriedades foram adicionadas às existentes. A teoria de *ADVs/ADOs* nos proporciona o interessante conceito de múltiplas visões para um mesmo serviço. Desta forma, um serviço pode ser utilizado de várias formas ao satisfazer um conjunto de necessidades nem sempre previstas durante a sua criação. Como exemplo podemos citar um serviço de *chat* produzido por um determinado portal e que pode ser reutilizado por uma grande variedade de outras aplicações, ou seja, podemos ter vários papéis para um mesmo serviço na Web.
- Desenvolvimento de uma *DSL* chamada *WebCompose* que possibilitou com que um conjunto básico de serviços já existentes na Web pudessem formar o *kernel* de várias aplicações nesta plataforma. A *DSL* possibilitou que a composição de serviços fosse realizada com facilidade e desta forma novas aplicações pudessem ser desenvolvidas sem causar grande impacto nos requisitos já elicitados previamente. A manutenção de aplicações Web, quando se identifica a necessidade de se adicionar novas funcionalidades através do uso da *WebCompose*, não requer o mesmo grau de esforço das abordagens mais tradicionais de desenvolvimento de *frameworks* orientados a objetos.
- Extensibilidade da *WebCompose*. A *DSL* foi implementada na forma de *framelets*, permitindo que vários de seus comandos possam ser instanciados, ou estendidos, de forma a especializá-los para a utilização em *scripts* mais particulares.

- Extensão do ambiente *CGILua* para permitir a construção e composição de aplicações na Web. A utilização da *WebCompose* juntamente com a linguagem oferecida pelo *CGILua* e Lua permite maior flexibilidade na criação de *scripts*.
- Reutilização em ponto grande de serviços na Web. A linguagem *Webcompose*, oferece mecanismos para que a construção de aplicações altamente personalizadas possam ser geradas a partir de fragmentos de informação residente nos mais variados portais da Web.

7.3 Trabalhos Futuros

Como trabalhos futuros, é possível investigar nas seguintes áreas:
Implementar um ambiente para a inserção automática dos comandos da *WebCompose* nos *scripts* para composição ou reutilização de serviços já existentes. Desta forma, a *WebCompose* pode ser utilizada como um mapeamento direto entre o *design* da aplicação e a sua implementação.

Utilizar o conceito de *espaço compartilhado*[Freeman, Hupfer, Arnold 1999] para gerar a *DSL WebCompose+* em que não se necessite informar o endereço físico dos serviços, informando apenas a sua identificação em um "*espaço*".
Generalizar o conceito de *ADOService* WWW de forma que o servidor para composição não tenha mais um endereço físico e necessite que os *scritps* para serem executados tenham que estar residentes fisicamente em um servidor WWW.
A idéia neste projeto, é utilizar *JavaSpace* [Freeman, Hupfer, Arnold 1999] para produzir espaços onde serviços possam ser requisitados e utilizados.
A figura 7.1 ilustra o que foi dito.

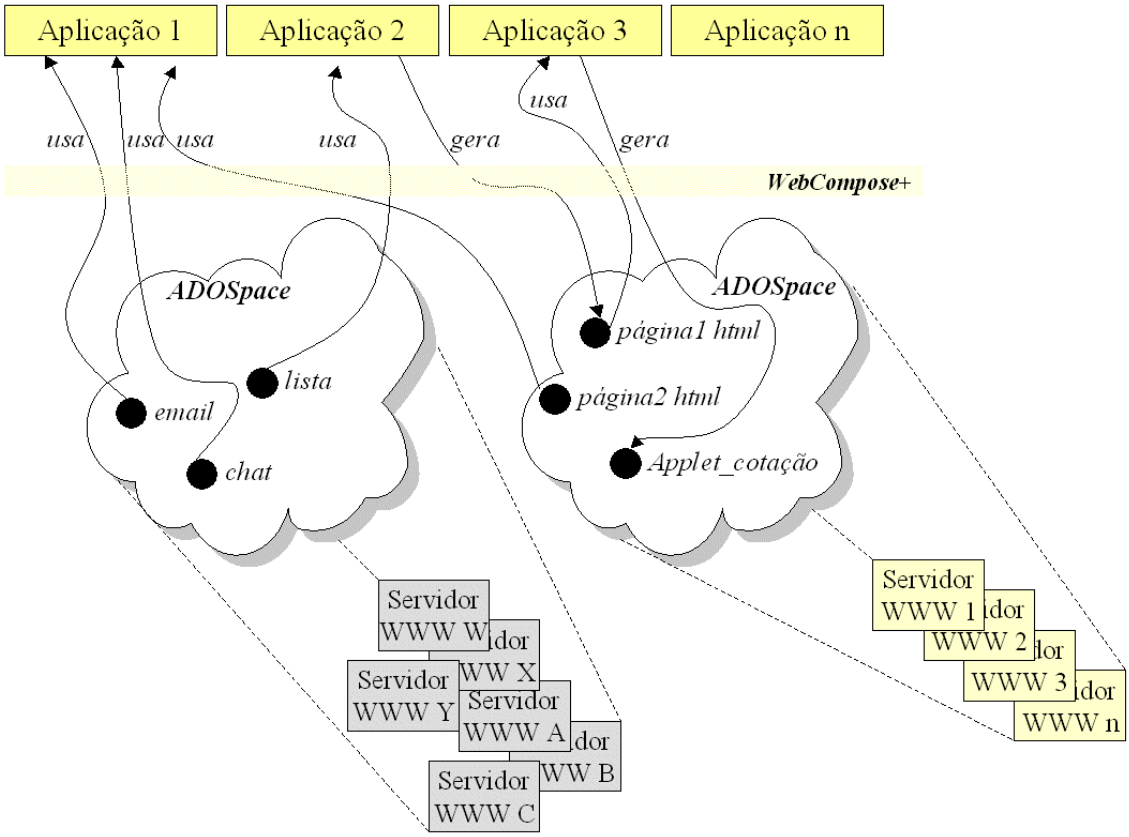

Figura 7.1 - Espaço de serviços

Como pode ser visto na figura acima, o *ADOSpace* possui um conjunto de servidores convencionais associados. Estes servidores que possuem um *IP* fixo e contém os serviços são mapeados pelo *ADOSpace* de forma que as aplicações que utilizam a *DSL WebCompose+*, não necessitam informar ou saber onde esta localizado o serviço a ser utilizado. Assim, um serviço passa a ser parte de um espaço e por meio da *DSL* (*WebCompose+*), fica facilitada a sua utilização. Hoje, para realizar composição de serviços é necessário saber fisicamente qual servidor oferece o serviço e onde ele esta localizado, o que torna a tarefa de utilização de serviço disponíveis um pouco trabalhosa.

Com o *ADOSpace* criado, e a *WebCompose+*, ficará facilitada a tarefa de criação de aplicações Web. Esta facilidade será fornecida pela simplicidade em se requisitar um serviço através de comandos que interagem com este espaço e pela nova arquitetura produzida para servidores de informação WWW.

Uma vez que os serviços possam ser requisitados de *ADOSpaces*, um *WebFramework* pode agora ser um conjunto de serviços em *ADOSpaces*.

Referências bibliográficas

[Achermann, Limpe, Schneider and Nierstrasz 2000] Franz Achermann, Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, "Piccola - a Small Composition Language," Formal Methods for Distributed Processing, an Object Oriented Approach, Howard Bowman and John Derrick. (Eds.), Cambridge University Press., 2000, to appear.

[Ainsworth, ett alli 1994] M. Ainsworth, A. H. Cruickshank, L. J. Groves, and P. J. L. Wallis, "Viewpoint specification and Z", Information and Software Technology, 36(1), pages 43-51, February 1994.

[Alencar, Cowan and Lucena 1995a] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, "Abstract data Views as a Formal Approach to Adaptable Software", Tec. Report, University of Waterloo, August 9, 1995.

[Alencar, Cowan and Lucena 1995b] P.S.C. Alencar, D.D. Cowan, C.J.P.Lucena, "A Fromal Model to Support Subject-Oriented Programming", Tec.Report, University of Waterloo, August 21, 1995.

[Alencar, Cowan and Lucena 1995c] P.S.C. Alencar, D.D. Cowan, C.J.P.Lucena, "Abstract Data Views as a Formal Approach to Subject-Oriented Programming", Tec.Report, University of Waterloo, May 12, 1995.

[Alencar, Cowan, Lucena and Nova 1995+] P.S.C. Alencar, D.D. Cowan, C.J.P.Lucena, L.C.M.Nova, "Formal specification of Reusable Interface Objects", Proceedings of the Symposium on Software Reusability (SSR'95), pages 88-86, ACM Press, 1995.

[Alencar, Cowan, Lucena et alli 1995] P.S.C.Alencar, D.D.Cowan, D.M. Germán, K.J.Lichtner, C.J.P.Lucena, L.C.M.Nova, "A Formal Approach to Design Pattern Definition & Application", Comput. Sci. Dep. , Univ.Waterloo, Waterloo, Ont., Canada, Tech.Rep., June 27, 1995.

[Bassett 1997] Paul G. Bassett, "Framing Software Reuse – Lessons From the Real World", Yourdon Press Computing Series, 1997.

[Booch, Jacobson, Rumbaugh 1998] Grady Booch, ,Jim Rumbaugh, Ivar Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, 1998.

[Bosh 1996] Jan Bosh, “Language Support for Component Communication in LayOM” , Presented at the ECOOP '96 Workshop on Component-Oriented Programming (WCOP '96), 1996.

[Bosch 1999b] Jan Bosch, “Superimposition: A Component Adaptation Technique”, Information and Software Technology, 1999.

[Bosch 1998] Jan Bosh, Specifying Frameworks and Design Patterns as Architectural Fragments, TOOLS ASIA'98.

[Bosch et alli 1999] Jan Bosch, Peter Molin, Michael Mattsson, PerOlof Bengtsson and Mohamed E. Fayad, "Framework Problems and Experiences", A Chapter in "Building Application Frameworks: Object Oriented Foundations of Framework Design" Eds: M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, 1999, pp. 55-82, 1999.

[Bouge & Francez 1998] L.Bougé, N.Francez, “A Compositional Approach to Superimposition”, Proceedings POPL’99, pp.240-249, 1988.

[Bruns, Cox 1999] David L Atkins, Thomas Ball, Glenn Bruns and Kenneth Cox, “Mawl: A domain-specific Language for From-Based Services”, IEEE Transaction on Software Engineering, Vol. 25, no. 3, May/Jun 1999.

[Buschmann 1994] F.Buschmann, “The Master-Slave Pattern”, Proceedings of the First Conference on Pattern Languages and Programming, Addison-Wesley 1994.

[Buschmann et alli 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Mchel Stal, “Pattern-Oriented Software Architectur A System of Patterns”, John Wiley & Sons, 1996

[Cardelli , Davies 1999] Luca Cardelli and Rowan Davies, “Service Combinators for Web Computing”, IEEE Transaction on Software Engineering, vol. 25, no. 3, May/Jun 1999.

[Carneiro, Coffin, Cowan e Lucena 1993] L.M.F. Carneiro, M.H.Coffin, D.D. Cowan, and C.J.P.Lucena, “User interface high-order architectural models”, Comput. Sci. Dep. Univ. Waterloo, Waterloo, Ont., Canada, Tech. Rep. 93-14, 1993.

[Carneiro, Cowan and Lucena 1993a] L.M.F. Carneiro, D.D.Cowan, C.J.P.Lucena, “ADVcharts: A Visual Formalism for Describing Abstract Data Views”, Technical Report 93-20, Computer Science Department, University of Waterloo, waterloo, Ontario, Canada, April 1993.

[Carneiro, Cowan and Lucena 1993b] L.M.F. Carneiro, D.D.Cowan, C.J.P.Lucena," ADVcharts: A Graphical Specification for Abstract Data Views, CASCON'93, Toronto, Ontario, Canada, October, pages 84-96, 1993.

[CGILua site] http://www.tecgraf.puc-rio.br/cgilua

[Coleman, Hayes and Bear 1992] Derek Coleman, Fiona Hayes, and Stephen Bear, Introducing Objectcharts or How to Use Statechrts in Object-Oriented Desing", IEEE Transaction on Software Engineering, 18(1):9-18, January 1992.

[Cowan, Lucena 1995]; Donald D. Cowan , Carlos J. P. Lucena, "Abstract Data Views: An Interface Specification Concenpt to Enhance Design for Reuse", IEEE Transaction on Software Engineering, vol. 21, No. 3, March 1995.

[Cowan ett alli 1992] D.D. Cowan, et al.,"Program Design using abstract data views – An illustrative example", Comput. Sci. Dep. , Univ.Waterloo, Waterloo, Ont., Canada, Tech.Rep. 92-54, Dec. 1992.

[Cowan, Barbosa, Ierusalimschy, Lucena and Oliveira 1992] D.D.Cowan, L.F. Barbosa, R. Ierusalimschy, C.J.P.Lucena and S.B.Oliveira, "Program Design Using Abstract Data Views-An Illustrative Example.", Technical Report 92-54, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, December 1992.

[Cowan, Mackie, Pianosi and Smit 1991] D.D.Cowan, E.W. Mackie, G.M. Pianosi, and V.Smit, "Rita-an Editor and User Interface for Manipulating Structured Documents.", Eletronic Publishing, 4(3):125-150, 1991.

[Crespo, Fontoura, Lucena e Moura 1998] S. Crespo, M. F. Fontoura, C. J. Lucena, and L. Moura, "ALADIN: An Architecture for Learningware Applications Design and Instantiation", submitted to WWW Journal, 1998.

[Fayad 1999] Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson, "Building Application Frameworks: Object Oriented Foundations of Framework Design" Eds: M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, 1999, pp. 03-28, 1999.

[Fayad, Schimidt and Johnson 1999] Mohamed Fayad , C. Douglas Schimidt and E. Raph Johnson; "Implementing Application Frameworks – Object-Oriented Frameworks at Work", Chapter 18, Wiley, 1999.

[Finkelstein 1996] Antony Finkelstein, "Relating Viewpoints" in Proc. ACM SIGSOFT 96 Workshop, Viewpoints 96, (ACM Press), pp157. , 1996

[Filkelstein ett alli 1992] A. Filkelstein, , J. Kramer, B. Nuseibeh , L. Filkelstein and M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in

System Development", Internation Journal of Software Engineering and Knowledge Engineering 2(1):31-58, World Scientific Publishing Co, March 1992.

[Flanagan 1998] David Flanagan, " JavaScript: The Definitive Guide, 3rd Edition 3rd Edition", O'Reilly & Associates, Inc. , June 1998.

[Fontoura 1999] Marcus Felipe M. C. Fontoura, A Systematic Approach for Framework Development, Tese de Doutorado, Departamento de Informática, PUC-Rio, 1999.

[Fortunato 1999] Fortunato, L. "2BuyNet: Um Framework para Instanciação e Administração de Lojas para a Internet" Dissertação de Mestrado, Departamento de Informática, PUC-Rio, 1999.

[Freeman, Hupfer, Arnold 1999] Eric Freeman, Susanne Hupfer, Ken Arnold, " JavaSpaces Principles, Patterns, and Practice", Addison-Wesley, 1999.

[Froehlich 1999 ett alli] G. Froehlich, H.J. Hoover, L. Liu and P. Sorenson, "Reusing Hooks", A Chapter in "Building Application Frameworks: Object Oriented Foundations of Framework Design" Eds: M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, 1999, pp. 218-236, 1999.

[Froehlich, Hoover, Liu and Sorenson 1997a] G. Froehlich, H.J. Hoover, L. Liu and P. Sorenson, "Hooking into Object-Oriented Application Frameworks", To appear in the Proceedings of the 1997 International Conference on Software Engineering. Boston (May 1997).

[Froehlich, Hoover, Liu and Sorenson 1997b] G. Froehlich, H.J. Hoover, L. Liu and P. Sorenson, Reusing Application Frameworks Through Hooks, Communications of the ACM special issue on object-oriented frameworks, 1997.

[Gamma et alli 1995] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[Guelich,Birznieks 2000] Scott Guelich, Gunther Birznieks, "CGI Programming with Perl, Second Edition", Published by O'Reilly & Associates, 2000.

[Haskell home page] http://www.haskell.org/

[Harel 1987] David Harel, "Statecharts: a Visual Formalism for Complex Systems.", Science of Computer Programming, 8:231-274, 1987.

[Helm 1990] Richard Helm, Ian M. Holland and Dipayan Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Programming Systems, Language and Applications", OOPSLA'90. ACM Press, Page 169-180, 1990..

[Hester 1999] Anna Magdalena Hester, "A ferramenta CGILua em múltiplas APIs com servidor HTTP", Tese de Mestrado, Departamento de Informática - PUC-Rio, Junho de 1999.

[Hester, Borges and Ierusalimschy 1997] Anna M. Hester, Renato Borges and Roberto Ierusalimschy. Cgilua: A multi-paradigmatic tool for creating dynamic WWW pages. In XI Simpósio Brasileiro de Engenharia de Software, 1997.

[Hester, Borges and Ierusalimschy 1998a] Anna Hester, Renato Borges and Roberto Ierusalimschy," Building flexible and extensible Web applications with Lua", Journal of Universal Computer Science, 4 (9):748-762, 1998.

[Hester, Borges and Ierusalimschy 1998b] Anna Hester, Renato Borges and Roberto Ierusalimschy, "CGILua: A multi-paradigmatic tool for creating dynamic WWW pages", XI Simpósio Brasileiro de Engenharia de Software, páginas 347-360, Fortaleza, CE, 1997.

[Ierusalimschy, Figueiredo, Celes] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes Filho ," Lua-an extensible extension language", *Software: Practice & Experience* **26** #6 (1996) 635-652. John Wiley & Sons, Ltd., 1996 .

[Ierusalimschy, Figueiredo,Celes 1996] Ierusalimschy, R.,Figueiredo, L. H.,Celes, W.: "Lua - An extensible extension language", Software: Pratice & Experience, 26(6):635-652,1996.

[ITU 1995] ITU Recommendation X.901-904 – ISO/IEC 10746 1-4, "Open Distributed Processing – Referring Model Parts 1-4", July 1995.

[Jacobsen, Kristensen,Nowack 1997] E. E. Jacobsen, B. B. Kristensen & P. Nowack: "Patterns in the Analysis, Design and Implementation of Frameworks", In Proceedings of the Twenty-First Annual International Computer Software and Application Conference, (COMPSAC'97), Washington D.C., USA, 1997.

[Johnson 1991] R.E Johnson, Reusing Object-Oriented Design, University of Illinois, Technical Report UIUCDCS 91-1696, 1991.

[Johnson, Foote 1998] R.E Johnson, B. Foote, Designing Reusable Classes, Journal of Object-Oriented Programming, June 1988.

[Kistler, Marais 1998] T. Kistler and H. Marais, "WebL – A Programming Language for the Web", Computer Networks and ISDN Systems, Proc. WWW7 Conf. , vol. 30, no. 1-7, pp. 259-270, Apr. 1998.

[Knowledge – Internet Statistics site]

[Krasner 1988] G.E. Krasner, S.T. Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”, Journal of Object-Oriented Programming, Vol.1, No3, August-September 1988.

[Lua site] http://www.tecgraf.puc-rio.br/lua

[Lucena, Cowan and Potengy 1992] C.J.P. Lucena, D.D.Cowan, A.B. Potengy, “A programming model for user interface compositions”, in Anais do V Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, SIBGRAPI’92, (Águas de Lindóia, SP, Brazil), Nov. 1992.

[Lumpe, Achermann and Nierstrasz 2000] Markus Lumpe, Franz Achermann and Oscar Nierstrasz, “A Formal Language for Composition,” Foundations of Component Based Systems, Gary Leavens and Murali Sitaraman (Eds.), pp. 69—90, Cambridge University Press, 2000.

[Lundberg, Mattsson 1996] C. Lundberg, M. Mattsson, On Using Legacy Software Components with Object-Oriented Frameworks, Proceedings of Systemarkitekturer'96, Borås, Sweden, 1996.

[Maia 2000] Ayrton Maia, “CommercePipe: Um Framework para criação de Canais Comerciais Consumer to Business na Internet” Dissertação de Mestrado, Departamento de Informática, PUC-Rio, 1999.

[Mattsson 1996] M. Mattsson, “Object-oriented Frameworks - A survey of methodological issues”, Licentiate Thesis, Department of Computer Science, Lund University, CODEN: LUTEDX/(TECS-3066)/1-130/(1996), also as Technical Report, LU-CS-TR: 96-167, Department of Computer Science, Lund University, 1996.

[Mattsson ett alli 1999] Michael Mattsson, Jan Bosch.and Mohamed E. Fayad, "Framework Integration: Problems, Causes and Solutions" , Communications of ACM, Vol. 42, No 10 (Oct. 1999), pp 80-87, 1999.

[Mattsson 2000] Michael Mattsson, “Evolution and Composition Object-Oriented Frameworks”, PhD Thesis, University of Karlskrona/Ronneby, Department of Software Engineering and Computer Science, 2000.

[Mattsson, Bosh 1999] Michael Mattsson and Jan Bosch, “Composition Problems, Causes and Solutions”, A Chapter in "Building Application Frameworks: Object Oriented Foundations of Framework Design" Eds: M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, 1999, pp. 467-487, 1999.

[Meunier 1995] R. Meunier, “The Pipes and Filters Architecture”, Proceedings of the First Conference on Pattern Language and Programming”, Addison-Wesley, 1994.

[Mezini, Seiter, Lieberherr 2000] Mira Mezini and Linda M. Seiter and Karl J. Lieberherr, "Software Architectures and Component Technology: The State of the Art in Research and Practice", Publisher: Kluwer Academic Publishers, Editor: Mehmet Aksit, 2000.

[Myers 1991] B.A. Myers, "Separating Application code from toolkits: Eliminating the spagueti of call-backs", in UIST- 4th Annu. Symp. User Interface Software Tecnol., pp. 211-220, 1991.

[NetSizer site] http://www.netsizer.com/

[Nua site] http://www.nua.ie/surveys/index.cgi?f=FS&cat_id=20

[OriginalSim 1997] http://www.originalsim.com/OriginalSim/Press/ATES.htm

[Ousterhout 1998] John K. Ousterhout, " Scripting: Higher-Level Programming for the 21st Century", IEEE Computer pp. 23-30, March 1998.

[Pree 1995] "Design Patterns for Object-Oriented Software Development", Addison-Wesley, 1995.

[Pree 1999a] Wolfgang Pree, Kai Koskimies, " Framelets-Small Is Beautiful", A Chapter in "Building Application Frameworks: Object Oriented Foundations of Framework Design" Eds: M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, 1999, pp. 411, 1999.

[Pree 1999b] Rearchitecturing Legacy Systems—Concepts & Case Study, WICSA '99 (First Working IFIP Conference on Software Architecture) San Antonio, Texas, 22-24 February 1999.

[Reenskaug, Wold, Lehne 1996] Trygve Reenskaug, P. Wold, O. A. Lehne ,Working With Objects: The Ooram Software Engineering Method, [ASIN: 1884777104] 1996.

[Reisig 1985] Wolfgang Reisig , "Petri Nets: an Introduction", Spring-Verlag, 1985.

[Riehle,Gross 1998] Dirk Riehle and Thomas Gross, "Role Model Based Framework Design and Integration." In Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98). ACM Press Page 117-133, 1998.

[Ripper 1999] Pedro Ripper,"VMarket: Um Framework para Sistemas de Comércio Eletrônico voltado para Mercados Virtuais Mediados por Agentes de Software" Dissertação de Mestrado,Departamento de Informática, PUC-Rio, 1999.

[Ripper,Fontoura,Maia,Lucena 2000] Ripper, P. Fontoura, M. F. Neto, A. M. Lucena, C. J. "V-Market: A Framework for e-Commerce Agent Systems" World Wide Web, Baltzer Science Publishers, 3(1), 2000.

[Schmidt 1995] D.S. Schmidt, P.Stephenson, "Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms", Proceddings of the 9th European Conference on Object-Oriented Programming, Aarbus, Denmark, 1995.

[Schneider, Lumpe 1997] Jean-Guy Schneider, Markus Lump, " Synchronizing Concurrent Objects in the Pi-Calculus. In Roland Ducournau and Serge Garlatti, editors, Proceedings of Langages et Modèles à Objets'97, pages 61-76, Roscoff, October 1997. Hermes.

[Schneider, Nierstrasz 1999] Jean-Guy Schneider, Oscar Nierstrasz, "Components, Scripts and WebCompose", Software Architectures – Advances and Applications, Leonor Barroca, Jon Hall, and Patrick Hall (Eds.), pp.13-245, Springer, 1999.

[Silva 2000] Viviane Torres da Silva, "ContentNet: Um Framework para Interoperabilidade de Conteúdos" Dissertação de Mestrado, Departamento de Informática, PUC-Rio, 1999.

[Statistics on the Web site] http://www.execpc.com/~helberg/statistics.html

[Skuikka 1998] http://www.automation.hut.fi/edu/340autumn98/kuikka/skuikka.htm

[Sparks et alli 1996] S. Sparks, K. Benner, C. Faris, Managing Object-Oriented Framework Reuse, IEEE Computer, Septmeber, pp 52-62, 1996.

[Sommerville 1995] Ian Sommerville, "Software Engineering", Addison-Wesley, 1995.

[Tong 1998] http://www.cs.ualberta.ca/~jonathan/Courses/622/Proposal/tongz/tongz.final.html

[Umar 1997] Amjad Umar, "Application (RE) Engineering Building Web-Based Applications and Dealing with Legacies" Prentice Hall PTR, 1997.

[Weinand 1994] A.Weinand,E. E.Gamma, R. Marty., "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework", Structured Programming, Vol.10, No2, July 1989.

[Wirfs-Brock, Johnson 1990] Wirfs-Brock and R. Johnson, "Surveying current research in object-oriented design", Communications of the ACM, 33(9), 1990.

[Wirfs-Brock 1999] Wirfs-Brock , "Responsibility-Driven Design: Practical Techniques for Modeling Object Behavior", OOPSLA '99 tutorial, 1999.