UNIVERSIDADE DO VALE DO RIO DOS SINOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

Rogério Samuel de Moura Martins

Composição dinâmica de Web Services

São Leopoldo
2007

Rogério Samuel de Moura Martins

Composição dinâmica de Web Services

Dissertação apresentada à Universidade do Vale do Rio dos Sinos como requisito parcial para obtenção do título de Mestre em Computação Aplicada.

Orientador Prof. Dr. Sérgio Crespo Coelho da Silva Pinto

São Leopoldo
2007

Ficha catalográfica elaborada pela Biblioteca da
Universidade do Vale do Rio dos Sinos

M386c Martins, Rogério Samuel de Moura
Composição dinâmica de Web *Services* / por Rogério Samuel de Moura Martins. -- 2007.
77p. : il. ; 30cm.

Dissertação (mestrado) -- Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, 2007.
"Orientação: Prof. Dr. Sérgio Crespo Coelho da Silva Pinto, Ciências Exatas e Tecnológicas".

1. Engenharia de *software*. 2. *Web Service*. 3. Padrões de projeto. I. Título.

CDU 004.41

Catalogação na Publicação:
Bibliotecária Eliete Mari Doncato Brasil - CRB 10/1184

“Só sei que nada sei.”
(Sócrates)

Dedicatória

à Val
por quem e com quem
eu vivo intensamente
cada segundo de minha vida

Agradecimentos

Eu devo meu maior agradecimento ao meu irmão Carlos Veríssimo por me agüentar em todos os momentos difíceis e sempre me obrigar a encarar de frente todos os problemas que apareceram pelo caminho.

Também devo muito aos meus pais que estiveram ao meu lado durante toda esta caminhada, me dando apoio para que aqui eu pudesse estar.

E para, talvez a pessoa mais importante por isto tudo ter acontecido, meu orientador e amigo Prof. Dr. Sérgio Crespo, que além de ter me dado esta oportunidade, o que certamente mudará meus caminhos, me ensinou tudo que daqui levo para minha vida.

Resumo

As invocações a serviços disponíveis na internet são construídas de forma estática, sempre referenciando o mesmo *web service* e o mesmo *web method*. Quando este serviço apresentar baixa disponibilidade o desempenho da aplicação será reduzido. Para evitar este problema é necessário que a aplicação tenha a habilidade de identificar o melhor serviço disponibilizado e então possa invocá-lo.

A inserção de novos protocolos e novas funcionalidades na arquitetura de *web services* pode permitir que as aplicações encontrem serviços disponíveis na internet e, além disso, possam medir a qualidade do serviço disponível e assim direcionar sua chamada para o melhor serviço.

Padrões de projeto são usados como um instrumento para uma melhor compreensão da arquitetura proposta.

Palavras-chave: Web Services, Composição de Serviços, Padrões de Projetos.

Abstract

The invocations to available services in the internet are built in a static way, always referring the same web service and the same web method. When this service presents low readiness the performance of the application it will be reduced. To avoid this problem it is necessary that the application has the ability to identify the best made available service and then it can invoke it.

The insert of new protocols and new functionalities in the architecture of web services can allow the applications to find available services in the internet and, besides, they can measure the quality of the available service and like this to address his call for the best service.

Design patterns are not a lust, but an instrument for a better understanding of the proposed architecture.

Keywords: Web Services, Composition of Services, Design Patterns.

Sumário

Abreviaturas

API	*Application Program Interface*
B2B	*Business to Business*
HTTP	*Hyper Text Transfer Protocol*
IMA	*Interface Matching Automatic*
MVC	*Model View Controller*
QoC	*Quality of Composition*
RPC	*Remote Procedure Call*
SOAP	*Simple Object Access Protocol*
SGML	*Standard Generalized Markup Language*
SPC	*Serviço de Proteção ao Crédito*
SRS	*Serasa*
UDDI	*Universal Distribution Discovery and Interoperability*
WS	*Web Service*
WSDL	*Web Services Description Language*
WWW	*Word Wide Web*
XML	*eXtensible Markup Language*

Lista de figuras

Lista de tabelas

Lista de gráficos

1. Introdução

A *world wide web* foi concebida originalmente como um meio de publicar texto e imagens, com o nível de interação restrito a recuperação por pedidos feitos por um usuário com uma aplicação *web browser* [McIlraith 01]. O crescimento do comércio eletrônico elevou o número de interações sem a presença de um usuário na WWW, com o *hyper text transfer protocol* sendo usado como um veículo para transações de negócios eletrônicos sem o envolvimento de um usuário [Cronin 01].

O temo *web service* significa um componente acessível de Internet que é autocontido, autodescrito, universalmente interoperável, configurável em tempo de execução e publicado e localizado por registros que são eles próprios *web services.*

A maioria das comunicações pré-*web* entre aplicações distribuídas eram síncronas. As tecnologias para prover estas comunicações distribuídas são complexas e funcionam bem apenas em um ambiente seguro como uma rede de área local, não se ajustando bem com a rota dinâmica e incerta da estrutura da Internet. Se uma aplicação estiver aberta à comunicação na Internet, esta comunicação torna-se imprevisível, assim deve haver algum sistema de prioridade quando a comunicação torna-se instável [Bosworth 01].

1.1. Motivação

Um cliente ao usar um *web service* deve escolher aquele que melhor otimiza sua execução, mas é impossível garantir que um serviço será sempre ótimo por causa da mudança constante, dinâmica e imprevisível da Internet. Para que o cliente garanta estar usando sempre o melhor serviço ele deve estar constantemente observando o estado atual da rede. Esta é uma tarefa que precisa ser executada durante todo o ciclo de vida do cliente, de maneira que o serviço seja escolhido em tempo de execução pela aplicação e não de forma estática pelo programador.

A principal motivação deste trabalho é permitir que os clientes possam manter uma visão atualizada dos serviços disponíveis e seus estados atuais para que seja possível escolher o serviço com maior disponibilidade para uma requisição. O diretório de serviços deve notificar o cliente cada vez que um serviço entra ou sai da rede e o cliente, com a lista de serviços disponíveis atualizada, deve manter estatísticas sobre os estados dos serviços. Assim é possível estabelecer uma conexão com o serviço com maior disponibilidade no momento da requisição.

1.2. O problema

A volatilidade é a principal característica da Internet [Castilho 05]. Esta volatilidade está diretamente ligada à arquitetura dos ambientes de rede utilizados na Internet que são altamente dinâmicos: não existe estabilidade na conexão entre dois nodos; e componentes

podem entrar e sair da rede a qualquer momento. Devido a esta fraqueza e dinamismo a composição de serviços ainda enfrenta sérios desafios [Foster 02].

Quando é criada uma composição de serviços, entre um cliente e um serviço, o cliente estabelece uma conexão com o serviço que otimize a execução segundo algum critério pré-definido. Por causa do dinamismo dos ambientes de rede esta escolha pode deixar de ser ótima ou se tornar inválida a qualquer instante durante a execução do serviço.

As três mudanças na rede que podem afetar a composição do serviço são [Cronin 01]:

- Ruptura da conexão: não é possível garantir que um provedor de serviços permanecerá conectado com o cliente do início ao fim da execução do serviço;
- Mudança na taxa de transferência: existem duas situações que podem fazer com que a composição deixe de ser ótima. A primeira é quando a taxa de transferência da composição cai abaixo de outra possível composição. A segunda é quando a taxa de outra possível composição sobe acima da taxa da composição atual. Isto acontece onde não existe a garantia de qualidade de serviço em pelo menos um elo da rede;
- Entrada de novos componentes: uma entrada de um novo provedor de serviço pode levar a segunda situação da taxa de transferência e invalidar a composição atual.

Um *web service* precisa monitorar estas mudanças em tempo de execução, refazendo a composição quando necessário, para garantir que ela sempre seja ótima e sempre execute da melhor forma possível.

1.3. Objetivo

O principal objetivo do trabalho é modificar a arquitetura de web services para que a aplicação possa suportar composição dinâmica com *web services*, permitindo que a composição seja feita em tempo de execução pela aplicação.

1.4. Contexto

Este trabalho está quase que completamente inserido na área de engenharia de software, buscando na disciplina de redes de computadores apenas os conteúdos necessários para a construção e a comunicação de *web services*. Para a realização do trabalho serão feitos estudos com *web services* e padrões de projetos.

A arquitetura de *web services* será modificada para suportar composição dinâmica, utilizando padrões de projeto.

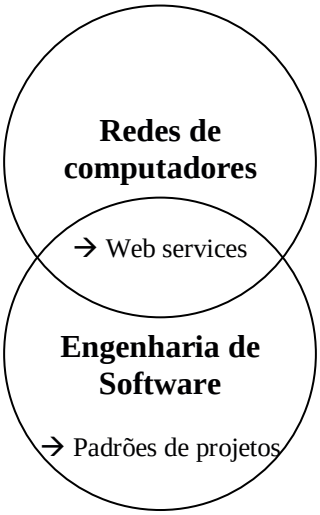

Figura 1.1 – Contexto do trabalho

1.5. Organização

Esta dissertação está dividida em oito capítulos, sendo o primeiro a introdução ao trabalho. Do segundo ao quinto capítulo é aprofundado o tema, introduzindo o estado da arte, o problema e algumas alternativas estudadas por outros autores. Os três últimos capítulos apresentam a pesquisa e os resultados obtidos:

- Capítulos 2 e 3: descrevem os conceitos e tecnologias utilizadas no desenvolvimento do trabalho. São apresentados *web services* e padrões de projetos;
- Capítulo 4: é apresentado o problema real encontrado no estado atual de *web services* como motivação para este trabalho;
- Capítulo 5: são apresentados 3 trabalhos que buscam resolver o problema da instabilidade das estruturas das redes e que são relevantes para o contexto deste trabalho;
- Capítulo 6: descreve a abordagem adotada por este trabalho detalhando a nova arquitetura proposta, seus respectivos componentes e o novo fluxo de controle por eles gerado;
- Capítulo 7: relata a implementação num ambiente real e os resultados obtidos;
- Capítulo 8: apresenta a conclusão ressaltando os pontos positivos e negativos da nova abordagem.

2. Web services

Os *web services* surgiram no intuito de substituírem as tradicionais estratégias de integração de aplicações corporativas (*Enterprise Application Integration*). Inicialmente, eram usados exclusivamente para designar a tentativa de uma empresa interligar suas aplicações internas de negócios, para que os dados fossem compartilhados. Recentemente, a sua aplicabilidade foi expandida para, também, englobar a união de dados e processos com parceiros comerciais – *Business to Business* (B2B) [Jagiello 03]. Eles apresentam uma estrutura que possibilita a comunicação entre aplicações, onde o serviço pode ser invocado remotamente, ou ser utilizado para compor um novo serviço.

Um *web service* nada mais é do que um componente de *software*, ou uma unidade lógica de aplicação, que se comunica através de tecnologias padrões de Internet [Newcomer]. Ele provê dados e serviços para outras aplicações ou serviços. Essa tecnologia combina os melhores aspectos do desenvolvimento baseado em componentes e a *web*. Como componentes, representam uma funcionalidade implementada em uma 'caixa-preta', que pode ser reutilizada sem a preocupação de como o serviço foi implementado. As aplicações acessam os *web services* através de protocolos e formatos de dados padrões, como HTTP, XML e SOAP [Dextra 03].

Diferentemente dos *web sites* tradicionais, projetados para as pessoas interagirem com informação, os *web services* conectam aplicações diretamente com outras aplicações. E a idéia básica é que essa conexão se dê sem que seja necessário efetuar grandes customizações nas próprias aplicações. Além disso, uma das premissas fundamentais é que o padrão usado pelas conexões seja aberto e independente de plataforma tecnológica ou linguagens de programação [Graham 02].

2.1. Arquitetura

A arquitetura de um *web service* é formada por três participantes: o solicitante de serviços que é o cliente, o provedor de serviços que é quem provê o *web service* e o registro de serviços que é o diretório. Os participantes são ilustrados pela figura 2.1 mostrando as colaborações entre eles.

O provedor de serviços é responsável por disponibilizar o serviço e armazenar sua descrição em WSDL contendo detalhes de interface, operação e mensagens de entrada e saída. O solicitante do serviço é uma aplicação que invoca uma interação com o serviço, podendo ser um navegador *web* ou outra aplicação qualquer como um outro *web service*. O registro de serviços é o local onde os provedores publicam seus serviços e onde os solicitantes fazem a procura [Souza 03].

Quando um serviço é disponibilizado por um provedor de serviços, este o publica sua descrição em um registro de serviços, assim um solicitante de serviços pode fazer uma busca no servidor por serviços disponíveis na Internet. Com base na análise da descrição

dos serviços retornada pelo registro de serviços, o solicitante pode decidir qual é o serviço mais adequado a ser usado. Após, o cliente inicia uma interação com o serviço fazendo uma requisição, onde o serviço retorna o resultado de sua execução em pacote SOAP. Assim outras iterações com o serviço podem ser executadas até que o requisito do cliente seja satisfeito.

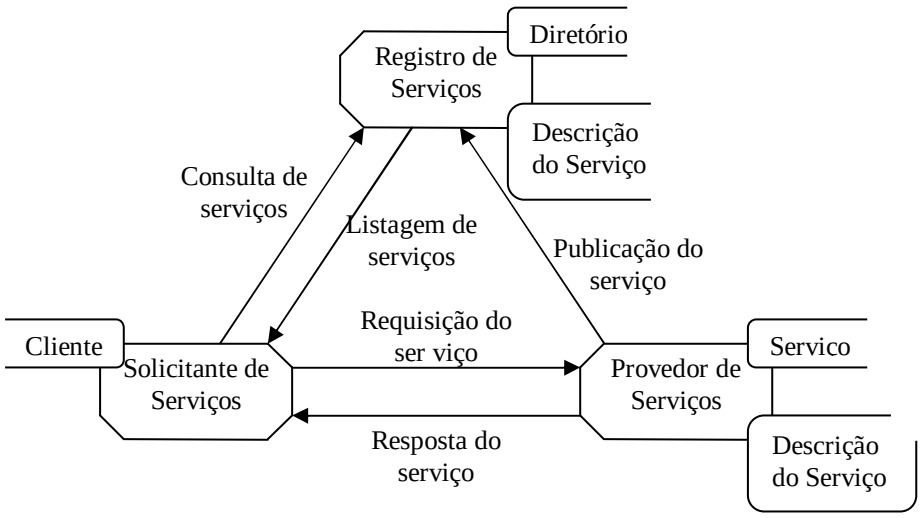

Figura 2.1 – Arquitetura de *web services*. Adaptada de [Ferris 03].

2.1.1. Camadas

Para que seja possível uma fácil integração entre os componentes da arquitetura de *web services*, a colaboração entre eles está baseada no uso de protocolos padronizados [Souza 03]. Estes protocolos são baseados em XML, destacando-se entre eles como os mais utilizados o *hypertext tranfer portocol* (HTTP), o *simple object access protocol* (SOAP), o *web service description language* (WSDL) e o *universal description, discovery and integration* (UDDI).

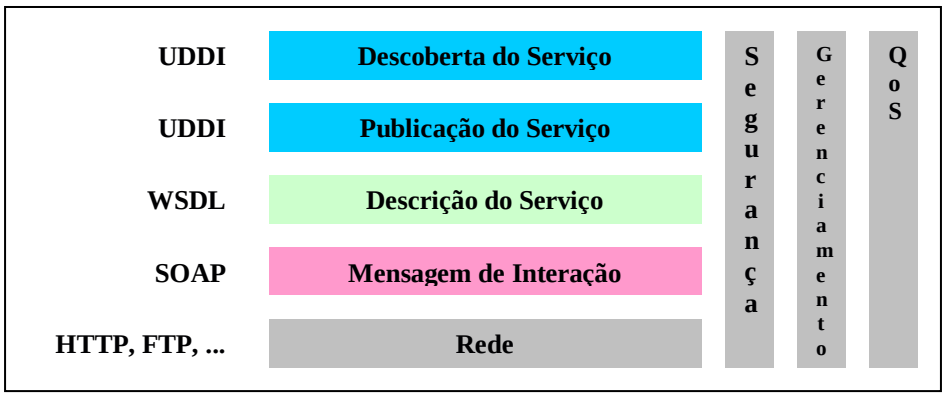

Figura 2.2 – Camadas conceituais de *web services*. Adaptada de [Rheinheimer 03].

A figura 2.2 apresenta a disposição das camadas conceituais de *web services*. A rede é a camada de mais baixo nível onde as mensagens da camada superior são transmitidas. A camada de mensagem de iteração é onde a mensagem é composta em formato XML sendo transmitida pela rede.

2.2. Tecnologia

Para a padronização dos *web services* são utilizadas diversas tecnologias que constroem os protocolos.

A linguagem XML é usada como base para o desenvolvimento de um *web service*, e provê uma linguagem para definição e processamento de dados. Os serviços são invocados e fornecem resultados através da troca de mensagens, empacotadas usando o protocolo SOAP, o qual provê um formato de serialização. A WSDL é utilizada com o objetivo de promover interoperabilidade entre sistemas heterogêneos, pois permite a construção de mensagens com a descrição precisa dos serviços. O registro UDDI é utilizado para publicação e descoberta de informações sobre *web services* [Booth 03].

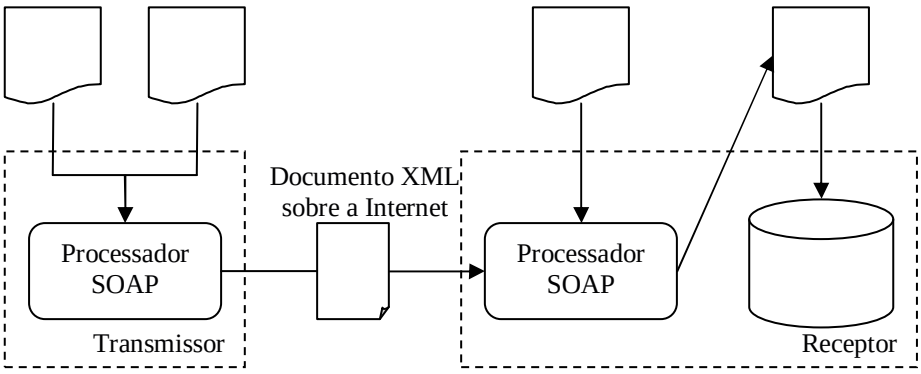

Figura 2.3 – Invocação de um *web service*. Adaptada de [Newcomer 02].

A figura 2.3 mostra o uso de algumas tecnologias por um simples *web service*. Uma vez que o WSDL é obtido do registro UDDI, uma mensagem SOAP é gerada para transmissão a um *site* remoto. Uma aplicação submete um documento (em XML) para um *web service* usando um *schema* em XML, tal como WSDL, que também será utilizado para gerar a saída para o método invocado. O computador que envia a requisição usa o SOAP para transformar os dados do formato nativo para um formato de tipo de dados predefinido em um esquema XML existente em um arquivo WSDL.

2.2.1. eXtensible Markup Language

No contexto dos *web services*, a XML, não é apenas utilizada como um formato para troca de mensagens,mas também a forma através da qual os serviços são definidos [Scopel 05].

A XML é uma linguagem de marcação que possui raízes em SGML (*standar generalized markup language*). Usando XML pode-se definir qualquer número de elementos que associam significado as informações [Santanchè 03]. Os componentes da linguagem XML são [Kratz 05]:

- Declaração: o uso da declaração da *tag* é importante, pois mostra que se trata de um documento XML e a versão em que foi escrita, podendo também declarar outros atributos que podem ser importantes na leitura do documento;
- Elementos: elementos iniciam-se com '<' e terminam com '>'. Outra característica da linguagem XML é que, assim como os sistemas da família Unix, o XML diferencia letras maiúsculas de minúsculas;
- Comentários: podem conter qualquer dado ou informação, sendo que o uso de comentários auxilia no entendimento do documento por usuários humanos, principalmente para sua transmissão. O comentário inicia com '<!--' e termina com '-->';
- Hierarquia: é baseada na semântica ou na estrutura lógica de documentos. A figura 2.4 mostra a representação gráfica e seu correspondente em XML de uma hierarquia. Toda hierarquia possui uma raiz, que é o inicio da hierarquia e geralmente uma abstração;
- Atributos: *tags* XML podem conter atributos, porém diferentes de elementos não possuem sub-atributos ou outros elementos. Na declaração de atributos devem ser dados seu nome e valor, podendo ter mais de um atributo a uma única *tag*, como pode ser visto na figura 2.5;
- Elementos vazios: na linguagem XML os elementos vazios possuem uma sintaxe modificada. Os elementos vazios são utilizados para marcar uma determinada ação , onde são representados delineados por '<' no inicio e '/>' no final, como no exemplo '<vazio/>'.

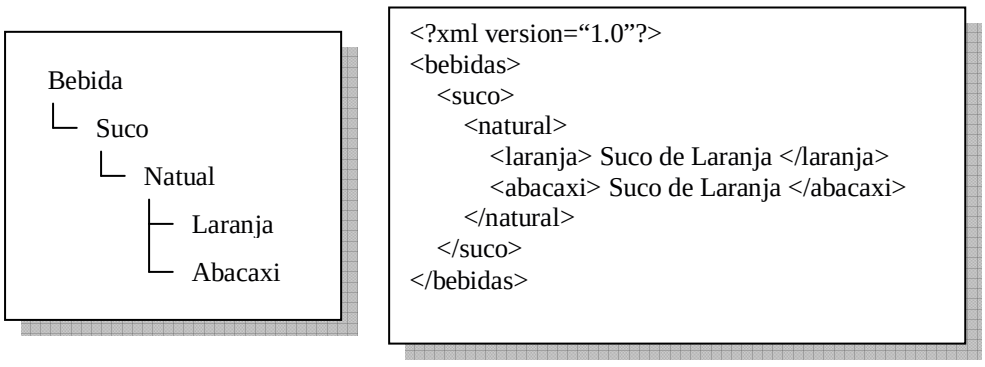

Figura 2.4 – Exemplo de uma hierarquia em XML.

A figura 2.5 mostra um exemplo de um arquivo XML. Nele as *tags* dão significado às informações e também constroem hierarquias, já os atributos dão características para as informações. Lendo o arquivo podemos compreender que a informação "Introdução ao

XML” é o nome de um artigo escrito em xml na língua portuguesa brasileira e que a informação “XML Introduction” é o nome do mesmo artigo só que para a língua inglesa.

```
<?xml version=“1.0”?>

<tagxml-1 valor=“1”> Exemplo de tag </tagxml-1>
<tagxml-2 valor1=“exemplo” valor2=“básico”> Exemplo de tag com múltiplos atributos</tagxml-2>

<artigos>
   <xml>
      <nome idioma=“pt-br”> Introdução ao XML </nome>
      <nome idioma=“eng”> XML Introduction </nome>
      <nome idioma=“es”> Introduccion de XML </nome>
      <nome idioma=“gm”> XML Einfuhtung </nome>
      <nome idioma=“it”> Introduzione di XML </nome>
   </xml>
</artigos>
```

Figura 2.5 – Estrutura básica de um documento XML. Adaptada de [Kratz 05].

Duas partes que troquem dados em XML poderão entender e interpretar os elementos da mesma forma somente se elas compartilham da mesma definição, só assim elas poderão entender o significado dos elementos entre as *tags*, que é a forma exata como os *web services* trabalham [Newcomer 02].

A sintaxe de XML usada em *web services* especifica como os dados são genericamente representados, define como e com que qualidade de serviços os dados são transmitidos, e detalhes de como os serviços são publicados e descobertos. Implementações de *web services* decodificam os conteúdos das mensagens em XML para interagir com várias aplicações e domínios de softwares que usam estes serviços [Coyle 02].

2.2.2. Web Sevice Description Language

A WSDL é uma linguagem padrão em XML com o objetivo de descrever os formatos e protocolos de um *web service* de forma simples. Ela é usada basicamente por registros de serviços para publicação de *web services*. Seus elementos contêm a descrição dos dados, a descrição das operações que podem ser realizadas com estes dados e informações sobre o protocolo de transporte que será utilizado [Newcomer 02]. Para cada *web service* há um arquivo WSDL descrevendo as operações que ele realiza.

A WSDL é divida em três elementos principais [Souza 03]:

- Definições de tipos de dados: determina a estrutura e o conteúdo das mensagens;
- Operações abstratas: determinam as operações possíveis;
- Protocolo de ligação: determina a forma de transmissão das mensagens pela rede até os destinatários.

O uso da WSDL permite a divisão da especificação de um *web service* em partes, propiciando o reuso de forma separada em diversos *web services* (figura 2.6).

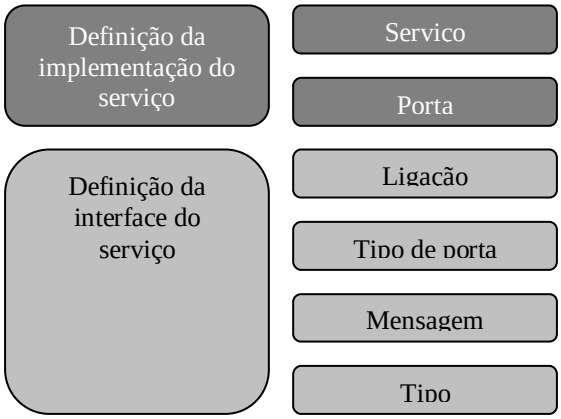

Figura 2.6 – Camada de descrição de serviços. Adaptada de [Souza 03].

A implementação do serviço descreve onde o *web service* será instalado e como este é acessado. A interface do serviço contém a definição do serviço. A ligação descreve o protocolo, o formato dos dados e outros atributos para a interface de um serviço particular. Os elementos das operações do *web sevice* são definidos no tipo de porta. A mensagem define os parâmetros de entrada e saída de uma operação. O tipo define o uso de tipos de dados complexos dentro de uma mensagem.

```
<?xml version="1.0"> encoding="UTF-8"?>
<definitions name="HelloService"
   targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap"
   xmlns:tns="http://schemas.xmlsoap.org/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
>

   <message name="SayHelloRequest">
      <part name="firstName" type="xsd:string"/>
   </message>

   <message name="SayHelloResponse">
      <part name="greeting" type="xsd:string"/>
   </message>

   <portType name="Hello_PortType">
      <operation name="sayHello">
         <input message="tns:SayHelloRequest"/>
         <output message="tns:SayHelloResponse"/>
      </operation>
   </portType>
```

```
<binding name="Hello_Binding" type="tns:Hello_PortType">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http">
   <operation name="sayHello">
      <soap:operation soapAction="sayHello"/>
      <input>
         <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:examples:helloservice"
            use="encoded"
         />
      <output>
         <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:examples:helloservice"
            use="encoded"
         />
      </output>
   </operation>
</binding>

<service name="Hello Service">
   <documentation>WSDL File for HelloService</documentation>
   <port binding="tns:Hello_Binding" name="Hello_port">
      <soap:address location="http://localhost:8080/soap/servlet/rpcrouter"/>
   </port>
</service>

</definitions>
```

Figura 2.7 – Documento WSDL HelloService.wsdl [Cerami 02].

O código da figura 2.7 descreve um exemplo de um documento WSDL de um serviço com apenas uma função pública, a função sayHello. Essa função é o conhecido programa "Alô mundo" que ao receber um nome como parâmetro retorna uma mensagem como resposta. Por exemplo, se for passado o parâmetro "Rogério", o serviço retorna a mensagem "Hello, Rogério!".

2.2.3. Simple Object Access Protocol

O SOAP consiste de um protocolo que é utilizado para a troca de informações em um ambiente descentralizado e distribuído, permitindo que isso seja feito entre diversas aplicações independente de sistema operacional, linguagem de programação ou plataforma [Newcomer 02]. De uma maneira geral, define o formato que as mensagens transportadas na rede devem ter para encaminhar requisições aos *web services*.

A comunicação é feita através de troca de mensagens, transmitidas em formato XML, incluindo parâmetros usados na chamada, bem como dados de resultados. Isto significa que as mensagens podem ser entendidas por quase todas as plataformas de *hardware*, sistemas operacionais, linguagens de programação ou mesmo *hardware* de rede. Também

pode ser utilizado para invocar, publicar e localizar *web services* no registro UDDI [Hansen 03].

De acordo com [Seely 02] um pacote SOAP é composto pelas seguintes partes:

- Envelope: responsável por definir o início e o fim das mensagens, quem pode processá-la, determinando se o tratamento é obrigatório ou opcional;
- Cabeçalho: local que possui os atributos opcionais das mensagens;
- Corpo: conteúdo da mensagem em XML;
- Anexo: consiste em um ou mais documentos anexados à mensagem principal;
- Codificação: define os mecanismos de serialização que podem ser usados para troca de instâncias ou tipos de dados pelas aplicações;
- *Remote procedure call*: define como o modelo RPC interage com o SOAP com o objetivo de invocar procedimentos em um sistema remoto.

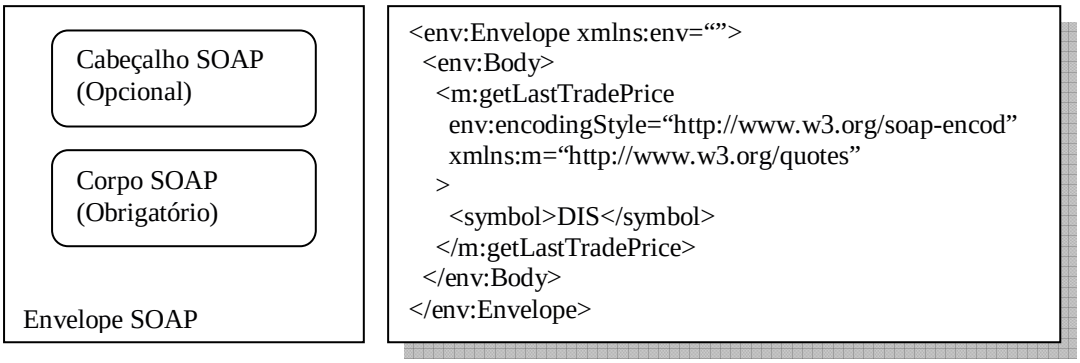

Figura 2.8 – Partes de uma mensagem SOAP [Scopel 05].

O SOAP não define o serviço propriamente, mas apenas o suficiente para que o processador SOAP possa reconhecê-lo. A invocação do serviço usando o SOAP é demonstrada na figura 2.9.

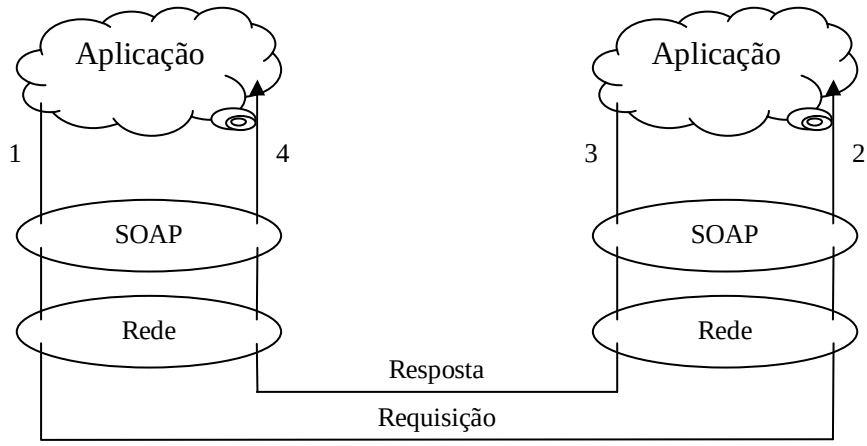

Figura 2.9 – Invocação de um serviço através do protocolo SOAP [Scopel 05].

A aplicação (1) requisita uma mensagem SOAP e invoca a operação do serviço através de um provedor de *web service*. O solicitante do serviço apresenta a mensagem com o endereço de rede do provedor *web service*. A infra-estrutura de rede(2) entrega a mensagem para um servidor SOAP. O servidor SOAP redireciona a mensagem para o *web service*. O *web service* (3) é responsável por processar a mensagem de requisição e formular a resposta. Quando a mensagem chega no requisitante (4), é convertida para uma linguagem de programação, sendo então entregue para a aplicação.

2.2.4. Universal Description, Discovery and Integration

O UDDI consiste em uma especificação técnica para descrever e integrar *web services*. Ele é composto por duas partes: uma especificação técnica para construir e distribuir *web services*, a qual permite que as informações sejam armazenadas em um formato XML específico e o UDDI *Business Registry*, que é uma implementação operacional completa da especificação UDDI [Newcomer 02].

Os dados UDDI podem ser divididos em três categorias principais [Hansen 03]:

- Páginas brancas: inclui informações gerais sobre uma empresa especifica tais como nome, descrição contato, endereço e número de telefone;
- Páginas amarelas: são incluídos dados de classificação gerais da empresa ou serviço oferecido;
- Páginas verdes: contém informações técnicas sobre *web services*. Geralmente possui um ponteiro para uma especificação externa e um endereço para invocar o *web service*.

A informação presente no WSDL completa a informação presente no UDDI [Scopel 05]. Desta forma o processo de registro das informações ocorre como demonstrado na figura 2.10.

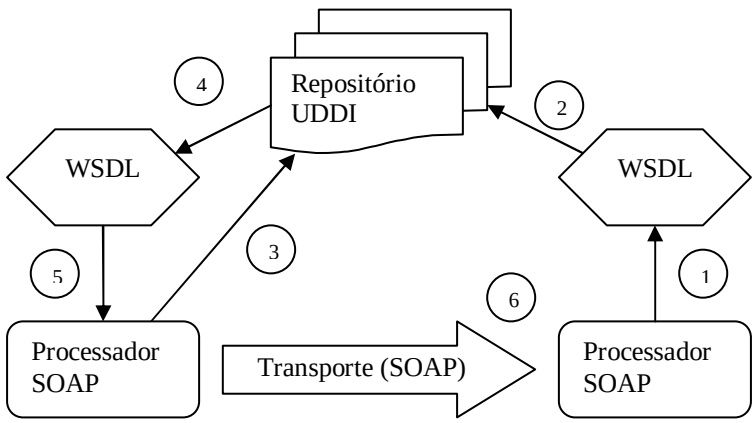

Figura 2.10 – UDDI utilizado para descobrir um *web service* [Newcomer 02].

Primeiramente, gera-se o arquivo WSDL para descrever o *web service* com suporte do Processador SOAP (1) e utiliza-se a *application program interface* (API) UDDI para registrar as informações no repositório (2). Os dados são transmitidos juntamente com as informações sobre contato e o registro possui uma entrada (URL que aponta para o Servidor SOAP) com a localização do WSDL, assim outro Processador SOAP pode requisitar o registro (3) para obter o WSDL (4). Após, o cliente gera a mensagem apropriada (5) para enviar uma operação específica através de determinado protocolo (6). O cliente e o servidor devem estabelecer o mesmo protocolo e compartilhar a mesma semântica para a definição do serviço, a qual neste exemplo o protocolo é SOAP sobre HTTP e a semântica é definida através da WSDL.

3. Padrões de Projetos

Um padrão é uma maneira de fazer algo, ou de buscar um objetivo. Em qualquer atividade que esteja madura ou em vias de amadurecer, encontraremos métodos eficazes comuns para atingir objetivos e para resolver problemas em vários contextos. A comunidade de pessoas que praticam um ofício geralmente cria um jargão que os ajuda a falar a respeito dele. Tal jargão freqüentemente se refere a padrões, ou maneiras padronizadas de atingir certos objetivos. Os escritores documentam esses padrões, ajudando a padronizar o jargão [Metsker 04].

Padrões de projetos de *software* são soluções genéricas para problemas recorrentes em engenharia de *software* [Rheinheimer 03]. Chistopher Alexander afirma: "cada padrão descreve um problema no nosso ambiente e o núcleo da solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira" [Alexander 77].

Um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto. Ele identifica as classes e instancias participantes, seus papéis, colaborações e a distribuição de responsabilidades [Gamma 00]. O uso dos padrões de projeto consiste em identificar os padrões que se propõem a resolver o problema do projeto em questão e a sua customização no contexto do projeto.

As razões mais comumente estabelecidas para utilizar padrões se devem ao fato que eles permitem [Shalloway 04]:

- Reutilizar soluções: utilizando soluções já testadas para problemas recorrentes garantimos maior qualidade;
- Estabelecer terminologia comum: fornecem um ponto comum de referência durante a fase de análise e elaboração do projeto;
- Visão de alto nível: os padrões de projeto fornecem uma perspectiva de mais alto nível acerca dos problemas e do processo de projeto, postergando os detalhes de implementação.

Em geral os padrões de projetos possuem quatro elementos essenciais [Gamma 00]:

- O nome do padrão: é uma referência usada para descrever o problema encontrado e a solução adotada;
- O problema: descreve a situação onde o padrão pode ser utilizado;
- A solução: descreve os elementos que compõem o projeto, seus relacionamentos, suas responsabilidades e colaborações. É como um gabarito que pode ser aplicado em muitas situações diferentes, não descrevendo um projeto concreto;
- As conseqüências: são os resultados e análises das vantagens e desvantagens da aplicação do padrão para que seja possível uma avaliação de alternativas de projetos e para a compreensão dos custos e benefícios da aplicação do padrão.

Os padrões de projeto são classificados segundo dois critérios: escopo e propósito. Esta classificação é feita devida à variação na granularidade e na abstração de um padrão para outro. Ela ajuda a aprender os padrões mais rapidamente, bem como direcionar esforços na descoberta de novos. O primeiro critério diz respeito à finalidade, refletindo o que o padrão faz, podendo ser de criação, estrutura ou comportamental. O segundo, refere-se ao escopo, e especifica se o padrão é estático ou dinâmico. Desta forma os padrões podem ser classificados conforme a tabela 3.1.

		Propósito		
		Criação	**Estrutura**	**Comportamento**
Escopo	**Classe**	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Tabela 3.1 – Classificação dos padrões de projeto segundo [Gamma 00].

Os padrões para classe lidam com os relacionamentos entre classes e suas subclasses. Estes mecanismos são estabelecidos através de herança. Os padrões para objetos lidam com relacionamentos entre objetos e podem ser mudados em tempo de execução.

Padrões de criação abstraem o processo de instanciação. Ajudam a tornar o sistema independente de como os objetos são criados, compostos e representados. Um padrão de criação de classes usa a herança para variar a classe que é instanciada, enquanto um padrão de criação de objetos delega a instanciação para outro objeto. Os padrões de criação dão muita flexibilidade quanto ao que é criado, quem cria, como e quando é criado. Permitem configurar um sistema com objetos "produto" que variam amplamente em estrutura e funcionalidade, sendo que esta configuração pode ser estática ou dinâmica [Gamma 00].

Padrões estruturais preocupam-se com a forma como classes e objetos são compostos para formar estruturas maiores. Os padrões estruturais de classes utilizam a herança para compor interfaces ou implementações. Este tipo de padrão é particularmente útil para fazer bibliotecas de classes desenvolvidas independentemente trabalharem juntas. Já os padrões estruturais de objetos, em lugar de compor interfaces ou implementações, descrevem maneiras de compor objetos para obter novas funcionalidades. A flexibilidade provém da capacidade de mudar a composição em tempo de execução, o que é impossível com a composição estática de classes [Gamma 00].

Padrões comportamentais preocupam-se com algoritmos e a atribuição de responsabilidades entre objetos. Não descrevem apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles. Estes padrões caracterizam fluxos de controle difíceis de seguir em tempo de execução; eles afastam o foco do fluxo de controle, para que seja possível concentrar-se somente na maneira como os objetos são interconectados. Padrões comportamentais de classes utilizam herança para distribuir o comportamento entre classes, enquanto padrões comportamentais de objetos utilizam a composição de objetos. Alguns descrevem como um grupo de objetos pares (peer objects) cooperam para a execução de uma tarefa que nenhum objeto sozinho poderia executar por si mesmo [Gamma 00].

Serão abordados neste capítulo os padrões *model view controller*, *observer* e *strategy* por constituírem a solução do problema deste trabalho.

3.1. Model View Controller

O padrão de projeto MVC é composto por três tipos de objetos. O modelo (*model*) é o objeto de aplicação, a vista (*view*) é a apresentação na tela e o controlador (*controller*) define a maneira como a interface do programa reage às entradas do usuário. Antes do MVC, os projetos de interface para o usuário tendiam a agrupar esses objetos. MVC separa esses objetos para aumentar a flexibilidade e a reutilização.

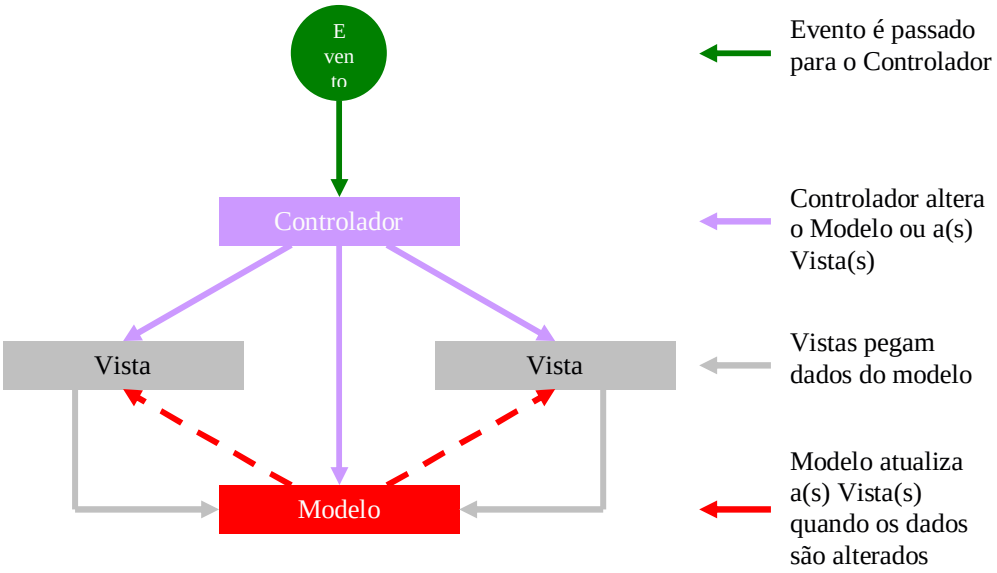

Figura 3.1 – Fluxo de controle do MVC.

Embora diferentes implementações trabalhem de diferentes maneiras, a figura 3.1 mostra o fluxo de controle do MVC trabalhando da seguinte forma [Buschmann 96]:

- Um evento é gerado a partir da interface do usuário de várias formas;
- A interface do usuário passa o evento para o controlador;

- O controlador acessa o modelo, possibilitando a atualização de uma forma apropriada;
- O modelo notifica as vistas que foi alterado e que o estado delas é inconsistente;
- A vista acessa o modelo para pegar os dados e gerar uma exibição coerente com o modelo;
- A interface do usuário espera por novas interações do usuário que causem eventos, recomeçando o ciclo.

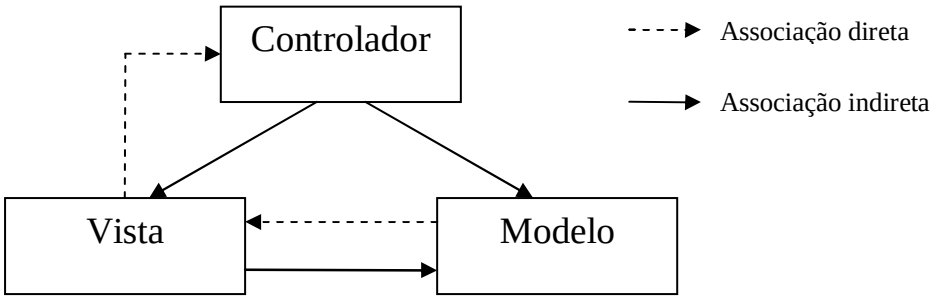

Figura 3.2 – Estrutura de relação entre os componentes do MVC.

A figura 3.2 mostra a estrutura de relação entre os componentes do MVC. As associações indiretas têm por objetivo diminuir o acoplamento entre os componentes. O modelo só conhece a interface da vista e a vista só conhece a interface do controlador, assim é criado um acoplamento abstrato entre os componentes.

Model View Controller **composto**

Intenção

Quebrar uma aplicação, ou apenas parte dela, em partes, com uma clara separação dos objetos de cada parte.

Problema

Aplicações que precisam manter múltiplas views do mesmo dado.

Solução

Encapsular os dados junto com o seu processamento (modelo), isolando de sua manipulação (controlador) e apresentação (vista).

Conseqüências

O desacoplamento modifica desde como os dados são manipulados até como são apresentados ou armazenados, enquanto unifica o código em cada componente. Por causa desta separação, múltiplas vistas e controladores podem interagir com o mesmo modelo.

Implementação

O MVC é um padrão composto de vários outros padrões:

- As vistas formam uma árvore usando *composite*;
- A relação entre vistas e modelo é feita pelo *observer*;
- Os controladores são estratégias das vistas (*strategy*);
- Controladores complexos são freqüentemente estruturados usando o padão *command*;
- O padrão *factory method* pode ser usado para especificar por falta a classe controladora para uma vista;
- O padrão *decorator* pode ser usado para acrescentar capacidade de rolagem a uma vista.

Outros padrões podem ser usados, mas os principais relacionamentos são fornecidos pelos padrões *observer*, *composite* e *strategy*.

Tabela 3.2 – Principais características do padrão MVC.

De acordo com a tabela 3.2 o MVC nada mais é do que a composição de padrões de projetos para garantir que seu relacionamento (figura 3.2) seja satisfeito.

Um dos objetivos do MVC é separar objetos de maneira que mudanças ocorridas em um possam afetar um número qualquer de outros objetos sem exigir que o objeto mudado conheça detalhes dos outros. Este projeto mais geral é descrito pelo padrão *observer*. Ele separa vistas e modelos pelo estabelecimento de um protocolo inserção/notificação (*subscribe*/*notify*) entre eles. Uma vista deve garantir que sua aparência reflita o estado do modelo. Sempre que os dados mudam, o modelo notifica as vistas que dependem dele. Em resposta, cada vista tem a oportunidade de atualizar-se. Essa abordagem permite ligar múltiplas vistas a um modelo para fornecer diferentes apresentações.

O MVC também permite mudar a maneira como uma vista responde às entradas do usuário sem mudar sua apresentação visual, encapsulando o mecanismo de resposta em um objeto controlador. Uma vista usa uma instância de uma interface de um controlador para implementar uma estratégia particular de respostas; para usar uma estratégia diferente basta simplesmente substituir a instância por um tipo diferente de controlador. O relacionamento vista-controlador é obtido através do uso do padrão *strategy*. Este padrão é útil quando se deseja definir uma família de algoritmos, encapsular cada um deles e fazê-los intercambiáveis. Ele permite que os clientes variem independentemente dos clientes. Outra característica do MVC é que as vistas podem ser encaixadas. Através do padrão *composite* é possível agrupar objetos e tratar como um objeto individual, já que ele permite criar uma hierarquia de classes na qual algumas subclasses definem objetos primitivos e outras classes definem objetos compostos.

3.2. O padrão Observer

O padrão *observer* permite transferir a responsabilidade do observador de monitorar a mudança no sujeito, para o próprio sujeito. Quando a mudança ocorre, o sujeito notifica o observador que se atualiza automaticamente. Assim o observador não precisa prever quando ocorrerá uma mudança no sujeito e nem ficar temporariamente fazendo uma

atualização, ele apenas precisa esperar uma notificação. Outra vantagem é que o sujeito não precisa conhecer quem é o observador, apenas saber que ele espera uma notificação.

Observer **comportamental de objetos**

Intenção

Definir uma dependência um-para-muitos entre objetos, de maneira que quando um objeto muda seu estado todos os seus dependentes são notifiados e atualizados automaticamente.

Também conhecido como

Dependents, Publish-Subscribe

Motivação

Um efeito colateral comum resultante do particionamento de um sistema em uma coleção de classes cooperantes é a necessidade de manter a consistência entre objetos relacionados.

Aplicabilidade

O padrão observer deve ser usado quando:

- uma abstração tem dois aspectos, e um deles é dependente do outro;
- uma mudança em um objeto exige mudanças em outros, e você não sabe quantos objetos precisam ser mudados;
- um objeto deveria ser capaz de notificar outros objetos sem fazer hipóteses, ou usar informações, de quem são estes objetos. Em outras palavras, não se deseja que estes objetos sejam fortemente acoplados.

Estrutura

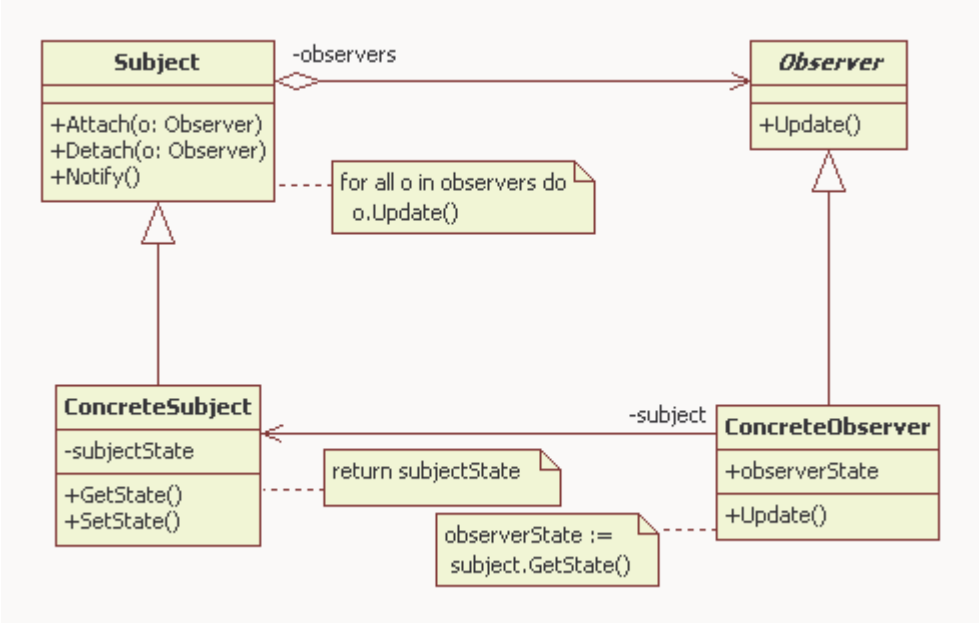

Figura 3.3 – Diagrama de classe UML do padrão *observer* [Shalloway 00].

Participantes

- *Subject*: conhece os seus observadores; fornece uma interface para acrescentar e remover objetos para associar e desassociar objetos *observer*;
- *Observer*: define uma interface de atualização para objetos que deveriam ser notificados sobre mudanças em um *subject*;
- *ConcreteSubject*: armazena estados de interesse para objetos *ConcreteObserver*; envia uma notificação para seus observadores quando seu estado muda;
- *ConcreteObserver*: mantém uma referencia para um objeto *ConcreteSubject*; armazena estados que deveriam permanecer consistentes com os do *Subject*; implementa a interface de atualização de *Observer*, para manter seu estado consistente com o do *Subject*.

Colaborações

- O *ConcreteSubject* notifica seus observadores sempre que ocorre uma mudança que poderia tornar inconsistente o estado deles com o seu próprio;
- Após ter sido informado de uma mudança no *Subject* concreto, um objeto *ConcreteObeserver* pode consultar o *Subject* para obter informações.

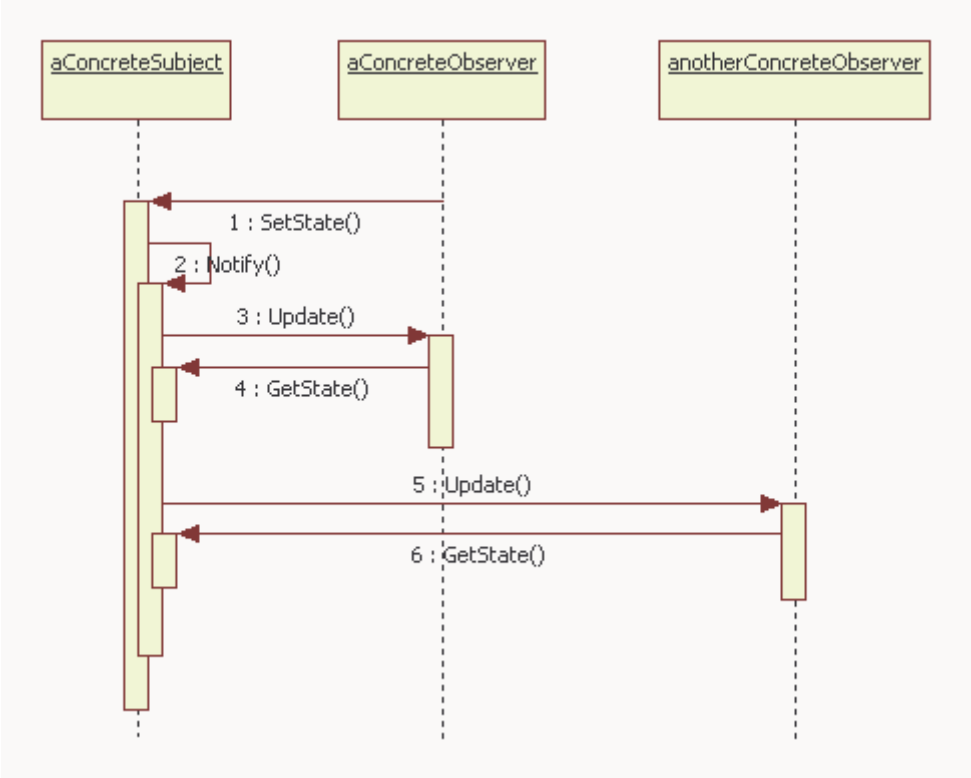

Figura 3.4 – Diagrama de seqüência mostrando a colaboração do padrão *observer*.

Conseqüências

O padrão *observer* permite variar *subjects* e observadores de forma independente. Pode-se reutilizar *subjects* sem reutilizar seus observadores e vice-versa. Permite

acrescentar observadores sem modificar o *Subject* ou outros observadores. Benefícios adicionais e deficiências do padrão incluem o seguinte: acoplamento abstrato entre *Subject* e *Observer*; suporte para comunicações broadcast; atualizações inesperadas.

Tabela 3.3 – Principais características do padrão *observer*.

Em projetos que não usam o padrão *observer* normalmente o sujeito está fortemente acoplado ao observador [Metsker 04]. Nesses projetos o sujeito mantém uma referência concreta do observador e chama diretamente o método de notificação. Toda vez que um observador é adicionado ao sujeito, o sujeito precisa ser modificado para suportar o novo observador. Com o padrão *observer*, nenhuma mudança no sujeito é preciso, pois ele mantém um acoplamento abstrato com o observador (figura 3.3). O observador só precisa se subscrever no sujeito e esperar ama notificação para se atualizar, como relatado pela tabela 3.3.

Em outros tipos de projetos sem o *observer* a complexidade aumenta ainda mais, pois ou o observador prevê quando ocorrerão mudanças que tornem o sujeito inconsistente com ele, ou ele fica de tempos em tempos monitorando o estado do sujeito. Dificilmente essa abordagem é empregada.

3.3. O padrão Strategy

O padrão *strategy* permite escolher a melhor estratégia de acordo com o contexto. Quando uma ação precisa ser executada para atingir um objetivo, a aplicação pode escolher a melhor dentre várias ações disponíveis analisando o estado atual do sistema, isso permite otimizar a execução segundo algum critério pré-estabelecido.

Strategy	**comportamental de objetos**

Intenção

Definir uma família de algoritmos, encapsular cada uma delas e torna-las intercambiáveis. *Strategy* permite que o algoritmo varie independentemente dos clientes que o utilizam.

Também conhecido como

Policy

Motivação

Existem muitos algoritmos para a realização de uma mesma tarefa. Codificar de maneira fixa e rígida tais algoritmos nas classes que os necessitam não é desejável, por várias razões: clientes que necessitem suportar diferentes algoritmos se tornam maiores e difíceis de manter; diferentes algoritmos serão apropriados em diferentes situações; é difícil adicionar novos algoritmos e variar os existentes quando eles são partes integrantes de um cliente. Podemos evitar estes problemas definindo classes

que encapsulam diferentes algoritmos e instanciando o mais adequado ao contexto do cliente.

Aplicabilidade

O padrão *strategy* deve ser usado quando:

- muitas classes relacionadas diferem somente no seu comportamento. As estratégias fornecem uma maneira de configurar uma classe com um, dentre muitos comportamentos;
- você necessita de variantes de um algoritmo;
- um algoritmo usa dados que os cliente não deveriam ter conhecimento;
- uma classe define muitos comportamentos, e estes aparecem em suas operações como múltiplos comandos condicionais da linguagem.

Estrutura

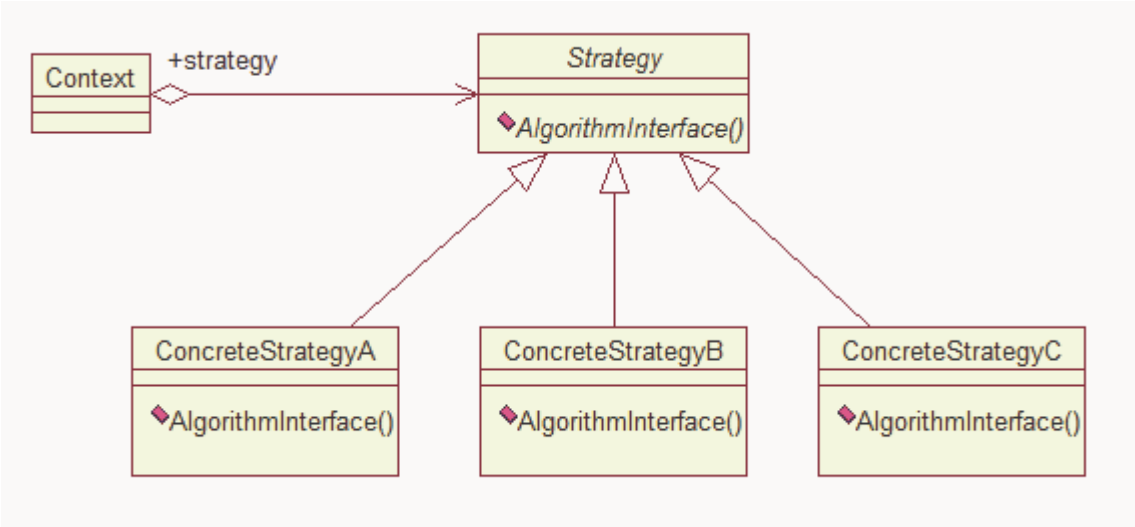

Figura 3.5 – Diagrama de classe UML do padrão *strategy* [Shalloway 00].

Participantes

- *Strategy*: define uma interface comum para todos os algoritmos suportados;
- *ConcreteStrategy*: implementa o algoritmo usando a interface de S*trategy*;
- *Context*: mantém uma referência para um objeto S*trategy* e é configurado com um objeto *ConcreteStrategy*.

Colaborações

- *Strategy* e *Context* interagem para implementar o algoritmo escolhido. Um contexto pode passar todos os dados requeridos pelo algoritmo para a estratégia quando o algoritmo é chamado;
- Um contexto repassa solicitações dos seus clientes para sua estratégia. Os clientes usualmente criam e passam um objeto *ConcreteStrategy* para o contexto; após isso, interagem exclusivamente com o contexto. Freqüentemente existe uma família de classes *ConcreteStrategy* para um cliente fazer sua escolha.

Conseqüências

O padrão *strategy* define uma família de algoritmos. Os comandos *switch* e condicionais podem ser eliminados. Deve-se invocar os algoritmos da mesma maneira (eles todos devem ter a mesma interface).

Tabela 3.4 – Principais características do padrão *strategy*.

O uso do padrão *strategy* permite modificar o conjunto de estratégias sem que sejam necessárias modificações diretas no contexto de execução da estratégia. O desuso do padrão implica em modificações diretas no código aonde a chamada da ação é realizada, o que torna o programa rígido e a inserção de novas estratégias podem ocorrer apenas em tempo de compilação.

4. Trabalhos relacionados

Para a realização desta dissertação foram destacados três trabalhos que terão maior influência sobre a forma como a solução será construída. O primeiro trabalho é de [Gang 04] que implementa um modelo de conector adaptável, o segundo é o trabalho de [Hausmann 03] que utiliza ontologias para seleção automática de serviços. Nesta mesma linha aparece o trabalho de [Zhang 03] que constrói composições automáticas de serviços utilizando ontologias.

4.1. Modelo de conector de serviço adaptável

Os autores deste trabalho [Gang 04] apresentam um modelo de conector de serviço adaptável como solução para minimizar os problemas da volatilidade dos ambientes de rede. Conexões de serviço são tratadas como componentes individuais chamados conectores de serviço, assim é criado um modelo de conector de serviço adaptável que adota um mecanismo baseado em papel para ajustar as conexões entre serviços. Um papel é uma abstração de serviços com funcionalidades em comum. Esta abstração oferece uma estrutura de conector mutável, habilita reconfiguração da interação dos serviços e encapsula mudanças nos participantes da interação, fazendo as conexões de serviço mais adaptáveis.

Para fazer a conexão adaptável, a estrutura de conexão precisa ser mutável. Como um conceito semântico, um papel prove um mecanismo de organização pelo qual a abstração de serviços com funções comuns são derivadas e marcadas pelas características do papel. Com este mecanismo, um papel oferece estrutura de conexão flexível, habilitando adaptação de conexão de serviço por reconfiguração.

Baseado no raciocínio acima, o autor Gang Li implementa uma conexão de serviço como um componente explícito chamado conector de serviço e criaram um modelo de conector de serviço baseado em papel. Neste modelo, um papel é usado para cumprir a adaptação de uma conexão de serviço com uma interface de interação estável e uma estrutura de conexão mutável [Gang 04].

4.1.1. Modelo

No modelo de conector de serviço baseado em papel, o cliente não interage com o serviço e sim com um papel, como mostra a figura 4.1. Este papel é responsável por estabelecer a conexão com o serviço. Descrito pelas características, o papel é uma abstração dos serviços. As características do papel apresentam as funções que este papel prove, estas funções são implementadas pelos serviços que são invisíveis ao cliente. Quando uma mudança inesperada causa uma modificação na interação do papel com o serviço, o conector pode se adaptar às mudanças de interação de serviço ou de requerimentos reconfigurando as características do papel ou os provedores de serviço. Quando um

serviço envolvido em uma interação está indisponível, outro com a mesma característica pode substituí-lo, isto aumenta a adaptabilidade e a confiabilidade da conexão de serviço. Um papel é um serviço virtual, e não só oferece uma estrutura de conector adaptável, mas também uma interface de interação de serviço unificada, provendo suporte para a adaptação de conexão em composições dinâmicas de serviço.

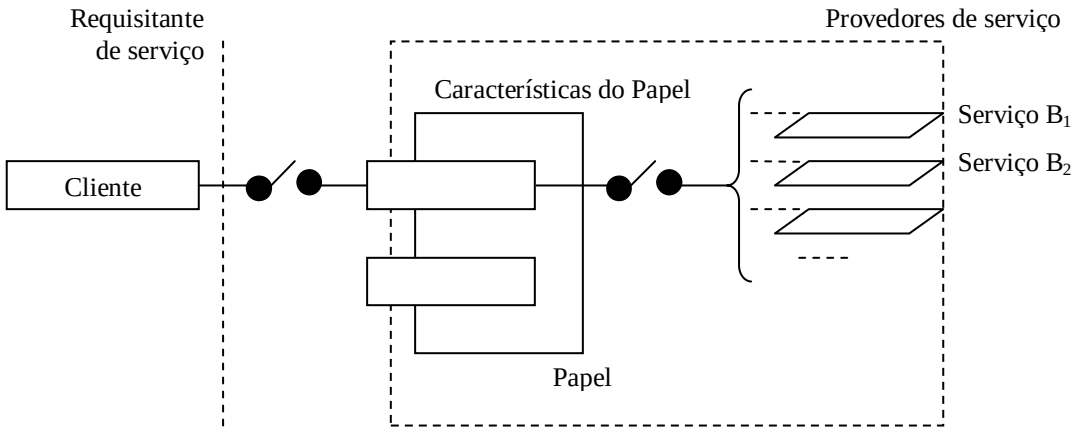

Figura 4.1 – Modelo do conector de serviço baseado em papel.

Formalmente um papel é uma tripla $<Name_r, Features_r, Service_r>$, onde $Name_r$ é o nome do papel; $Features_r$ é um conjunto de características do papel, $Ferатures_r = \{f_r | f_r = <rn, fn_r, va_r>\}$, onde *rn* é o nome do papel que a característica f_r pertence, fn_r é o nome da característica f_r e va_r é o vetor de argumentos de f_r. Características do papel é a interface com a qual um papel interage com os clientes e serviços; $Service_r$ é um conjunto de serviços que estão relacionados as características.

Um tripa define um conector de serviço baseado em papel, se e somente se ela tem as seguintes propriedades:

- Há um conjunto de função, denominado *Map*. Dado $\forall f_r \in Features_r$ então $\exists m \in Map$, $S_{sr} \subseteq Service_r$, $m(f_r) = S_{sr}$, e serviços que pertencem ao conjunto S_{sr} tem a mesma interface descrita por f_r;
- Há um conjunto de função, denominado *Selectors*, considerando f_r, $\exists sel \in Selectors$, $Ser \in S_{sr}$, $sel(S_{sr}) = Ser$.

A função *m* é chamada de função de mapeamento de característica e a função *sel* é chamada de função de seleção de serviço.

Assim um conector apresenta um serviço configurável chamado ao invés do serviço requerido. O conector seleciona um provedor de serviços apropriado e mapeia os parâmetros para o pedido. Ele é um encapsulamento configurável de um serviço ou um grupo de serviços semanticamente similares, abstraídos como um papel. Existem dois padrões de interação envolvidos em um modelo de conector de serviço baseado em papel:

serviço-papel e papel-papel. No primeiro padrão, os serviços são empacotados pelo papel, como mostra a figura 4.1. O padrão ajusta o contexto onde os serviços são voláteis. No segundo padrão, os clientes e os serviços são empacotados pelos papéis, o qual ajusta o contexto onde ambas as partes são voláteis.

Os protocolos do modelo descrevem como configurar o conector automaticamente. Além disso, eles enfatizam especialmente os estados de conexão e da interação dos participantes. Por um lado, um papel suporta a seleção dinâmica de serviços de acordo com o estado da conexão e do serviço; por outro lado quando a conexão precisa ser reconfigurada, estes estados vão determinar se a reconfiguração é possível. Durante a reconfiguração os estados dos participantes da interação são salvos, após eles são restaurados para permitir que a composição do serviço seja refeita.

4.1.2. Análise

A tabela 4.1, extraída de [Gang 04], lista a comparação dos tipos de adaptações de conexão de serviço. Ela compara os conectores de serviço baseados em papel com as conexões de serviços baseadas em fluxo de controle [Casati 00] e adaptadores de serviço [Andrews 03] [Krishnan 02].

Característica	**Conexão de serviço baseada em fluxo de controle**	**Adaptadores de serviço**	**Conectores de serviço baseados em papel**
Executor da adaptação	Programadores	Programadores	Usuários
Modo de adaptação	Por modificação do código fonte	Por modificação do código fonte ou customização	Por reconfiguração
Grau de automação	Não automático	Não automático	Semi-automático
Momento de adaptação	Em tempo de construção	Em tempo de construção	Ambos os tempos de execução e de construção
Mudanças incrementais	Não	Não	Sim
Efeitos semânticos	Sim	Sim	Sim
Controle de impactos de mudança	Não	Não	Sim

Tabela 4.1 – Propriedades dos adaptadores de conexão de serviço.

Com base na tabela percebemos que estas conexões podem ser ajustadas, e estes ajustes têm total impacto no comportamento da aplicação com efeitos na semântica. Entretanto, adaptações em conexões de serviço baseadas em fluxo de controle e adaptadores de serviço envolvem mais esforços para serem adaptados. Conectores de serviço baseados em papel podem ser reconfigurados em tempo de execução. E as mudanças na conexão

são incrementais de forma que um novo serviço pode ser incorporado na composição de serviços enquanto os antigos co-existem [Gang 02]. No artigo [Gang 04] é usado o *framework* CASIFE [Han 03], que torna o processo de adaptação semi-automático e controla os impactos das mudanças. A comparação mostra que conectores de serviço baseados em papel provêem mais suporte para a adaptação da conexão.

Na avaliação do estudo de caso de [Gang 04] são destacados os seguintes aspectos que melhoram a adaptabilidade de conexão:

- Estabilidade de comunicação: comunicação é a função essencial de um conector de serviço baseado em papel que se encarrega da troca de dados entre os serviços envolvidos em uma interação. Por meio de características, papeis expõem uma interface de interação e aceitam clientes para invocarem serviços. Características do papel oferecem uma interface unificada para interação de serviço, melhorando a estabilidade da comunicação do ponto de vista da estrutura de conexão;
- Expansibilidade de estrutura: um conector de serviço baseado em papel é extensível no aspecto estrutural. Recursos de serviço e requerimentos de usuário são vários e mutáveis. Inevitavelmente, conexões de serviço precisam co-evoluir com mudanças. Um conector de serviços baseado em papel provê uma estrutura extensível composta de <*Feature*>, <*Services*> e <*Selector*>, que possibilitam estender e reconfigurar o conector de acordo com as mudanças;
- Adaptabilidade de conexão: o modelo de conector de serviço baseado em papel provê suporte essencial para adaptação de conexão. Com a estrutura flexível, pode ser modificado e reconfigurado. Além disso, pode acomodar mudanças de conexão até certo ponto por encapsular mudanças nos provedores se serviços. E a conexão pode ser adaptada em tempo de execução, já que um conector de serviço baseado em papel pode trocar dinamicamente provedores de serviço modificando os parâmetros para alterar a estrutura de conexão em tempo de execução.

Os autores de [Gang 04] destacaram como trabalhos futuros um conector que ofereça em tempo de execução monitoração do estado da conexão de serviço para reduzir os efeitos colaterais da adaptação, e também a aplicação do modelo encadeado, como BPEL4WS [Andrews 03].

4.2. Seleção automática de web services usando regras de transformação de grafo

A idéia do trabalho de [Hausmann 03] é permitir que um cliente possa descobrir um serviço em tempo de execução. Como a descoberta do serviço envolve a compreensão da semântica do serviço, ela é executada manualmente em tempo de desenvolvimento. Assim o autor propõe o uso de regras de transformação de grafos para descrever a semântica de *web services*, permitindo uma especificação precisa da semântica necessária para a descoberta automática de serviços.

Para a construção de uma aplicação que utiliza *web services*, o desenvolvedor consulta um servidor de UDDI e analisa os serviços disponíveis através da sua interface e possivelmente de algum texto adicional que explica os efeitos causados pelo serviço sobre o sistema, deduzindo assim sua semântica, podendo escolher um ou um grupo de serviços que realizam a tarefa desejada. No exemplo de uma compra de livros, descrita por [Hausmann 03], temos os seguintes serviços:

- placeOrder(ISBN : Integer, Address : String, CCData : Integer);
- placeOrder(ISBN : Integer, Address : String, BankAcc : Integer);

Fazendo uma analise das interfaces dos serviços podemos deduzir que eles recebem o numero do livro que se deseja comprar, o endereço onde a compra deverá ser entregue e ou o número do cartão de crédito ou o número da conta para transferência do dinheiro, devolvendo o número do pedido.

Para um sistema de descoberta automática a simples analise da interface não é suficiente, pois dela não é possível deduzir a semântica do serviço. Para que isto seja possível, em [Hausmann 03] é proposto o uso de ontologias estendidas com regras marcando a interface do servidor. Assim o cliente pode verificar de forma automática se o serviço realiza a tarefa desejada.

A figura 4.2 mostra uma ontologia para a venda de livros, em UML. A compra contém um endereço de entrega, um livro e uma conta. A conta contém um pagamento, o qual pode ser uma transferência bancária ou um cartão de crédito.

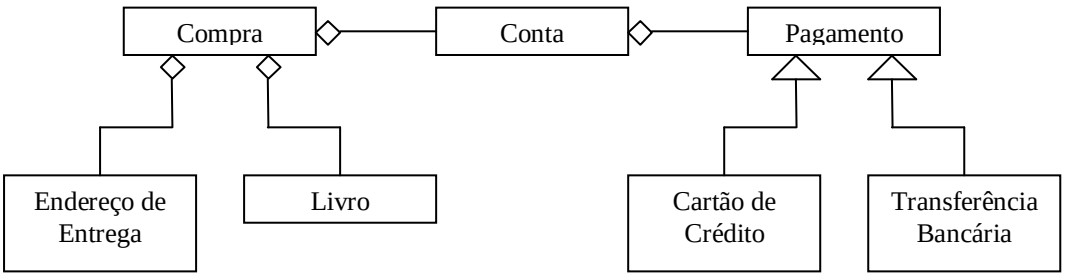

Figura 4.2 – Ontologia para venda de livros. Adaptada de [Hausmann 03].

Um *web service* pode usar ontologia para especificar sua semântica. A descrição semântica de um *web service* nada mais é do que um contrato especificando o seu significado e o seu propósito [Champion 02]. De acordo com [Fensel 03] um contrato consiste de uma pré e uma pós-condição. A pré-condição caracteriza a situação antes do comportamento ser executado e a pós-condição caracteriza a situação depois do comportamento ser executado e seus efeitos sobre o sistema. A figura 4.3 mostra o contrato de um serviço de vendas de livro sendo visualizado usando o formalismo das regras de transformação de grafo. A figura 4.3.a mostra a codificação da pré-condição na regra, e a figura 4.3.b mostra da pós-condição. O serviço recebe o livro, o endereço de

entrega e o cartão de crédito como parâmetros e devolve uma compra que é efetuada com um cartão de crédito, como especificado pela ontologia.

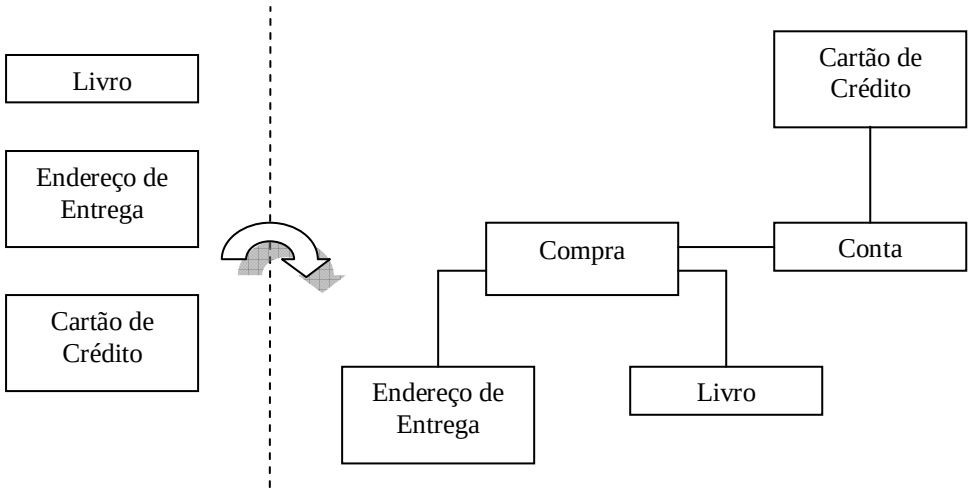

Figura 4.3 – Regra descrevendo a semântica de um *web service* para venda de livros, a) interface do serviço, b) efeitos da invocação do serviço. Adaptada de [Hausmann 03].

Para que possa ser feita a procura do serviço desejado pelo cliente, ele também precisa fazer uma descrição do serviço como mostra a figura 4.4. Na pré-condição é descrita as informações que o cliente irá enviar ao serviço e na pós-condição é descrito o que o cliente espera como conseqüência da chamada do serviço. No exemplo é esperada uma compra contendo um livro e um endereço de entrega. Essa é a exigência mínima do cliente para comprar um livro. A conta pode ser desprezada da pós-condição por estarmos primariamente interessados em comprar um livro.

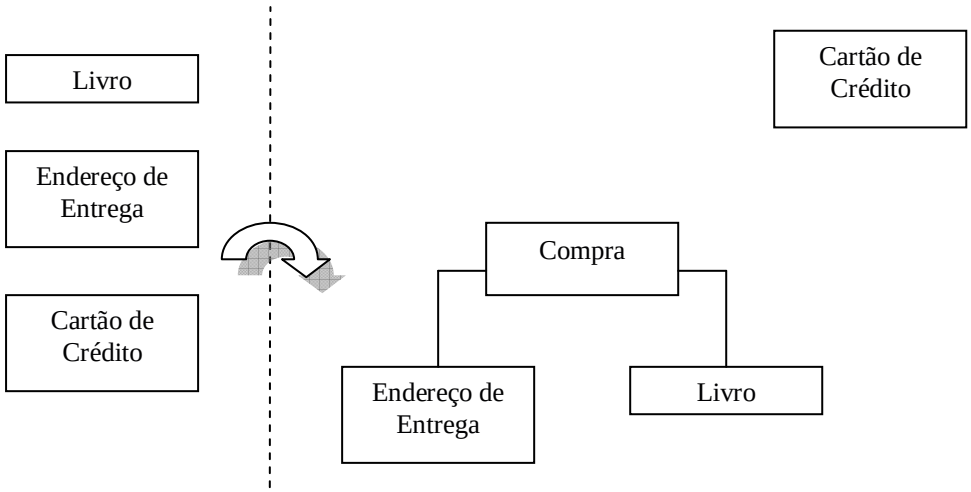

Figura 4.4 – Regra descrevendo a exigência mínima de um serviço requerido pelo cliente. Adaptada de [Hausmann 03].

Para decidir automaticamente qual serviço usar, um cliente precisa simplesmente usar um algoritmo que compare sua pré e pós-condição com a pré e pós-condição dos serviços oferecidos. Se a pré-condição do serviço é um sub-grafo da pré-condição do cliente então o cliente prove toda a informação necessária para executar o serviço; se a pós-condição do cliente é um sub-grafo da pós-condição do serviço então o serviço gera todos os efeitos esperados pelo cliente com alguns resultados a mais. Um requerimento fundamental destacado por [Hausmann 03] é que as regras do cliente e do serviço estejam escritas usando a mesma ontologia e tenham o mesmo entendimento sobre os conceitos da ontologia.

Mas um simples grafo de relação nem sempre é suficiente para decidir se um serviço satisfaz os requisitos de um cliente. Na figura 4.5 temos um cliente que fornece o cartão de crédito e a transferência bancária como pré-condição, podendo fazer o pagamento por um ou por outro, dependendo do que o serviço requisitar. O problema do uso de relação simples de sub-grafos é que a pós-condição do cliente não é um sub-grafo do serviço que ou usa um cartão de crédito ou usa uma transferência bancária. Para uma decisão automática, é necessário decompor as regras em uma pós-condição descrevendo apenas os efeitos de uma operação e uma pré-condição separada descrevendo os requisitos. Esta estrutura corresponde a uma regra de transformação de grafo com aplicação condicionada descrita por [Habel 96].

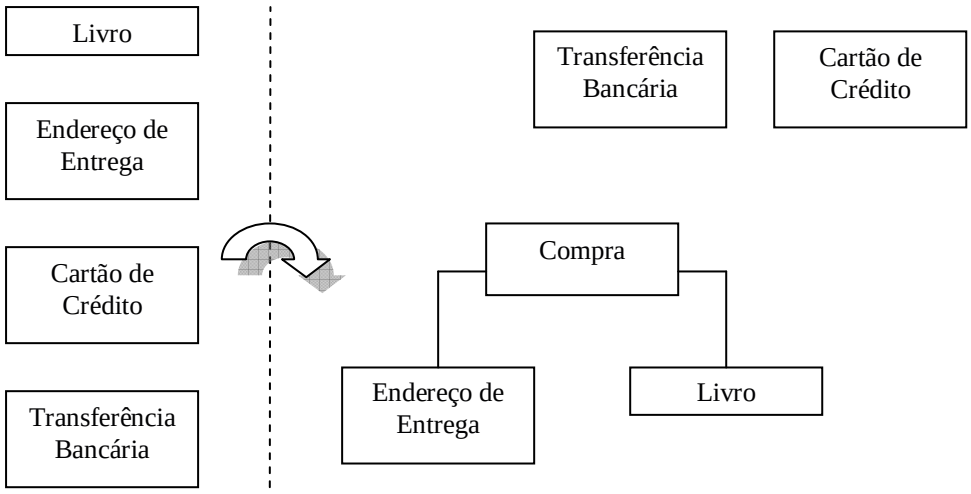

Figura 4.5 – Regra descrevendo um serviço requerido pelo cliente. Adaptada de [Hausmann 03].

Em [Hausmann 03] é destacado como um importante trabalho futuro a integração de desenvolvimento de contratos com o processo de desenvolvimento da aplicação e a implementação dos contratos baseado na semântica das linguagens *web* como WSDL.

4.3. Composição automática de web services semânticos

Como em [Hausmann 03], [Zhang 03] propõem o uso de ontologias para fazer a descoberta automática dos serviços que satisfazem a necessidade do cliente. A diferença está na forma do uso da ontologia, que em vez de ser usada para identificar as entradas do serviço e seus efeitos sobre o sistema, é usada para identificar o serviço em si. E o algoritmo não apenas escolhe um serviço, mas sim constrói uma composição de serviços, com o menor tempo de execução e o melhor fluxo de dados, para a realização da necessidade do cliente.

4.3.1 Modelagem de web services semânticos

Para a especificação da ontologia do serviço, é construída uma estrutura hierárquica de ontologia que parte do serviço mais genérico até o serviço mais especializado, como mostra a figura 4.6. Cada nodo da hierarquia irá especificar um serviço abstrato, definindo o nome e o tipo dos parâmetros de entrada e saída. Como nas ontologias de domínio, um serviço herda as propriedades do serviço pai.

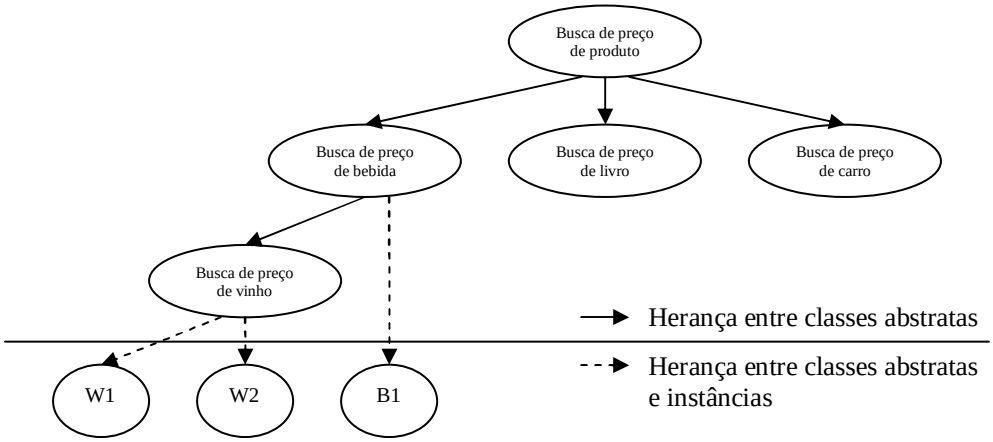

Figura 4.6 – Ontologia do domínio de procura de preço e instancias de serviços. Adaptada de [Zhang 03].

No exemplo da figura 4.6 observamos que o serviço *W1* e o serviço *W2* pertencem a mesma ontologia, assim podemos deduzir que eles tem a mesma interface e mesma semântica. Já o serviço *B1* tem uma semântica mais genérica, significando que os parâmetros de entrada e de saída recebem e retornam uma categoria mais genérica [Ankolenkar 02].

Uma necessidade do cliente é expressa através de uma questão de serviço composta (*composite service query*) de uma maneira muito similar a uma descrição de serviço em DAML-S, como na descrição 4.1. Ela inclui a descrição e a interface do serviço composto, definindo as entradas, as saídas e as restrições da composição. O usuário pode

especificar parcialmente como a composição do serviço deve trabalhar e o tipo dos serviços individuais esperados.

Interface da questão:
- Parâmetro de entrada 1: tipo da comida
- Parâmetro de saída 1: preços do vinho
- Parâmetro de saída 2: nome do vinho

Descrição da questão:
- Procura o vinho adequado com a comida
- Retorna os preços dos vinhos

Figura 4.7 – Uma pergunta de serviço composta. Adaptada de [Zhang 03].

A figura 4.7 mostra um exemplo de um dono de restaurante que quer buscar vinhos para as refeições e saber o preço destes vinhos.

4.3.2. Composição automática de serviço por emparelhamento de interface

Esta é uma técnica para a geração de composição de *web service* automática. Ela captura os objetivos do usuário e constrói uma composição de serviços que recebe as entradas fornecidas pelo usuário e devolve as saídas esperadas. A composição precisa estar de acordo com restrições especificadas, como tempo, custo e propriedades de qualidade de composição (QoC).

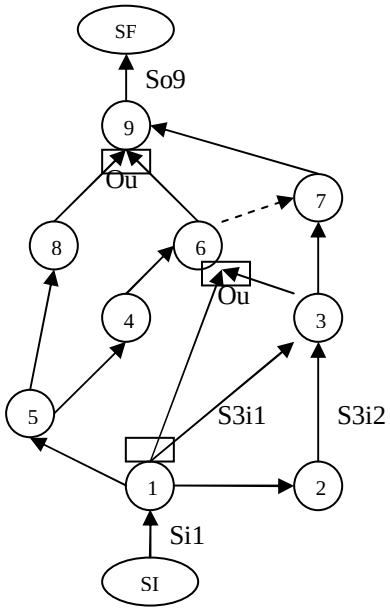

Figura 4.8 – Técnica de composição IMA

Cada serviço está relacionado com uma ontologia, que permite deduzir quais são as entradas esperadas pelo serviço e quais são as saídas produzidas. Assim é possível construir um grafo de relacionamento entre os serviços como mostrado na figura 4.8. Um serviço que produz como saída uma entrada de outro serviço é conectado com este serviço através de uma aresta direcional. O que o algoritmo faz nada mais é que compor um sub-grafo mínimo que produza os efeitos esperados pelo usuário, está será a composição de serviços ideal. A composição inicia no serviço que precisa um ou mais dos parâmetros de entrada dado pelo usuário. Se ele não produz todas as saídas esperadas, mais *web services* precisam ser encontrados para prover as saídas esperadas. Este processo continua até a composição produza a saída esperada para a entrada do usuário.

5. Composição dinâmica de Web Services

"Geralmente os clientes coletam informações de um objeto interessante chamando seus métodos. No entanto, quando um objeto interessante muda seu estado, surge um problema. Como os clientes que dependem das informações do objeto sabem que estas informações mudaram?" [Metsker 04].

E também:

"Freqüentemente temos que fazer escolhas no que diz respeito ao enfoque geral para realização de uma tarefa ou para a resolução de um problema. A maioria de nós aprendeu que tomar o caminho mais fácil a curto prazo pode conduzir a sérias complicações a longo prazo." [Adaptado de Shalloway 04].

Para uma aplicação executar uma chamada a um serviço de forma otimizada é necessário que ela saiba quais são os serviços disponíveis para requisição e destes possa escolher o serviço com maior disponibilidade, assim não é possível apenas uma chamada estática a um único serviço. É preciso saber que novos serviços estão disponíveis, e é necessário poder escolher aquele serviço que melhor otimize a aplicação segundo algum critério pré-definido.

Podemos destacar duas características que precisam estar presentes numa composição de serviços ótima:

- Receber notificação de novos serviços publicados no diretório de serviços;
- Poder escolher um dentre vários serviços disponíveis para composição.

O critério sobre a eleição do melhor serviço é atribuído à aplicação, o qual pode variar dependendo do objetivo a ser atingido pela otimização.

5.1 Arquitetura baseada em padrões de projeto

As duas características presentes na composição dinâmica de serviços podem ser capturadas por padrões de projeto. A primeira característica é capturada pelo padrão *Observer* e a segunda é capturada pelo padrão *Strategy*, as quais são compreendidas pelo padrão MCV: definir uma dependência um-para-muitos entre objetos, de maneira que quando um objeto muda seu estado todos os seus dependentes são notificados e atualizados automaticamente; e também: variar o comando a ser executado dependendo do contexto do sistema.

Usar padrões de projeto na modelagem da arquitetura de *web services* torna mais legível o diagrama e explicita o papel de cada componente dentro da estrutura.

5.1.1. Fluxo de controle

Para que a primeira característica da composição dinâmica seja suportada pelos *web services* é necessário mudar o fluxo de controle de construção. Como mostra a figura 5.1, na composição estática o diretório lista apenas uma vez para a aplicação os serviços disponíveis, muitas vezes em tempo de compilação. A partir daí a aplicação interage somente com o serviço escolhido e mais nenhum outro.

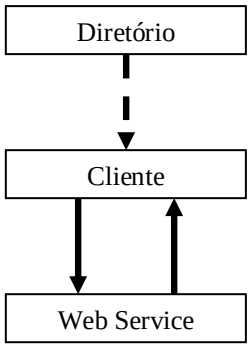

Figura 5.1 – Fluxo de controle de composição estático

A primeira característica requer um modelo onde o diretório e a aplicação estejam interagindo constantemente para que a aplicação seja notificada sobre a publicação de novos serviços. A figura 5.1 mostra o fluxo de controle de *web services* baseado no padrão MVC.

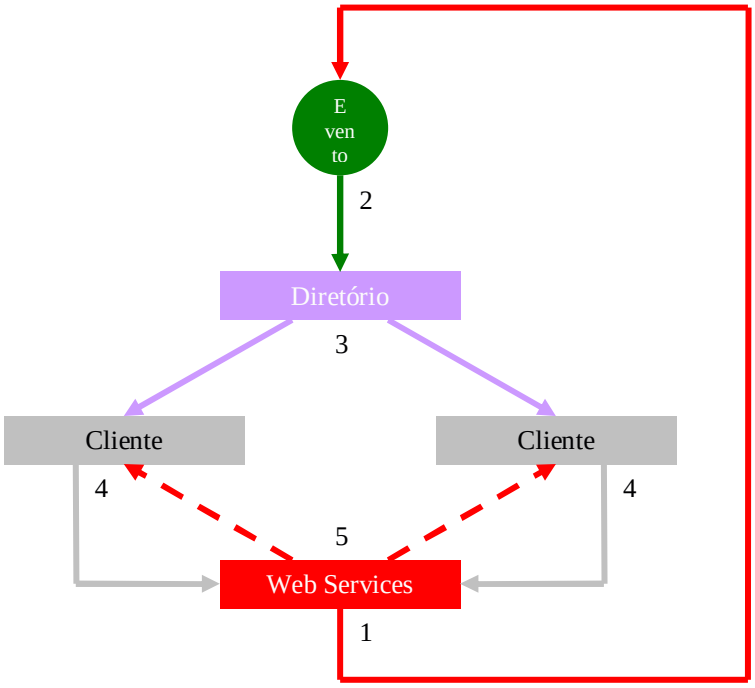

Figura 5.2 – Fluxo de controle de composição baseado no modelo MVC

O modelo do padrão é formado pelos *web services* que são os interesses dos clientes. Os clientes são como as vistas do MVC, precisam refletir o modelo para que possam garantir que a composição atual é ótima. Os diretórios são colocados no lugar do controlador, pois a responsabilidade de monitorar o modelo (publicação de serviços) é retirada dos clientes.

Quando uma publicação de um novo serviço ocorre, é gerado um evento (1). Esta publicação é capturada pelo diretório de serviços (2), por estar recebendo o registro de serviços. Assim o diretório pode notificar os clientes (3) dizendo que um novo serviço está disponível. Com a notificação os clientes atualizam a lista de serviços disponíveis e passam a levantar estatísticas sobre os novos serviços. Quando existir uma composição mais otimizada do que a composição atual, os clientes desfazem a composição e criam uma nova composição (4 e 5).

Este fluxo de controle permite que quando novo um serviço seja publicado, um cliente possa ser notificado e então refazer sua composição se for preciso.

5.1.2. Estrutura

Para que o novo fluxo de controle seja suportado e para que a arquitetura apresente as duas características pertinentes à composição dinâmica anteriormente citadas, é necessário modificar a arquitetura para que o diretório de serviços possa notificar a aplicação sobre o registro de um novo serviço e também para que a aplicação possa refazer sua composição quando necessário.

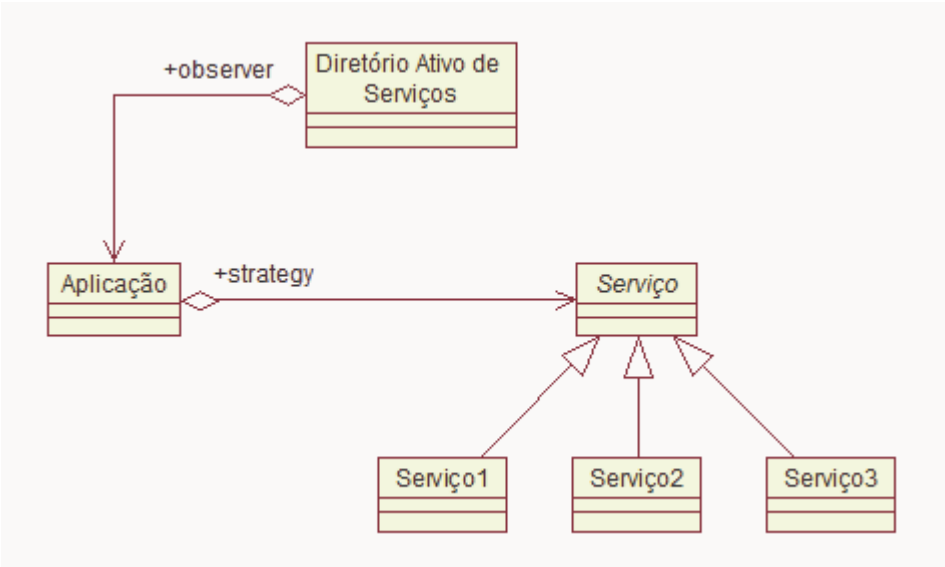

Figura 5.3 – Pseudo-arquitetura de *web services* com suporte a composição dinâmica baseada em padrões de projetos.

Cada uma das características está relacionada com um padrão de projeto, como mostra a figura 5.3. A primeira característica está relacionada com o padrão *Observer*, pois a

aplicação deve se registrar no diretório como observadora e a cada nova publicação de serviços o diretório deve notificar a aplicação, que deve então tomar as medidas necessárias para sua atualização. A segunda característica está relacionada ao padrão *Strategy*, que deve escolher o serviço mais adequado de acordo com o contexto que a aplicação se encontra. O contexto deve ser baseado na disponibilidade dos serviços para a aplicação.

O diretório de serviços cumpre o papel do sujeito, que contém as informações (que é a lista de serviços publicados) relevantes à aplicação. Os serviços são as estratégias que a aplicação deve executar.

A figura 5.4 mostra uma arquitetura real utilizando como molde a pseudo-arquitetura baseada em padrões de projeto. Para que as colaborações entre os componentes sejam realizadas os componentes precisam ser alterados e novos protocolos de comunicação precisam ser inseridos.

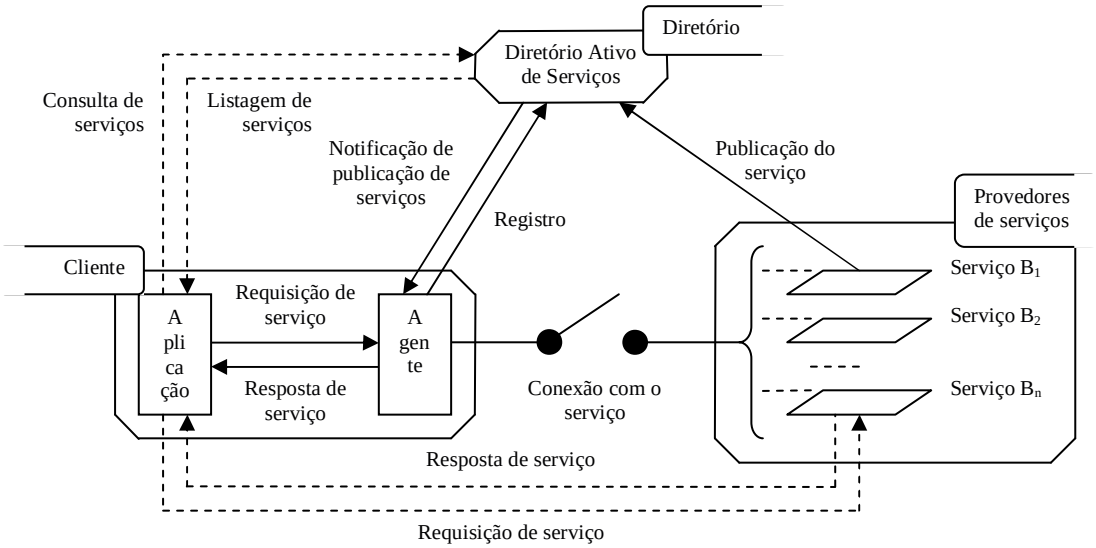

Figura 5.4 – Arquitetura de *web services* com suporte a composição dinâmica.

O diretório de serviços passa a ser chamado diretório ativo de serviços por notificar a aplicação cada vez que um novo serviço é publicado. Os protocolos de consulta de serviços e listagem de serviços passam a ser desnecessários, sendo substituídos pelo registro e notificação de publicação de serviços. O registro do cliente no diretório ativo é uma chamada *web service* que passa sua localização (URL) e qual o tipo de serviço esperado (rotulo utilizando ontologias para marcação como estabelecido na seção §4.3). Utilizando estes parâmetros, o diretório de serviços faz uma chamada *web service* para a aplicação, invocando a notificação de publicação de serviços, passando como parâmetros a localização, o nome do método e os parâmetros do serviço publicado.

O cliente precisa ser dividido em duas partes: uma que é a aplicação propriamente dita e a outra que é o agente responsável pela composição. Nesta nova estrutura a aplicação não

deve fazer a chamada diretamente ao *web service*, mas sim fazer uma chamada de método a um agente que redireciona a chamada ao *web service* com maior disponibilidade.

O agente deve funcionar como um mecanismo de QoS para poder escolher qual o melhor serviço para fazer a requisição. A principal unidade de medida é o tempo de resposta do serviço. Como a internet é muito volátil e o tempo de resposta vária constantemente, é preciso utilizar um histórico das chamadas realizadas ao serviço, esse histórico deve ser ponderado, considerando a última chamada com maior peso. Matematicamente a equação do tempo de resposta ponderado pode ser dada por [simplificado de Larson 04]:

$$T_r = \frac{\frac{T_{r-1}}{\alpha t} + T_{uc}}{2}$$

Onde T_{uc} é o tempo de resposta da última chamada, T_{r-1} é a média ponderada do tempo de resposta anterior, t é o intervalo de tempo entre a medição de T_{r-1} e T_r e α é um fator de ajuste do tempo, quanto menor t e α, maior será o peso de T_{r-1}. O serviço que tiver menor média é o serviço com maior disponibilidade e será o escolhido para composição.

O agente, por assim ser chamado, pode ser uma entidade ativa e estar periodicamente coletando informações sobre os serviços. Para isto, ele deve fazer uma chamada com todos os parâmetros nulos ao web service, medindo o tempo de resposta e atualizando suas estatísticas. Necessariamente os serviços não deverão executar nenhuma ação semântica sobre qualquer sistema e sobre si próprios, apenas deverão suportar uma execução simulada. Outras informações também podem ser coletadas pelos agentes, dependendo do que se pretende medir.

A identificação semântica do serviço é dada pela marcação ontológica hierárquica como descrito na seção §4.3. Esta marcação é uma etiqueta do tipo texto contendo todos os níveis hierárquicos do serviço separando cada nível por uma barra dupla. Como exemplo, os serviços W1 e W2 da figura 4.6 receberiam como marca a etiqueta: “Busca de preço de produto//Busca de preço de bebida//Busca de preço de vinho” e o serviço B1 receberia a etiqueta: “Busca de preço de produto//Busca de preço de bebida”. Como o serviço B1 é mais genérico que os serviços W1 e W2, seus parâmetros de entrada e de saída são um subconjunto dos parâmetros de W1 e W2, já W1 e W2 possuem os mesmos parâmetros, não diferindo na ordem e na quantidade. Serviços em outros ramos na hierarquia podem ter parâmetros em comum, mas não precisam respeitar nenhuma estruturação.

Em termos de padrões de projeto temos como participantes:

- Diretório Ativo de Serviços: contém a lista de serviços publicados;
- Cliente: composto pela aplicação e pelo agente;
    - Aplicação: faz uma requisição ao serviço através do agente;
    - Agente: estabelece qual é o serviço com maior disponibilidade para a composição e faz a interface entre a requisição do cliente e o serviço;

- Serviço: executa a ação requisitada pelo cliente.

E temos como colaborações:

- Publicação do serviço: inclui o serviço na lista de serviços disponíveis do Diretório Ativo de Serviços;
- Registro: registra a aplicação para receber notificações sobre a inclusão de novos serviços;
- Notificação de publicação de serviço: notifica a aplicação sobre a publicação de um novo serviço;
- Requisição de serviço: solicitação da aplicação para o agente fazer uma requisição a um serviço;
- Resposta de serviço: resposta do agente devolvendo à aplicação a resposta dada pelo serviço;
- Conexão com o serviço: composição entre o agente e o serviço.

As outras colaborações podem ser mantidas por questão de compatibilidade entre os sistemas, não causando nenhuma influência sobre a nova arquitetura.

5.2. Análise

A tabela 5.1, estendida da tabela 4.1, lista a comparação entre os adaptadores de conexão de serviços incluindo o conector usando composição dinâmica, apresentado anteriormente neste capítulo. Ela compara os quatro modelos de conectores de serviços até aqui estabelecidos, que são eles: conexão de serviço baseada em fluxo de controle [Casati], adaptadores de serviço [Andrews 03][Krishnan 02], conectores de serviços baseados em papel [Gang 04] e composição dinâmica.

Como esperado, o conector de composição dinâmica, por estender o conector de serviço baseado em papel, utilizando ontologias para descoberta de serviços, podendo fazer a adaptação de forma automática, atinge o grau máximo de autonomia e retira a interferência do usuário que fica apenas com a responsabilidade de especificar a qual ontologia o serviço a ser usado deve pertencer.

Enquanto que nos outros modelos é necessária a interferência de usuários ou programadores, na composição dinâmica a própria aplicação é responsável por decidir qual é o serviço que deve ser usado. Quando um serviço que estava sendo usado torna-se indisponível, nenhuma adaptação é necessária, já que a aplicação está preparada, através do agente, para executar esta adaptação, refazendo a composição com outro serviço que possui a mesma semântica, através das marcações ontológicas. Na conexão de serviço baseado em fluxo de controle e nos adaptadores de serviço, quando isto acontece, é necessária a modificação do código fonte, neste momento fica explicito o problema causado pelo desuso do padrão *Strategy*, que tenta resolver justamente este tipo de problema. No conector de serviço baseado em papel é necessário uma reconfiguração

manual da lista de serviços que podem ser usados quando nenhum serviço disponível está listado, valendo-se de uma aplicação parcial do padrão *Strategy*.

Como, na composição dinâmica, não existe a interferência de usuários ou programadores, o adaptador tem um grau de autonomia automático, não necessitando de qualquer ajuste em tempo de construção. Como nos dois primeiros adaptadores listados a modificação precisa ser feita em código fonte e pelos programadores, não existe autonomia e as modificações precisam ser feitas em tempo de construção, não permitindo qualquer mudança incremental. Já o conector de serviço baseado em papel permite refazer a sua conexão entre qualquer serviço presente em sua configuração e permite ser reconfigurado em qualquer tempo, mas a disponibilização de um novo serviço exige sua reconfiguração manual.

Na composição dinâmica não é preciso fazer nenhuma mudança incremental, não necessitando de nenhum controle de impacto de mudança. Como nenhuma mudança incremental é realizada, nenhum efeito semântico é inserido na aplicação, pois todos os serviços executados têm a mesma ontologia e assim devem possuir a mesma semântica. Já os outros três conectores apresentam efeitos semânticos por poderem refazer sua conexão com qualquer serviço independente da semântica envolvida.

Característica	**Conexão de serviço baseada em fluxo de controle**	**Adaptadores de serviço**	**Conector de serviço baseado em papel**	**Composição dinâmica**
Executor da adaptação	Programadores	Programadores	Usuários	Aplicação
Modo de adaptação	Por modificação do código fonte	Por modificação do código fonte	Por reconfiguração	Desnecessária
Grau de autonomia	Não automático	Não automático	Semi-automático	Automático
Momento de adaptação	Em tempo de construção	Em tempo de construção	Ambos os tempos de execução e de construção	Em tempo de execução
Mudanças incrementais	Não	Não	Sim	Desnecessário
Efeitos semânticos	Sim	Sim	Sim	Não
Controle de impactos de mudança	Não	Não	Sim	Desnecessário

Tabela 5.1 – Propriedades dos adaptadores de conexão de serviço [Estendida de Gang 04].

Podemos destacar os seguintes aspectos que são acrescidos ao conector de serviço baseado em papel pela composição dinâmica:

- Autonomia de reconfiguração: a localização e a medição de disponibilidade de serviços são as principais características da composição dinâmica. Isto permite que novos serviços sejam encontrados e requisitados sem nenhuma interferência do usuário ou do programador;
- Maior estabilidade de comunicação: como a interface de serviço está inserida dentro do cliente, na forma de agente, deixa de existir qualquer conexão fixa e que apresente os problemas da estrutura dinâmica e instável da internet.

A composição dinâmica de serviços é baseada da identificação semântica dos serviços e na sua disponibilidade, para que assim possa ser refeita toda e qualquer configuração necessária que tenha impacto sobre a execução da aplicação.

5.3. Implementação

A implementação do sistema foi realizada em ambiente Windows utilizando como plataforma de execução o ambiente Asp.Net 2.0 e como linguagem de programação C#. A edição e compilação do código fonte gerado foi feita com o Visual Studio 2005. Os componentes implementados foram o cliente e o diretório ativo de serviços, nenhum serviço foi implementado tendo como objetivo simular ambientes reais de execução, assim foram escolhidos serviços já disponíveis para o usuário na internet.

O código fonte completo dos componentes implementados é listado no Anexo A. Grande parte do código foi gerado pelo *wizard* do Visual Studio, sendo adptado algumas partes do código para compreender o objetivo proposto. Um pequeno *framework* foi construído para facilitar a construção da aplicação.

5.3.1. Diretório Ativo de Serviços

O diretório ativo de serviços é um *web service* com dois *web methods*. Um *web method* é responsável pela publicação do serviço e deve ser invocado quando o serviço se torna disponível pela primeira vez, ele recebe como parâmetros a localização do serviço e a ontologia que o serviço pertence. A descrição UDDI foi removida para simplificar a implementação e por ser substituída pela marca ontológica, que nos deduzir todas as informações relacionadas ao serviço. O outro *web method* é o registro do cliente no diretório ativo. Ele recebe como parâmetros o endereço do cliente e a ontologia que o serviço a ser invocado deve ter.

Como mostra o esqueleto de código da figura 5.5, ao ser publicado um serviço, todos os clientes que esperam aquele tipo de serviço são notificados recebendo como parâmetro o endereço do serviço, para então coletarem estatísticas. O mesmo acontece quando um

cliente se registra no diretório, ele recebe de imediato a notificação de todos os serviços que já estão cadastrados e que se enquadram na ontologia especificada.

O diretório faz uso do *framework* do agente (seção §5.3.2) para fazer a notificação aos clientes, chamando o *web method* Update, que é implemento pelo *web service* do agente, passando como parâmetros o endereço do novo serviço publicado.

```
[...]
public class DAS : System.Web.Services.WebService
{

    ...

    [WebMethod]
    public void Publish(string address, string ontologies)
    {
        Services.Add(new Register(address, ontologies));
        foreach (Register client in Client)
        {
            if (ontologies.Substring(0, client.Ontologies.Length) ==
client.Ontologies)
            {
                notify(client.Address, address);
            }
        }
    }

    [WebMethod]
    public void Register(string address, string ontologies)
    {
        Client.Add(new Register(address, ontologies));
        foreach (Register service in Services)
        {
            if (service.Ontologies.Substring(0, ontologies.Length) == ontologies)
            {
                notify(address, service.Address);
            }
        }
    }

    private void notify(string client_addreess, string service_address)
    {
        AbstractAgent agent = new AbstractAgent();
        agent.Url = client_address;
        agent.Update(service_address);
    }

}
```

Recebe a publicação de um serviço notificando todos os clientes registrados.

Recebe o registro de um cliente, notificando-o com todos os serviços já publicados.

Invoca o *web service* do cliente para notificação.

Figura 5.5 – Esqueleto de código do diretório ativo de serviços.

5.3.2. Cliente

O cliente é dividido em dois módulos, um deles é a aplicação propriamente dita e o outro é o agente. A aplicação contém a lógica de negocio do cliente e por tanto é específico de cada implementação e não será detalhada aqui. O que os clientes têm em comum é o código que está por traz do agente e por isso pode ser colocado em um *framework* para ser compartilhado, como mostra a figura 5.6. Este código contém o fluxo de execução para a tomada de decisão sobre qual *web service* o cliente deve ser invocado.

```
[...]
public abstract class AbstractAgent : System.Web.Services.WebService
{

    [...]

    public AbstractAgent()
    {
        DAS das = new DAS();
        das.Register(GetHostName(), ontologies);
        Collector = new Thread(new ThreadStart(this.Collect));
    }

    [...]

    [WebMethod]
    public void Update(string address)
    {
        Services.Add(new Measure(address, Alfa));
    }

    private void Collect()
    {
        while (!Exit)
        {
            Thread.Sleep(Interval);
            foreach (Measure service in Services)
            {
                [...]
            }
        }
    }

    public abstract Object CallWebService(string Address);

    public string BestServiceAddress
    {
        get
        {
            [...]
        }
    }

    public object CallBestWebService()
    {
        return CallWebService(BestServiceAddress);
    }

}
```

Registra o cliente no diretório ativo de serviços. E inicia o coletor de estatísticas

Recebe a notificação do diretório ativo de serviços.

De tempos em tempos, coleta estatísticas de todos os serviços.

Invoca o melhor serviço.

Figura 5.6 – Esqueleto de código do Framework do Agente.

Ao ser instanciado, o agente deve primeiramente se registrar no diretório ativo de serviços passando como parâmetros a ontologia e seu endereço e então deve iniciar o coletor de estatísticas. Assim podemos ver o agente como uma *thread* e ao mesmo tempo como um *web service*.

Como *web service*, o agente possui o *web method* Update que é o método a ser invocado pelo diretório ativo de serviços. Quando o diretório invoca esse método, é passado como argumento o endereço do serviço publicado, então o agente adiciona este serviço em sua lista de coleta de estatísticas.

Como o agente é uma entidade ativa, ela cria uma *thread*, a qual é a responsável pela coleta de informações sobre os serviços. Em certos intervalos de tempos a *thread* executa um *looping* sobre todos os serviços cadastrados e então mede o tempo de resposta de cada um atualizando as estatísticas sobre os serviços.

Baseado nas estatísticas o agente pode definir qual é o *web service* com maior disponibilidade e através do método *CallBestWebService*, chamado pela aplicação, este serviço é requisitado.

5.3.3. Protocolos

A tabela 5.2 lista todos os protocolos da arquitetura, onde cada um deles é implementado e quem é o responsável pela chamada.

Protocolo	**Entidade invocadora**	**Entidade**	***Web method***
Publicação do serviço	*Web service*	Diretório Ativo de Serviços	*Publish*
Registro	Agente	Diretório Ativo de Serviços	*Register*
Notificação de publicação de serviço	Diretório Ativo de Serviços	Agente	*Update*
Conexão com o serviço	Agente	*Web service*	Não definido

Tabela 5.2 – Ponto de implementação dos protocolos da arquitetura de web services usando composição dinâmica.

6. Resultados

Até este ponto foi feita apenas uma analise teórica da composição dinâmica de *web services*, para uma analise de mais baixo nível e uma validação quantitativa dos resultados foi realizado dois estudos de casos com o objetivo de extrair e analisar dados reais gerados pela composição dinâmica.

6.1. Configuração do ambiente de execução

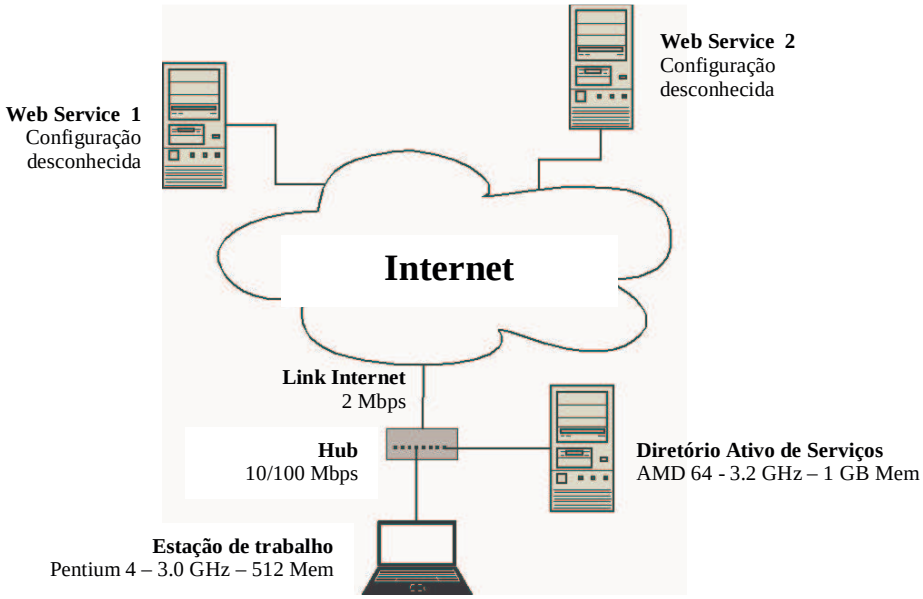

Figura 6.1 – Ambiente de execução

Para a execução dos testes de avaliação da nova arquitetura foi construído um ambiente de execução para os estudos de casos. Como mostra a figura 6.1, a estação de trabalho onde é executado o cliente fica completamente separado dos *web services*, que executam cada um em servidores distintos, na internet. A localização do diretório ativo de serviços não é importante por apresentar baixa iteração com o cliente e com os serviços, apenas necessitando ter disponibilidade para ambos, assim optou-se por colocá-lo na mesma rede local do cliente.

O cliente é executado em uma máquina *desktop* com processador Pentium 4 de 3 GHz e com memória de 512 MB. O diretório ativo de serviços também é executado numa máquina *desktop* com processador AMD de 64 bits e *clock* de 3.2 GHz. Estas duas máquinas estão na mesma rede local conectadas por um *hub fastethernet* de 10/100Mbps. O *hub* disponibiliza acesso à internet através de um link de 2Mbps com o provedor de acesso. O sistema operacional instalado nas máquinas é o *Windows XP* da *Microsoft* com

o *IES* como servidor *web*. Tanto o cliente como o diretório foram implementados na plataforma *Asp.Net* usando como linguagem de programação o C#.

O conhecimento da configuração dos servidores dos serviços é irrelevante para o cliente já que suas disponibilidades serão medidas pelo agente e assim a configuração das máquinas e velocidades de conexões poderão ser desprezadas e substituídas pelas estatísticas de disponibilidade dos serviços.

6.2. Estudos de casos

Os testes de avaliação da arquitetura se basearam em dois estudos de casos. O primeiro estudo aconteceu em cima de um sistema de analise de crédito e o outro em cima de um sistema de compra de livros. Os sistemas foram construídos usando *web services* reais e disponíveis na internet.

6.2.1. Análise de crédito

Quando uma pessoa vai fazer empréstimos em financeiras ou instituições bancárias, sua proposta normalmente é submetida a uma análise automática de crédito. A análise ocorre através da aplicação de algumas regras contidas no sistema na proposta e a principal regra a ser considerada é a consulta do CPF do cliente em alguns serviços de proteção ao crédito. Os serviços mais usados são o SPC e o SERASA. Estes serviços disponibilizam meios de conexão com seus bancos de dados para que os sistemas automáticos de analise possam realizar as consultas necessárias. E uma das formas de conexão é através de *web services*.

O sistema construído para o estudo de caso (figura 6.2) recebe uma proposta contendo um CPF de um cliente, então em posse deste CPF ele realiza uma consulta a um serviço de proteção ao crédito usando *web services*. Dois serviços foram cadastrados no diretório de ativo de serviços, o SPC e o SERASA. Assim o agente mede a disponibilidade dos serviços e então quando a consulta é solicitada ele invocava o melhor serviço.

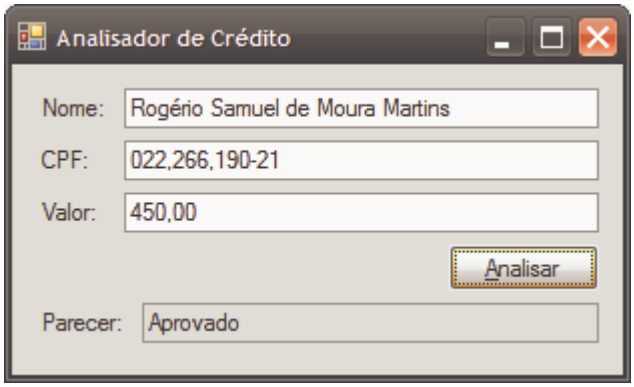

Figura 6.2 – Tela do sistema de análise de crédito usado no estudo de caso.

As tabelas 6.1 e 6.2 mostram as estatísticas levantadas pelo agente durante um dia de execução do sistema, que podem ser visualizadas nos gráficos 6.1 e 6.2. Os tempos de resposta foram gerados em milisegundos e convertidos para minutos. O alfa foi parametrizado em 1000 e o intervalo de medição foi estipulado em 1 hora.

WS	**01h**	**02h**	**03h**	**04h**	**05h**	**06h**	**07h**	**08h**	**09h**	**10h**	**11h**	**12h**
SCP	4:18	4:50	4:17	4:46	4:35	4:27	4:53	4:53	4:59	4:29	4:04	4:48
T_r^{SPC}	129	145	128	143	137	133	146	146	149	134	122	144
SRS	2:55	3:02	3:11	2:53	3:08	3:02	5:40	5:30	5:45	5:51	5:41	5:43
T_r^{SRS}	87	91	95	86	94	91	170	165	172	175	170	171

Tabela 6.1 – Tempo de resposta do SPC e do SERASA nas doze primeiras horas.

WS	**13h**	**14h**	**15h**	**16h**	**17h**	**18h**	**19h**	**20h**	**21h**	**22h**	**23h**	**24h**
SCP	4:33	5:07	4:32	4:48	4:41	4:46	4:41	4:42	4:15	4:10	4:46	4:35
T_r^{SPC}	136	153	136	144	140	143	140	141	127	140	143	137
SRS	5:34	5:55	5:48	6:01	6:03	5:45	3:14	3:05	3:47	3:43	4:00	3:54
T_r^{SRS}	167	177	174	180	181	172	127	122	118	111	125	122

Tabela 6.2 – Tempo de resposta do SPC e do SERASA nas doze ultimas horas.

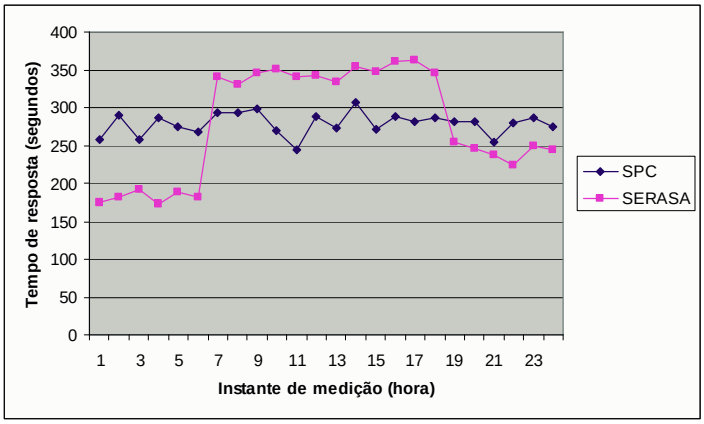

Gráfico 6.1 – Tempo de resposta do SPC e SERASA.

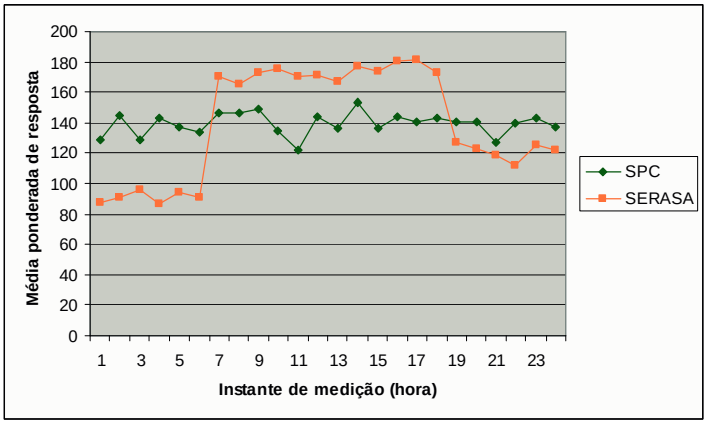

Gráfico 6.2 – Média ponderada de resposta do SPC e SERASA.

Como mostra o gráfico 6.1 o *web service* do SPC apresentou tempo de resposta mais rápido do que o *web service* do SERASA entre as 7h e 19h, nos outros momentos o serviço do SERASA apresentou maior disponibilidade. Desta forma, a figura 6.3 mostra o momento de invocação de cada *web service*, que ocorre quando apresenta menor tempo de resposta.

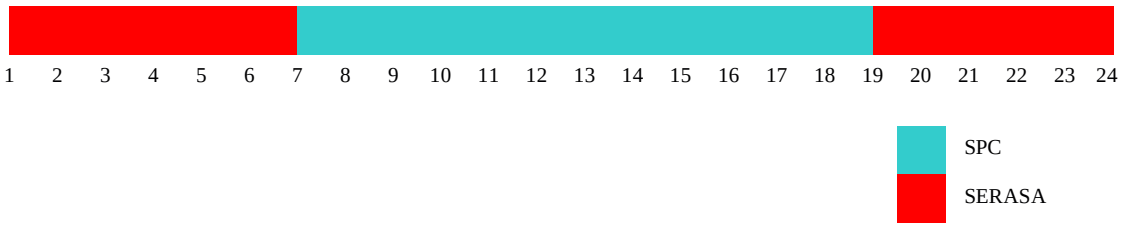

Figura 6.3 – Instante de uso de cada *web service*.

6.2.2. Consulta títulos de livros

A grande maioria dos sites de venda de livros passou a disponibilizar *web services* para que seus clientes possam consultar seus bancos de dados através de uma aplicação. Isto permite integrar o serviço de vendas de livros dos sites com outras ferramentas quaisquer sem o uso de um web broser. Assim é possível desenvolver aplicações que mostrem informações dos livros de uma livraria virtual como ilustra a figura 6.4.

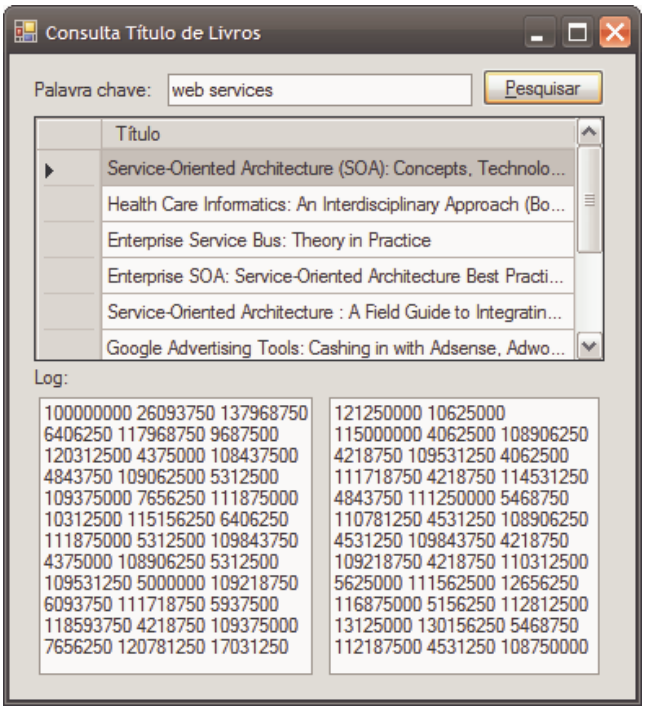

Figura 6.4 – Tela do sistema de consulta de títulos de livros na *Amazon*.

O sistema permite que o usuário entre com uma palavra-chave e a partir dela são listados todos os títulos dos livros cadastrados na livraria virtual.

A livraria virtual escolhida para este estudo de caso foi a Amazon (http://www.amazon.com) por ter sites espalhados pelo mundo. Exatamente como se pretende neste trabalho, a livraria permite escolher o *web service* com maior disponibilidade e invocá-lo, pois todos os serviços espalhados pela rede mundial possuem a mesma semântica, e mais, todos são iguais, possuindo banco de dados e regras de negócios idênticas. Dois web services foram cadastrados no diretório ativo de serviços aonde o agente do cliente se registrou. Um deles foi o serviço disponibilizado no site internacional (http://soap.amazon.com/schemas2/AmazonWebServices.wsdl) e o outro foi o serviço disponibilizado no site do Reino Unido (http://soap.amazon.co.uk/schemas2/AmazonWebServices.wsdl).

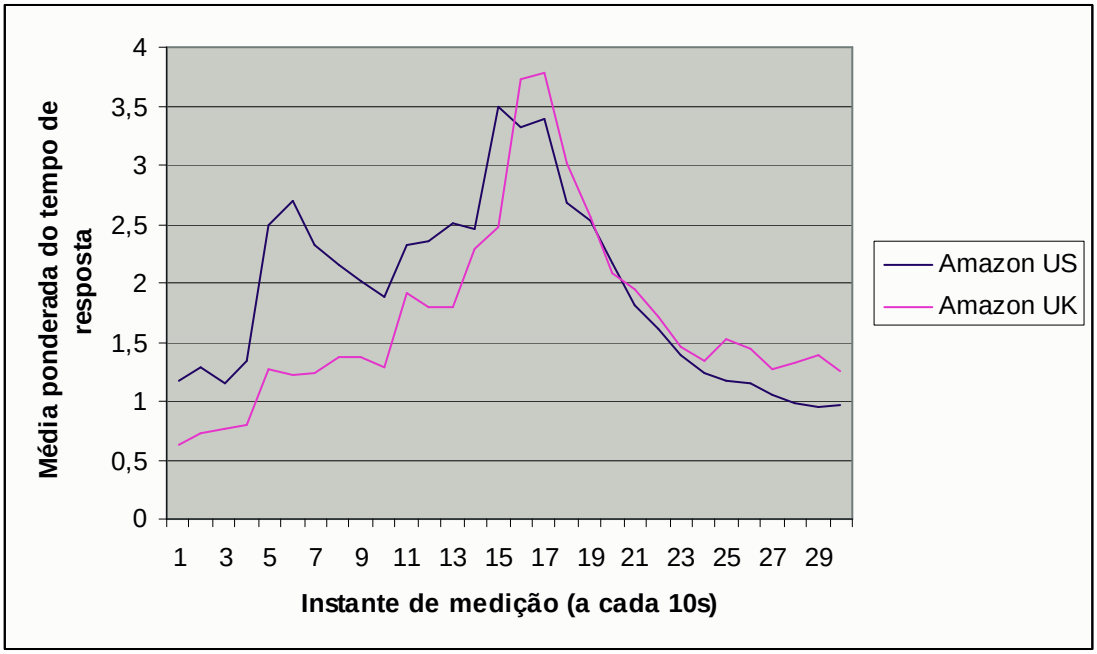

Gráfico 6.3 – Média ponderada do tempo de resposta dos *web services* da *Amazon*.

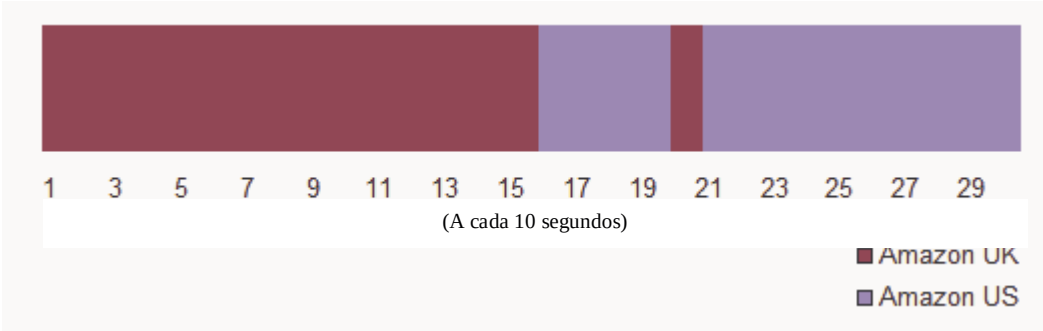

Figura 6.5 – Instante de uso de cada *web service*.

Para a execução do agente o tempo do intervalo de medição foi configurado em 10 segundos e o alfa em 7,5. O gráfico 6.3 mostra a média ponderada do tempo de resposta dos *web services*. Como o esperado, a disponibilidade entre os web services variou, tornando em alguns momentos um melhor do que o outro, assim o agente alternou as invocações sempre para o serviço com maior disponibilidade, como mostra a figura 6.5.

7. Conclusão

As medições do tempo de resposta dos *web services* nos experimentos refletem a estrutura dinâmica da internet. Os dados coletados revelam a constante variação no tempo de resposta de cada serviço oferecido, mostrando a dura realidade: não existe um padrão no tempo de resposta, o que torna imprevisível a disponibilidade de um serviço no momento de sua execução.

Aplicações, como o sistema de analise de crédito apresentado, que tem como principal requisito não funcional o desempenho e o tempo de resposta ao usuário, necessitam usar em qualquer momento o serviço que irá oferecer o melhor tempo de resposta. Para isto surge a necessidade de estar constantemente monitorando todos os serviços para medir sua disponibilidade e assim invocar o melhor serviço.

Fica evidente a necessidade da aplicação de poder medir a performance dos serviços e também poder alternar sua chamada ao serviço que melhor satisfazer algum critério estabelecido.

A arquitetura de composição dinâmica de *web services* resolve este problema através de dois padrões de projeto: *observer* e *strategy*. O primeiro permite que a aplicação atualize seu registro de serviços disponíveis e o segundo permite que escolha o serviço com melhor desempenho.

O padrão *strategy* cumpre com a principal funcionalidade da arquitetura: poder variar a invocação dos serviços. O que permitiu otimizar o tempo de resposta ao usuário aumentando significativamente o desempenho da aplicação.

Já o padrão *observer* remove dos usuários e programadores a necessidade de cadastro dos serviços a serem utilizados e habilita na aplicação a capacidade de encontrar os serviços no diretório ativo de serviços.

7.1. Trabalho futuro

Um trabalho futuro de extrema importância é o suporte a transações nas invocações. Neste trabalho a invocação simplesmente é direcionada para qualquer serviço com maior disponibilidade. Mas muitas vezes surge a necessidade de realizar uma seqüência de invocações todas para o mesmo serviço, e isto não é tratado pela arquitetura e implementação atual.

Bibliografia

[Alexander 77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max lacobson, Ingrid Fiksdahl-King, Shlomo Angel. **A Pattern Language**. Oxford University Press, 1977.

[Andrews 03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, etc. **Business Process Execution Language for Web Services Version 1.1**. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, May 2003.

[Ankolenkar 02] A. Ankolenkar, M. Burstein, J. R. Hobbs, O. Lassila, et. Al. **WS Description for the semantic web**. The First Intl Semantic Web Conference, 2002.

[Booth 03] David Booth, et Al. **Web service architecture**. W3C Working Draft, 2003.

[Bosworth 01] Adam Bosworth. **Developing web services**. Proceedings 17º International Conference on Data Engineerig. IEEE Compt Soc, (477-81), 2001.

[Buschmann 96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Machael Stal. **Pattern-oriented software architecture a system of patterns**. John Wiley and Sons, 1996.

[Casati 00] F. Casati, S. Ilnicki, J. LiJie, S. Ming-Chien. **An Open, Flexible, and Configurable System for E-Service Composition**. The Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems, Milpitas, USA, June 2000.

[Castilho 05] Daniela Castilho. **Sobre blogs e blogueiros**. http://www.digestivocultural.com/colunistas/coluna.asp?codigo=1631, 2005

[Cerami 02] E. Cerami. **Web services essentials – distributed applications with XML-RPC, SOAP, UDDI, and WSDL**. O'Reilly, 2002.

[Champion 02] M. Champion, C. Ferris, E. Newcomer, D. Orchard. **Web service architecture**. W3C Working Draft, 2002.

[Coyle 02] Frank P. Coyle. **XML, web services and data revolution**. Wesley Information Technology Series, 2002.

[Cronin 01] Gareth Cronin. **Web services: distributed computing for the new millennium?** In: University of Auckland, 2001.

[Dextra 03] Boletim Dextra #02. **Web Services na Integração de Sistemas Corporativos**. http://www.dextra.com.br/empresa/boletim/0302-02/02tecnologia.htm, fevereiro, 2003.

[Fensel 03] D. Fensel, C. Bussler. **The web service modeling framework**. 2003.

[Ferris 03] Christopher Ferris, Joel Farrell. **What are web services?** Communications of ACM. vol46#6, junho, 2003.

[Foster 02] I. Foster, C. Kesselman, J. Nick, S. Tuecke. **Grid services for distributed system integration**. Computer, vol35#6. 2002.

[Gamma 00] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. **Padrões de projeto soluções reutilizáveis de software orientado a objetos**. Bookman, 2000.

[Gang 02] Gang Li. **Adaptative software architecture and adaptative software architecture development**. Ph. D. Disseretation, Beijing University of Aeronautics and Astronautics, 2002.

[Gang 04] Gang Li, Yanbo Han, Zhuofeng Zhao, Jianwu Wang, and Roland M. Wagner. **An Adaptable Service Connector Model**. In: Chinese Academy of Science #20026180-22, 2004.

[Graham 02] S. Graham, S. Simeonov, T. Boubez, D. Davis, at Al. **Building web services with Java**. Sams, 2002

[Habel 96] A. Habel, R. Heckel, G. Taentzer. **Graph grammars with negative application conditions**. Fundamenta Informaticae. 26(3,4):287-313, 1996.

[Han 03] Y. Han, Z. Zhao, G. Li, etc. **CASIFE: An approach to enabling Adaptative service configuration of service grid applications**. Journal of Computer Science and Technology, vol. 18, no.4, 2003.

[Hansen 03] R. Hansen. Gluescript: uma linguagem especifica de domínio para composição de web services. In: Universiade do Vale do Rio dos Sinos, 2003.

[Hausmann 03] Jan Hendrik Hausmann, Reiko Heckel, Marc Lohmann. **Towards automatic selection of web services using graph transformation rules**. CiteSeer, 2003.

[Jagiello 03] I. Jagiello, E. Júnior. **Web services uma solução para aplicações distribuídas na internet**. In Pontifícia Universidade Católica do Paraná, 2003.

[Larson 04] Ron Larson, Betsy Farber. **Estatística Aplicada**. Prentice Hall Brasil, 2004.

[McIlraith 01] McIlraith SA, Son TC, Honglei Zeng. **Semantic Web Services**. IEEE Intelligent Systems, vol.16#2(46-53), 2001.

[Metsker 04] Steve John Metsker. **Padrões de projeto em Java**. Bookman, 2004.

[Newcomer 02] E. Newcomer. **Understanding web services independent technology guide**. Series Editor, 2002.

[Kratz 05] Ricardo de Andrade Kratz, **Fábrica de adequação de conteúdo de ensino para objetos de aprendizagem reutilizáveis (RLOs) respeitando a norma SCORM**. In: Universidade do Vale do Rio dos Sinos, 2005.

[Krishnan 02] S. Krishnan, P. Wagstrom, G. Laszewski. **GSFL: A Workflow Framework for Grid Services**. http://www-unix.globus.org/cog/projects/workflow/, July 2002.

[Rheinheimer 03] Leticia Rafaela Rheinheimer, Sergio Crespo Coelho da Silva Pinto. **Usando o framework JLearningServices para instanciar serviços síncronos para ambientes de EAD**. In: Simpósio Brasileiro de Informática na Educação – SBIE, 2003.

[Santanchè 03] André Santanchè, Cezar A. C. Teixeira. **Mais pontes e menos ilhas estratégia para integração de software educacional**. In: Simpósio Brasileiro de Informática na Educação, 2003.

[Scopel 05] Marcelo Scopel. **WSMEL uma arquitetura para integração de serviços educacionais usando dispositivos móveis na formação de comunidades virtuais espontâneas**. In: Universidade do Vale do Rio dos Sinos, 2005.

[Seely 02] Scot Seely. **SOAP: cross plataform web service development using xml**. Prentice Hall, 2002.

[Shalloway 04] Alan Shalloway, James E. Trott. Explicando padrões de projeto uma nova perspective em projeto orientado a objeto. Bookman, 2004.

[Souza 03] Vinicius Costa de Souza, Segio Crespo Coelho da Silva Pinto. **Sign WebMessage: uma ferramenta para comunicação via web através da Língua Brasileira de Sinais – Libras**. In: Simpósio Brasileiro de Informática e Educação, Vol1, pg. 421-430, 2003.

[Wiki 01] Portland Patterns Repository Wiki, **WebServices**. http://www.c2.com/cgi/wiki?WebServices, 2001.

[Zhang 03] Ruoyan Zhang, I. Budak Arpinar, Boanerges Aleman-Meza. **Automatic composition of semantic web services**. In: University of Geórgia Junior Faculty, 2003.

Anexo A

Abaixo a listagem dos códigos fonte dos componentes implementados.

Diretório Ativo de Serviços

DAS.cs

```
using System;
using System.Collections;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using WebService;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class DAS : System.Web.Services.WebService
{

    private static ArrayList _Services = new ArrayList();
    public ArrayList Services
    {
        get
        {
            return _Services;
        }
        set
        {
            _Services = value;
        }
    }

    private static ArrayList _Client = new ArrayList();
    public ArrayList Client
    {
        get
        {
            return _Client;
        }
        set
        {
            _Client = value;
        }
    }

    [WebMethod]
    public void Publish(string address, string ontologies)
    {
        Services.Add(new Register(address, ontologies));
        foreach (Register client in Client)
        {
```

```
            if (ontologies.Substring(0, client.Ontologies.Length) ==
client.Ontologies)
            {
                notify(client.Address, address);
            }
        }
    }

    [WebMethod]
    public void Register(string address, string ontologies)
    {
        Client.Add(new Register(address, ontologies));
        foreach (Register service in Services)
        {
            if (service.Ontologies.Substring(0, ontologies.Length) ==
ontologies)
            {
                notify(address, service.Address);
            }
        }
    }

    private void notify(string client_address, string service_address)
    {
        AbstractAgent agent = new AbstractAgent();
        agent.Url = client_address;
        agent.Update(service_address);
    }

}
```

Register.cs

```
using System;
using System.Collections;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using WebService;

public class Register
{

    public Register(string address, string ontologies)
    {
        Address = address;
        Ontologies = ontologies;
    }

    private string _Address = string.Empty;
    public string Address
    {
        get
        {
            return _Address;
        }
```

```
        set
        {
            _Address = value;
        }
    }

    private string _Ontologies = string.Empty;
    public string Ontologies
    {
        get
        {
            return _Ontologies;
        }
        set
        {
            _Ontologies = value;
        }
    }

}
```

Framework Abstract Agent

AbstractAgent.cs

```
using System;
using System.Collections;
using System.Net;
using System.Threading;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using WebService;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public abstract class AbstractAgent : System.Web.Services.WebService
{

    private static ArrayList _Services = new ArrayList();
    public ArrayList Services
    {
        get
        {
            return _Services;
        }
        set
        {
            _Services = value;
        }
    }

    private Thread _Collector;
    private Thread Collector
```

```
{
    get
    {
        return _Collector;
    }
    set
    {
        _Collector = value;
    }
}

private int _Interval;
private int Interval
{
    get
    {
        return _Interval;
    }
    set
    {
        _Interval = value;
    }
}

private bool _Exit = false;
private bool Exit
{
    get
    {
        return _Exit;
    }
    set
    {
        _Exit = value;
    }
}

private double _Alfa = 1;
public double Alfa
{
    get
    {
        return _Alfa;
    }
    set
    {
        _Alfa = value;
    }
}

public AbstractAgent()
{
    DAS das = new DAS();
    das.Register(GetHostName(), ontologies);
    Collector = new Thread(new ThreadStart(this.Collect));
}
```

```
~AbstractAgent()
{
    Exit = true;
}

public abstract string GetHostName();

[WebMethod]
public void Update(string address)
{
    Services.Add(new Measure(address, Alfa));
}

private void Collect()
{
    while (!Exit)
    {
        Thread.Sleep(Interval);
        foreach (Measure service in Services)
        {
            try
            {
                DateTime begin = DateTime.Now;
                CallWebService(service.Address);
                DateTime end = DateTime.Now;
                long tuc = ((TimeSpan)(begin - end)).Milliseconds;
                service.doMeasure(tuc);
            }
            catch (Exception)
            {
                service.Tr = double.MaxValue;
            }
        }
    }
}

public abstract void CallWebService(string Address);

public string BestServiceAddress
{
    get
    {
        string address = string.Empty;
        double bestTr = double.MaxValue;
        foreach (Measure service in Services)
        {
            if (service.Tr < bestTr)
            {
                address = service.Address;
                bestTr = service.Tr;
            }
        }
        return address;
    }
}

public object CallBestWebService()
```

```
    {
        return CallWebService(BestServiceAddress);
    }

}
```

Measure.cs

```
using System;
using System.Collections;
using System.Net;
using System.Threading;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using WebService;

public class Measure
{

    public Measure(string address, double alfa)
    {
        Address = address;
        Alfa = alfa;
        Tr = 0;
    }

    public void doMeasure(long tuc)
    {
        long t = ((TimeSpan)(LastCall - DateTime.Now)).Milliseconds;
        Tr = ((tuc / (t * Alfa)) + Tr) / 2;
        LastCall = DateTime.Now;
    }

    private string _Address = string.Empty;
    public string Address
    {
        get
        {
            return _Address;
        }
        set
        {
            _Address = value;
        }
    }

    private double _Alfa = 1;
    public double Alfa
    {
        get
        {
            return _Alfa;
        }
        set
        {
```

```
            _Alfa = value;
        }
    }

    private double _Tr = 0.0;
    public double Tr
    {
        get
        {
            return _Tr;
        }
        set
        {
            _Tr = value;
        }
    }

    private DateTime _LastCall = DateTime.MinValue;
    public DateTime LastCall
    {
        get
        {
            return _LastCall;
        }
        set
        {
            _LastCall = value;
        }
    }

}
```

Anexo B

Abaixo a listagem dos arquivos de *logs* gerados pelos clientes.

Análise de Crédito

SPC.log

01:00:00 258 129
02:00:00 290 145.0179167
03:00:00 257 128.5201414
04:00:00 286 143.01785
05:00:00 275 137.5198636
06:00:00 267 133.5191
07:00:00 293 146.5185443
08:00:00 293 146.5203498
09:00:00 299 149.52035
10:00:00 269 134.5207667
11:00:00 244 122.0186834
12:00:00 288 144.016947
13:00:00 273 136.5200024
14:00:00 307 153.5189611
15:00:00 272 136.0213221
16:00:00 288 144.0188919
17:00:00 281 140.5200026
18:00:00 286 143.0195167
19:00:00 281 140.5198638
20:00:00 282 141.0195166
21:00:00 255 127.519586
22:00:00 280 140.0177111
23:00:00 286 143.0194469
24:00:00 275 137.5198638

SERASA.log

01:00:00 175 87.5
02:00:00 182 91.01215278
03:00:00 191 95.51264058
04:00:00 173 86.51326564
05:00:00 188 94.01201573
06:00:00 182 91.01305722
07:00:00 340 170.0126407
08:00:00 330 165.0236129
09:00:00 345 172.5229199
10:00:00 351 175.5239615
11:00:00 341 170.5243783
12:00:00 343 171.5236839
13:00:00 334 167.0238227
14:00:00 355 177.5231978
15:00:00 348 174.024656
16:00:00 361 180.5241701

17:00:00 363 181.5250728
18:00:00 345 172.5252118
19:00:00 254 127.0239618
20:00:00 245 122.5176422
21:00:00 237 118.5170163
22:00:00 223 111.5164607
23:00:00 250 125.0154884
24:00:00 244 122.0173633

Consulta Títulos de Livros

AmazonUS.log

100000000 23437500
135937500 4531250
109375000 4218750
108906250 10156250
114218750 29218750
140625000 7343750
111562500 6250000
1131250008125000
116562500 6875000
113281250 7031250
111875000 18437500
135625000 5000000
109687500 15937500
123125000 7968750
125937500 28437500
133281250 4375000
143125000 4375000
108593750 4531250
110000000 11093750
115625000 4531250
108750000 4687500
113593750 4843750
109531250 4375000
108750000 4687500
110000000 5156250
116250000 4843750
110156250 4062500
108593750 4531250
112343750 4375000
112343750 4843750

AmazonUK.log

123437500 12500000
117031250 4843750
109062500 4687500
114843750 4062500
133281250 11406250
118750000 4218750
110468750 6875000

115000000 8437500
115312500 6406250
113437500 4843750
123281250 17187500
122187500 4687500
120625000 7187500
115156250 17968750
146406250 4843750
109218750 38750000
143125000 4218750
108750000 5468750
116562500 4531250
109062500 4218750
108906250 8906250
113750000 4687500
109062500 4375000
109062500 5312500
110468750 10781250
115937500 5312500
109375000 4531250
109062500 7812500
112187500 7968750
112812500 4218750