UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MARCELO REBELO BENITES

Interface para Exibição de Versões de Serviços Web

Trabalho de Graduação.

Prof. Dra. Renata Galante
Orientadora

Porto Alegre, julho de 2012.

AGRADECIMENTOS

Agradeço aos meus pais, pelo carinho, amor e pela formação moral que me deu a base necessária para superar esta etapa de vida, materializada por este trabalho de graduação. Ao meu irmão, minha admiração e minha constante vontade de imita-lo nas suas qualidades. À minha namorada, meu amor e meu agradecimento, pois vê-la enfrentar com coragem dificuldades muito maiores que as minhas, meu deu força para continuar com ânimo e esperança. A Universidade Federal do Rio Grande do Sul terá sempre minha lembrança carinhosa, pela excelência no ensino de graduação, e pelo seu maravilhoso Instituto de Informática, referencia na sua área de atuação. Por fim, à todos os professores com quem tive contato, minha profunda admiração e agradecimento, em especial minha orientadora que foi muito gentil e atenciosa, me ajudando a concluir esta última tarefa.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

WSDL	*Web Service Description Language*
XSD	XML *Schema*
API	*Application Programming Interface*
BSD	*Berkeley Software Distribution*
UFRGS	Universidade Federal do Rio Grande do Sul
XML	*Extensible Markup Language*
SVN	Subversion

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Web services são utilizados amplamente nas diversas áreas da tecnologia da informação. Um *web service* é disponibilizado por um provedor e comutado por diversos clientes concorrentes. Tendo em vista a evolução constante de um serviço, faz-se necessária a criação de versões, de forma a administrar os efeitos das mudanças nas aplicações clientes. Considerar a compatibilidade, quando na análise destes efeitos, adiciona ao provedor uma ferramenta a mais para gerência de suas versões.

O trabalho "A Flexible Approach for Acessing Service Compatibility at Feature Level", propõe um algoritmo de compatibilidade baseado num modelo granular de versionamento, além da implementação de um conjunto de funcionalidades de armazenamento e de avaliação de compatibilidade entre versões.

Este trabalho, baseado no trabalho anteriormente citado, visa construir e descrever uma interface para exibição e avaliação de versões de *web services* cujo objetivo é a visualização das versões armazenadas e de seus pontos de compatibilidade/incompatibilidade. A implementação resultante constitui uma ferramenta para auxiliar na tomada de decisões no âmbito da gerência de versões, tendo os resultados da avaliação de sua performance, quando manipulando serviços de grande porte, também apresentados neste trabalho.

Palavras-Chave: interface, versões, *web services*.

Web service version repository interface

ABSTRACT

Web services are widely used in several information technology areas. A web service is exposed by a provider and switched by different concurrent clients. In view of constant evolution of services is necessary to create versions, so that we can manage the effect of changes on client applications. Considering the compatibility in the analysis of these effects, adds to the provider a new tool to manage your versions.

The work “A Flexible Approach for Acessing Service Compatibility at Feature Level”, proposes a compatibility algorithm based on a grainy versioning model, as well the implementation of a set of features, concerning the storage and compatibility assessment of web service versions.

This work, based on previously referenced work, aims to build and describe an interface to display and assess web services versions which has the goal of displaying the stored versions evolution and its points of compatibility/incompatibility. The resulting implementation is a tool to help making decisions in the version management scope, having its performance assessment results, when manipulating large services, also presented in this paper.

Keywords: interface, versions, web services.

1 INTRODUÇÃO

Web services são amplamente utilizados como uma ferramenta de integração entre aplicações na área da tecnologia da informação. Conectando-se pela internet, sistemas escritos em diferentes linguagens de programação, executados em diferentes plataformas, interagem entre si utilizando *web services*.

Serviços estão sujeitos a mudanças e variações constantes, levando a seu re-projeto e melhoria frequentes. Por exemplo, *web services* provenientes do eBay[1], da Amazon[2], ou do Google[3] são atualizados regularmente de forma quinzenal ou mesmo semanal. Motivado pela constante mudança e necessidade de um ambiente que auxilie na tomada de decisões dentro deste contexto, um projeto está sendo desenvolvido no instituto de informática da UFRGS chamado "WS-Evolv: Um ambiente para apoio à gestão da evolução de serviços web".

Dentro do escopo do projeto anteriormente citado existe a criação de um repositório que armazene versões de *web services* e que possua mecanismos de manipulação, incluindo detecção automática de diferenças, denominada compatibilidade. O conjunto de funções do repositório, bem como o algoritmo que verifica as diferenças entre as mesmas foram desenvolvidos no artigo "A Flexible Approach for Acessing Service Compatibility at Feature Level". Uma versão A é compatível com uma versão B, quando A pode ser substituída por B, sem que haja impacto nas aplicações dependentes de A (YAMASHITA, 2011).

Este trabalho propõe a descrição e a implementação de uma interface para as funções do repositório, que possibilite um melhor relacionamento entre o usuário e as versões por ele armazenadas, de forma que este possa interagir de forma visual com as mesmas. Deverá ser possível acrescentar, acessar e visualizar as versões, bem como verificar a compatibilidade/incompatibilidade entre as mesmas.

A principal motivação deste trabalho é a necessidade da criação de uma interface para o acesso das funções do repositório que permita a visualização da evolução das versões por ele armazenadas, de forma que todos os participantes do projeto tenham acesso facilitado ao mesmo.

O objetivo do trabalho é primeiramente desenvolver um projeto de interface que satisfaça a necessidade de acesso dos diversos usuários do repositório e permita a visualização da evolução das versões dos serviços por ele armazenados. Após o projeto,

[1] developer.ebay.com/developercenter/java

[2] www.amazon.com/gp

[3] code.google.com

serão verificadas quais tecnologias estão disponíveis para auxiliar a exibição do modelo de representação de versões, sabendo-se que ele tem o grafo como estrutura básica. Definidos o projeto e as ferramentas a serem utilizadas, inicia-se o desenvolvimento da interface, explorando as funcionalidades existentes, e eventualmente modificando as mesmas e acrescentando novas funcionalidades. Finalmente será feito um teste de usabilidade vinculado a *performance* do repositório, onde será possível verificar seu comportamento quando manipulando serviços reais que quando convertidos para o modelo de versionamento proposto, geram um grande volume de dados. Eventuais problemas encontrados nesta fase de teste, poderão ter suas causas e possíveis soluções discutidas, para implementação em trabalhos futuros.

1.1 Organização do Texto

O texto será organizado em seis capítulos com os seguintes objetivos:

- Capítulo 2: apresentar a base conceitual do repositório proposto em Yamashita (2011). O modelo de versionamento, bem como o algoritmo de compatibilidade.
- Capítulo 3: discutir resumidamente interfaces existentes que possuam objetivos similares aos objetivos da interface proposta por este trabalho.
- Capítulo 4: descrever a interface implementada, discutindo as funcionalidades da mesma, bem como as razões das escolhas de seu projeto.
- Capítulo 5: descrever o comportamento da aplicação, quando carregados descrições de serviços reais no repositório. Discutir os resultados obtidos, além de possíveis soluções para os problemas encontrados.
- Capítulo 6: conclusões sobre a interface obtida e futuros trabalhos a serem desenvolvidos

2 BASE CONCEITUAL DO REPOSITÓRIO

Versões de serviços geralmente são disponibilizadas, quando o serviço sofre algum tipo de modificação em relação a alguma versão do mesmo, anteriormente disponibilizada. O modelo de versionamento descrito neste capítulo, particiona o serviço em *features*, criando versões somente para aquelas que sofreram o impacto de uma mudança. Features são versionadas individualmente como serviços, operações, ou tipos, embora mantendo seus relacionamentos, permitindo localizar e quantificar o impacto das mudanças.

Este capítulo também descreve um algoritmo de compatibilidade. Ele determina se uma versão é compatível com outra, ou seja se as mudanças da segunda com relação a primeira, tem impacto nas aplicações clientes dependentes do serviço. Esse algoritmo permite qualificar as mudanças, determinando quais *features* são compatíveis ou incompatíveis, na comparação entre as versões.

O algoritmo de compatibilidade e o modelo de versionamento propostos em Yamashita (2011), constituem a base conceitual para o entendimento da interface proposta. Primeiramente será dado uma explicação geral do contexto que motivou o desenvolvimento de ambos, de forma a situar o leitor.

2.1 Motivação do Modelo de Versionamento e do Algoritmo de Compatibilidade

Serviços são constantemente modificados, requerendo estratégias apropriadas para manter e gerenciar múltiplas versões durante seu ciclo de vida. Com a intenção de minimizar o impacto nos clientes, uma abordagem comum para gerenciar versões de serviços da perspectiva do provedor, é o versionamento da interface de descrição do serviço.

A interface de descrição se comporta como um contrato, estabelecido pelo provedor da versão do serviço, que guia os clientes em como acessar as funcionalidades do serviço. Este relacionamento é descrito na Figura 2.1.

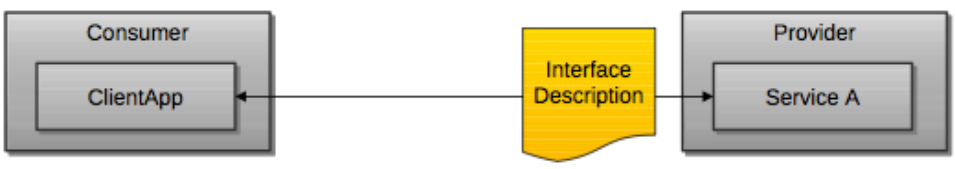


Figura 2.1: Relacionamento provedor-cliente (YAMASHITA, 2011).

Entretanto as notações atuais para interface de descrição de serviço, incluindo os padrões WSDL/XSD, não lidam adequadamente com versionamento (ANDRIKOPOULOS, 2011). Normalmente, apesar da maioria das *features* do serviço manterem-se inalteradas, todo o serviço é versionado, sendo difícil determinar o real impacto de uma mudança.

Na ausência de uma notação completa, provedores lançam novas versões dos serviços, utilizando um identificador único, conjuntamente com um documento de lançamento. No entanto, tipicamente estes documentos descrevem apenas mudanças diretas, mas falham em identificar como as mudanças se propagam pelo resto do serviço (ZOU, 2008)(FOKAEFS, 2011). Por exemplo, ao modificarmos um tipo que é referenciado por uma operação, devemos considerar que esta operação também foi afetada pela mudança.

A avaliação da compatibilidade entre versões pode prover informações preciosas sobre o impacto de uma mudança nas aplicações clientes (BECKER, 2011). Trabalhos tradicionais sobre compatibilidade, por exemplo Fang (2007), analisam a compatibilidade/incompatibilidade considerando o serviço completo. Porém, geralmente, aplicações clientes estão vinculadas a *features* específicas do serviço e avaliar a compatibilidade baseado no serviço completo não revelaria o real impacto das mudanças.

2.2 Modelo de Versionamento

O modelo proposto por Yamashita (2011) visa mensurar o escopo da mudança no serviço, para isso utiliza-se de uma granularidade mais fina para qualificar sua compatibilidade. Para este propósito qualifica-se um serviço como um conjunto de operações que podem ser acessadas por aplicações clientes. Estas operações possuem um formato particular de entrada e saída (ordenamento e tipo de parâmetros), o que é geralmente definido por elementos do esquema (referidos como tipos), o que por sua vez podem depender de outros tipos. A representação abstrata das *features* e de seus relacionamentos está descrita na Figura 2.2.

Figura 2.2: Representação abstrata do serviço (YAMASHITA, 2011).

Cada *feature* corresponde a uma porção de um documento de descrição de interface WSDL/XSD. As *features* são versionadas separadamente, mantendo-se seu relacionamento, com o intuito de gerenciar diferentes partes da interface permitindo que versione-se somente as *features* modificadas ao invés do serviço completo. Desta forma, quando um novo documento de interface de serviço é liberado, este é convertido para o um modelo abstrato interno. As descrições de suas *features* são comparadas considerando as descrições anteriores e seus relacionamentos, de forma a serem versionadas somente quando mudanças ocorrem. Como consequência um novo serviço é representado por um conjunto de *features* interdependentes associadas com novas versões, ou com versões previamente existentes.

O modelo de versionamento é exibido na Figura 2.3 utilizando um diagrama de classe UML. Uma *feature* é uma generalização de um serviço, uma operação, ou um tipo. Cada *feature* tem ao menos uma versão que por sua vez depende de outras versões vinculadas a outras *features*. A descrição da versão corresponde a descrição do documento WSDL, de acordo com o tipo (serviço, operação, ou tipo) da *feature*.

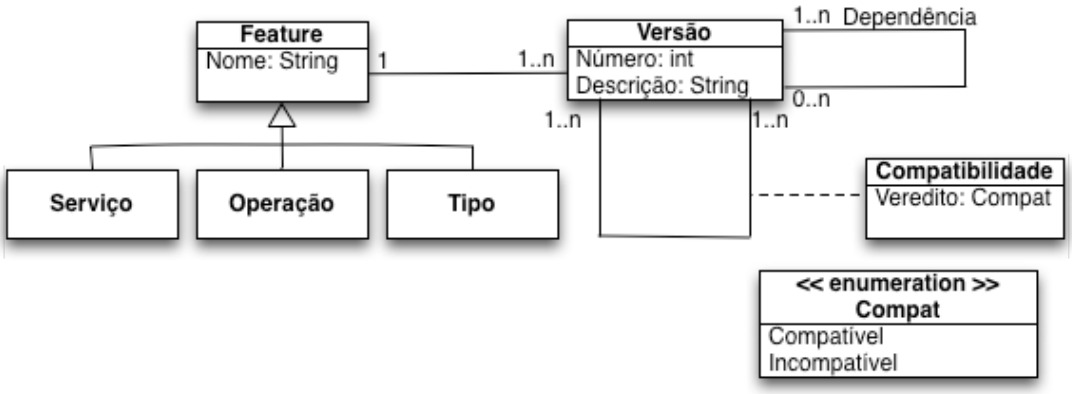
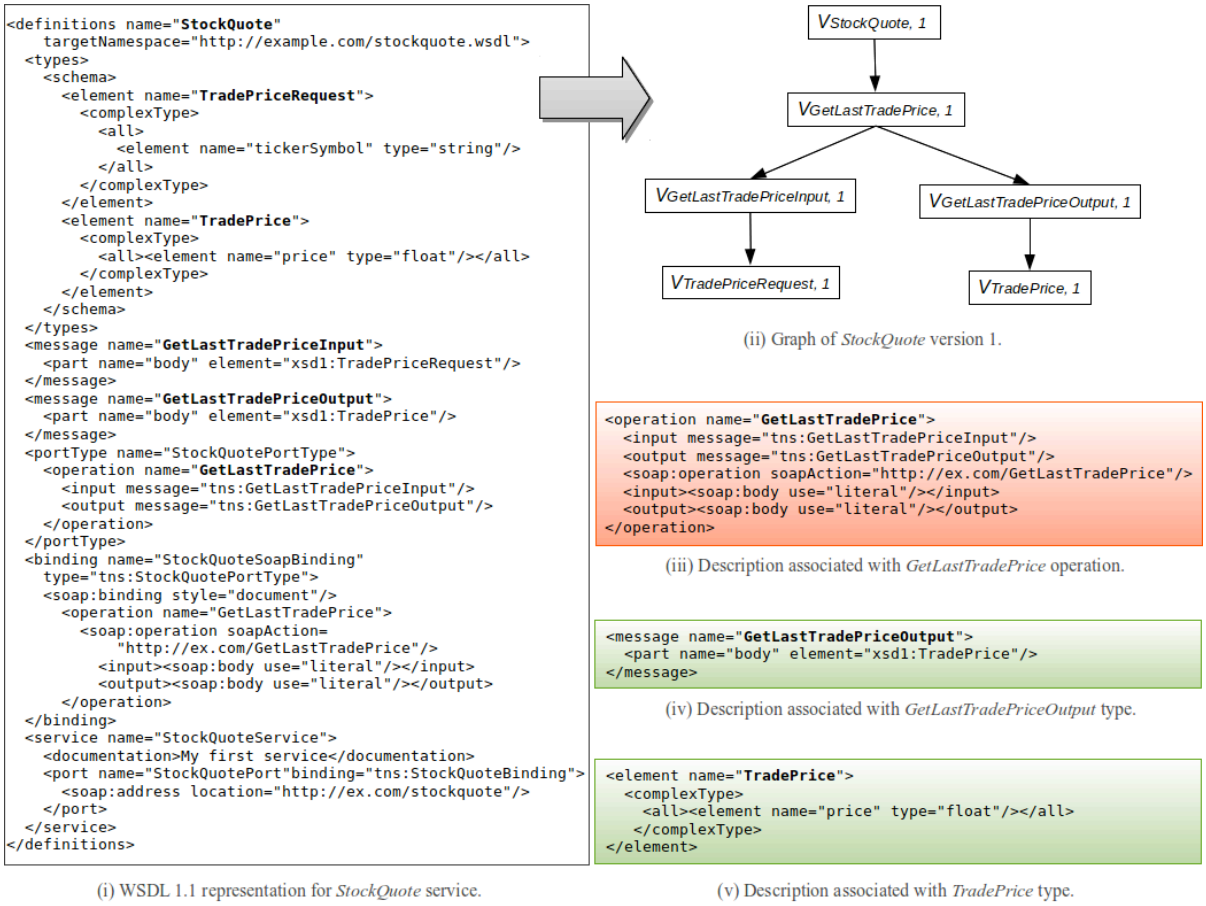


Figura 2.3: Modelo de versionamento em nível de versão (YAMASHITA, 2011).

2.2.1 Representação em Nível de Versão

Foi estabelecido a seguinte correspondência entre a representação textual WSDL/XSD do serviço e o modelo de versionamento orientado a *feature* proposto:

- Operação: relacionada ao conteúdo da *tag* <operation> dentro de ambas as *tags* <portType> e <binding>
- Tipo: relacionada ao conteúdo das *tags* <element>, <complexType> ou <simpleType> dentro da *tag* <schema>, ou do conteúdo da *tag* <message>. No que se refere a tipos, são considerados para versionamento apenas aqueles definidos fora do contexto dos elementos complexos XSD, o que significa que são versionados apenas tipos próprios para reuso. Consequentemente, não há versões de tipos primitivos (por exemplo, string, double, etc), nem de tipos compostos que não podem ser referenciados em outros lugares.



(i) WSDL 1.1 representation for *StockQuote* service.

(ii) Graph of *StockQuote* version 1.

(iii) Description associated with *GetLastTradePrice* operation.

(iv) Description associated with *GetLastTradePriceOutput* type.

(v) Description associated with *TradePrice* type.

Figura 2.4: Exemplo da descrição WSDL e da representação proposta (YAMASHITA, 2011).

É ilustrado na Figura 2.4 a maneira como são mapeados os fragmentos de uma descrição WSDL 1.6 utilizando a representação proposta, usando o serviço StockQuote. A descrição da Figura 3(i) foi separada em fragmentos para representar o serviço, suas operações e seus tipos. O resultado da fragmentação é um grafo com raiz, contendo as versões das *features* (Figura 3(ii)). Cada *feature* está associada com sua correspondente descrição, como nos exemplos na Figura 3 3(iii), 3(iv) e 3 (v).

2.2.2 Versionando as Features de uma Interface de Descrição

Com o intuito de versionar as *features* de um documento de descrição WSDL, é necessário identificar as *features* dentro do documento, relacioná-las com as versões apropriadas, possivelmente criando novas versões neste processo, e armazenar esta representação abstrata em um repositório. Isto requer dois passos: a) a conversão do documento de descrição de interface para uma perspectiva granular de *features* e b) a análise das *features* para verificar se elas mudaram, considerando todas as suas versões anteriores armazenadas no repositório.

Pretende-se versionar apenas *features* explicitamente modificadas, ou *features* que foram indiretamente modificadas pelas mudanças. Por modificada, refere-se a *feature* que teve seu fragmento de descrição modificado de alguma forma, que depende de uma *feature* que não dependia anteriormente, ou, reciprocamente, não depende mais

de uma *feature* que anteriormente dependia. Por afetada, refere-se a uma *feature* que não mudou explicitamente, mas que depende de alguma *feature* que mudou.

Primeiramente o documento de descrição de interface é convertido para um nível de representação de *feature*, o que resulta num grafo representando as versões das *features* e suas relações de dependência. Para cada *feature* são analisadas suas correspondentes no repositório de forma a comparar com verões existentes. A análise do grafo de *features* é feita com uma estratégia *bottom-up* de forma a verificar propriamente as mudanças de dependência. A análise leva a quatro possibilidades:

- Se a *feature* não existe, então ela é criada conjuntamente com sua primeira versão.
- Se a *feature* já existe (foi previamente versionada) e a sua descrição difere de todas as versões de sua *feature* particular, então ela é marcada como modificada, e uma nova versão correspondente é criada.
- Se a *feature* já existe e sua descrição é igual a de uma versão existente:
  - Se ela depende de outra *feature* que já foi marcada como modificada, então uma nova versão é criada por propagação.
  - Se ela não depende de nenhuma *feature* modificada, então cada *feature* que depende dela é referenciada para uma versão já existente (igual).

Para ilustrar a ideia do versionamento por *feature*, suponhamos que um provedor libera uma descrição de interface para a primeira versão do serviço StockQuote como demonstrado na Figura 2.4(i). Converte-se a descrição da interface para uma representação em nível de *feature*, executa-se o versionamento e cada *feature* é associada com sua primeira versão (Figura 2.4(ii)). Suponhamos agora que o provedor libere uma nova interface para este serviço que possui duas grandes mudanças: a) uma nova operação relacionada a tipos trocados em mensagens, e b) mudança no tipo de uma determinada *feature*. Por enquanto, suponhamos que o tipo relacionado com TradePrice foi modificado de *float* (Figura 2.4(v)) para *double*. Esta nova descrição é convertida para representação de *feature* e a mudança na descrição de TradePrice é identificada. Logo, uma nova versão é criada, e associada com essa *feature*. Por propagação, as *features* GetLastTradePriceOutput, GetLastTradePrice e StockQuote são afetadas, e por isso são igualmente versionadas. Ainda, as *features*, conjuntamente com suas respectivas versões, são criadas para a operação GetBestOffer, que a seu turno, depende das *features* recentemente criadas GetBestOfferInput e GetBestOfferOutput. Estas a seu turno dependem de outras *features*, que ou já existiam anteriormente (TradePrice), ou necessitam ser criadas (BestOffer, StatusType). O grafo resultante é mostrado na Figura 2.5, sendo os nodos marcados em laranja aqueles que foram afetados por propagação; os nodos marcados em vermelho e amarelo aqueles que foram modificados (incluindo os novos); os nodos marcados em vermelho, amarelo e laranja foram aqueles versionados conjuntamente com StockQuote.2.

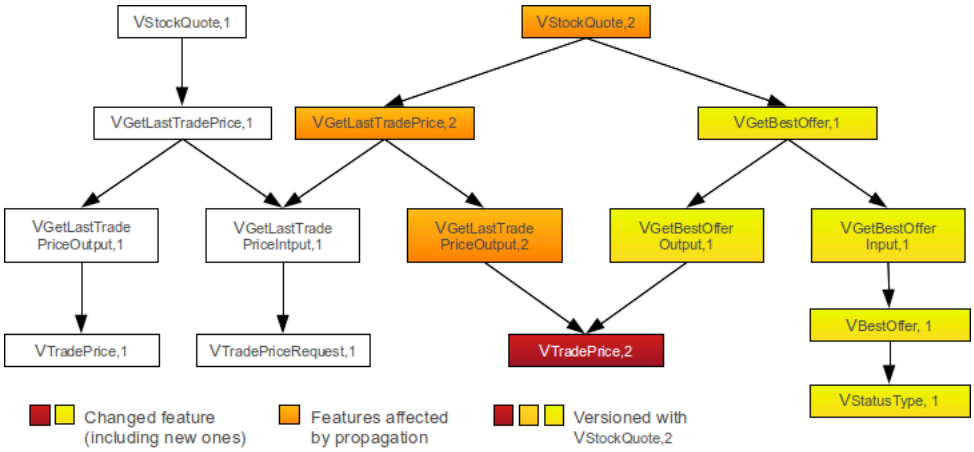


Figura 2.5: Exemplo de versionamento de StockQuote (YAMASHITA, 2011).

2.3 Avaliação de Compatibilidade

O algoritmo de compatibilidade proposto em Yamashita (2011), visa avaliar a compatibilidade entre quaisquer duas versões de um serviço, o que implica examinar a compatibilidade considerando todas as *features* que descrevem o serviço. Poucas mudanças são compatíveis com versões anteriores, basicamente adição de novas operações, e tipos que não são referenciados por operações e tipos existentes (ANDIKOPULOS, 2011)(BECKER, 2011)(FANG, 2007)(BROWN, 2004). Os casos exibidos na Tabela 2.1 traduzem os principais casos encontrados na literatura para operações de mudança no modelo proposto.

O algoritmo visa, recursivamente, avaliar a relação de compatibilidade entre duas versões de *features* de acordo com as regras da Tabela 2.1, e estabelecer a relação de compatibilidade entre elas, com o veredito correspondente (Figura 2.3). O pseudocódigo é apresentado na Figura 2.6. a função recebe duas versões de *features* como entrada, vfeature,p e vfeature,q, e avalia a compatibilidade da última em relação a primeira. Assume-se que as duas versões são relativas a mesma *feature* (tem o mesmo nome). O algoritmo executa a avaliação de acordo com o fragmento de descrição associado com as versões comparadas vfeature,p e vfeature,q (linhas 1-3), e então, recursivamente, avalia a compatibilidade de todas as versões de *features* dependentes correspondentes (linhas 4-10), e finalmente marca as relações de compatibilidade e o veredito (linha 11). O grafo com raiz vfeature,q é percorrido em profundidade, o que possibilita a propagação das incompatibilidades detectadas para as versões dependentes.

O primeiro passo do algoritmo (linha 2) visa avaliar se as dependências da *feature* foram removidas de vfeature,q comparativamente com vfeature,p. A função evaluateRemovedDependencies verifica se todas as *fatures* no conjunto de dependências de vfeature,p ainda existem no conjunto de dependências de vfeature,q. Detecta-se que uma dependência foi removida, então as versões vfeature,p e vfeature,q são incompatíveis de acordo com os casos 5 e 6 da Tabela 2.1. O algoritmo também avalia a descrição textual associada com as versões vfeature,q e vfeature,p. A avaliação da descrição é detalhada na Figura 2.7.

Então, o algoritmo percorre as *features,* as quais vfeature,q depende, com o intuito de avaliar a sua compatibilidade com suas correspondentes no conjunto de dependência em vfeature,p que possuem o mesmo nome e diferente número de versão (linha 5). Então, o algoritmo é chamado recursivamente para avaliar a compatibilidade destas versões (linha 6). Se qualquer dependência é incompatível, então o algoritmo atualiza o veredito para incompatível por propagação. Se existe uma versão dependente a vfeature,q que não existe em setOfDependencies(vfeature,p), então isto indica que a nova versão para os casos 1 e 2 da Tabela 2.1 é incompatível. Finalmente, o algoritmo retorna a avaliação compatibilidade de vfeature,q de acordo com vfeature,p. Note-se que o algoritmo não poderia parar em qualquer ponto onde a incompatibilidade é detectada, porém optou-se por continuar a avaliação para deixar espaço para mais tarde adicionar a descrição de todas as inconsistências encontradas.

Tabela 2.1: Casos de mudança para compatibilidade de versões

Casos	Mudança	Tipo de *Feature*	Descrição	Compatibilidade
1	Adição	Operação	Adicionar nova operação a um serviço	Compatível
2	Adição	Tipo	Adicionar novo tipo como dependência de uma nova operação/tipo	Compatível
3	Adição	Tipo	Adicionar novo tipo como dependência de uma operação/tipo existente	Incompatível
4	Atualização	Tipo	Mudança na descrição relativa a ordem, cardinalidade ou tipo	Incompatível
5	Remoção	Operação	Remoção de dependência de operação	Incompatível
6	Remoção	Tipo	Remoção de dependência de tipo	Incompatível

Fonte: YAMASHITA, 2011. p. 6.

```
Listing 1 compatibilityAssessment(v_featureP, v_featureQ)
1  boolean compat ← true;
2  compat ← evaluateRemovedDependencies(v_featureP ,v_featureQ);
3  compat ← compat ∧ evaluateDescription(v_featureP , v_featureQ );
4  foreach v_depQ,j ∈ setOfDependencies(v_feature,q) do
       // If there is a dependency feature version with the same name and different version
5      if exists v_depP,i ∈ setOfDependencies(v_featureP) ∧ (depP = depQ) ∧ (i ≠ j) then
           // If the assessment of any dependency is false, then the dependent is set false
6          compat ← compat ∧ compatibilityAssessment(v_depP,i, v_depQ,j)
9      end if
10 end foreach
11 setVerditct(v_feature,q, v_feature,p , compat);
12 return compat;
```

Figura 2.6: Avaliação de Compatibilidade (YAMASHITA, 2011).

A avaliação da descrição Figura 2.6 (linha 4), detalhada na Figura 2.7, visa avaliar os casos restantes de incompatibilidade (3 e 4) da Tabela 2.1. A avaliação de descrição recebe como entrada as versões vfeature,p e vfeature,q e se suas descrições são diferentes (linha 2), ele avalia para os casos de incompatibilidade. Atualmente, a avaliação de descrição considera apenas *features* de tipo.

```
Listing 2 evaluateDescription(v_featureP, v_featureQ)
1  boolean compat ← true;
2  if v_feature,p(description) ≠ v_feature,q(description) then
3     if v_featureP(Feature.Type) = type then
4        foreach e_j ∈ setOfElements(v_feature,q) do
5           if not exists e_i ∈ setOfElements(v_featureP) ∧ e_i(name) = e_j(name) then
6              compat ← false;
7           else if (e_i(order) ≠ e_j(order) ∨ e_i(type) ≠ e_j(type) ∨ e_i(cardinality) ≠ e_j(cardinality)) then
8                 compat ← false;
9           end if
10       end foreach
11       foreach e_i ∈ setOfElements(v_featureP) do
12          if not exists e_j ∈ setOfElements(v_featureQ) ∧ e_i(name) = e_j(name) then
13             compat ← false;
14          end if
15       end foreach
16    else
17       compat ← false;
18    end if
19 end if
20 return compat;
```

Figura 2.7: Avaliação de Descrição (YAMASHITA, 2011).

Com o intuito de extrair os elementos da descrição e suas propriedades do fragmento WSDL, a função setOfElements é responsável por analisar a parte do WSDL correspondente a versão. O algoritmo verifica se existe algum elemento em vfeature,p (linha 5), que leve a verificar o caso 3 da tabela 2.1. Depois, se o elemento existir em ambas as versões compara-se suas propriedades (linha 7) com o intuito de verificar se elas mudaram. A comparação de propriedades dos elementos refere-se ao caso 4 da Tabela 2.1. Elementos de descrição removidos são verificados nas linhas 11 a 15.

Finalmente, a avaliação de descrição retorna o veredito da compatibilidade para vfeature,q considerando vfeature,p.

2.4 Considerações Finais

Este capítulo apresentou o modelo de versionamento e o algoritmo de compatibilidade propostos em Yamashita (2011), de maneira que possa ser compreendido, em seguida, o funcionamento da interface e do conjunto de funcionalidades no qual ela se baseia. A interface deverá exibir os grafos de dependência resultantes das agregações de novas versões ao modelo de versionamento, além de apontar os nodos compatíveis/incompatíveis executando o algoritmo de compatibilidade.

A interface proposta e desenvolvida neste trabalho, utilizando o embasamento teórico exposto neste capítulo funcionará como uma ferramenta de visualização da evolução não só dos serviços, como também, com a granularidade mais fina do modelo de versionamento, das operações e dos tipos armazenados. A comparação entre as versões, utilizando o algoritmo de compatibilidade, permitirá analisar o impacto das modificações, determinando os pontos específicos de incompatibilidade e compatibilidade das mesmas. Desta forma o usuário poderá visualizar cada *feature* extraída de uma interface de descrição de serviço, além de suas relações de dependência utilizando o grafo como ferramenta.

3 TRABALHOS RELACIONADOS

Com o intuito de prover subsídios para o desenvolvimento da interface proposta, foi conduzida uma pesquisa por trabalhos que tivessem objetivos relacionados ao desta monografia. As finalidades principais a serem relacionadas são: a apresentação de dados de maneira hierárquica e a apresentação de grafos de dependência.

Em uma primeira varredura por aplicações que visualizam dados de forma hierárquica, naturalmente voltamo-nos para *softwares* da área da tecnologia da informação. Dentro desta gama de aplicações, está o Visual Studio 2010[4], ambiente de desenvolvimento fornecido pela Microsoft[5]. Neste capítulo vamos nos ater à funcionalidade chamada Solution Navigator[6], que tem como principal característica o gerenciamento de arquivos de maneira hierárquica.

Com relação a exibição de grafos de dependência, por ser uma funcionalidade específica, houve dificuldade em encontrar trabalhos relacionados, porém dentro da esfera de trabalho do autor, temos o ambiente de desenvolvimento chamado Eclipse[7], que desde 2008, possui um *plugin* que visa gerar grafos de revisão. Disponibilizado pela Apache Foundation[8] para ser utilizado em conjunto com o Subversion[9], sistema de versionamento de *software* e controle de revisões, este *plugin* tem como grande desafio a interação com grafos de grande porte.

Este capítulo descreve parcialmente a funcionalidade Solution Navigator e o *plugin* gerador de grafos de revisão, atendo-se as características dos mesmos que possam ser aproveitadas no desenvolvimento da interface proposta.

3.1 Solution Navigator

Solution Navigator é uma interface que apresenta elementos do Visual Studio 2010 (arquivos, classes, métodos, atributos, etc) de maneira hierárquica na figura de uma

---

[4] www.microsoft.com/visualstudio/pt-br/home-produtos

[5] www.microsoft.com

[6] http://blogs.msdn.com/b/visualstudio/archive/2010/07/20/solution-navigator-blog-post.aspx

[7] www.eclipse.org

[8] http://www.apache.org/

[9] http://subversion.apache.org/

árvore (Figura 3.1). De seu conjunto de funcionalidades, destacam-se a expansão e a busca de nodos na árvore.

A expansão dos nodos, repassa ao usuário a responsabilidade de exibir os dados dos nodos, agregando interatividade e dinamismo a atividade, sendo possível ao clicar em um determinado nodo, esconder ou mostrar seus filhos.

A busca de nodos permite ao usuário digitar o nome parcial de um dos nodos independente de seu tipo (arquivo, classe, método, atributo, etc...), e a árvore será expandida de forma a focar nos elementos resultantes. Ainda é possível selecionar um nodo específico da árvore e realizar a busca com o escopo reduzido aos seu filhos.

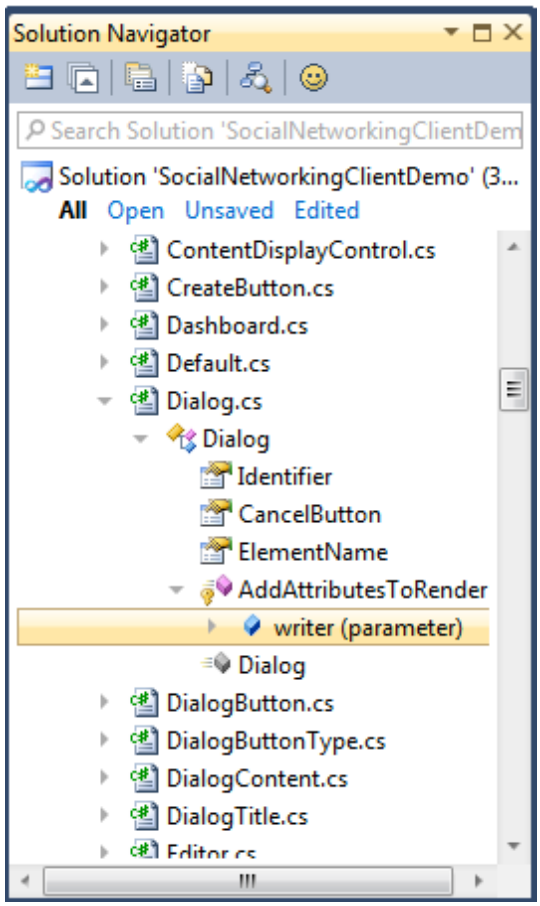


Figura 3.1: Solution Navigator

Todas essas funcionalidades foram aproveitadas no desenvolvimento da interface proposta neste trabalho. A apresentação hierárquica, no entanto foi utilizada para a apresentação das versões dos *features* armazenadas considerando o modelo de versionamento detalhado no capítulo anterior, ao invés de apresentar classes, métodos e atributos como no Solution Navigator. A busca seja no conjunto completo de nodos da árvore, seja no escopo reduzido dos filhos de um determinado nodo, foram aproveitadas igualmente neste trabalho, sendo possível procurar por todas as *features* armazenadas no repositório, ou apenas no conjunto de *features* dependentes de uma determinada *feature*.

O Visual Studio 2010 não permite a exibição do conjunto de classes, métodos e atributos, expostos na árvore na forma de um grafo, apenas é possível visualizar o código escrito. A falta dessa possibilidade, pode ser considerada uma desvantagem na sua implementação, pois o usuário, caso esta funcionalidade estivesse disponível poderia visualizar as relações existentes entre estes componentes de forma visual. A interface proposta neste trabalho permite a visualização das *features* na forma de um grafo, disponibilizando desta maneira ao usuário a possibilidade de visualizar as relações de dependência existentes entre as mesmas.

3.2 Gerador de Grafos de Revisão

O denominado gerador de grafos de revisão é um *plugin* para o SVN que visa a exibição das versões armazenadas num repositório na forma de um grafo. O que é interessante nesta extensão é que ela lida, eventualmente, com um conjunto grande de nodos, sendo complicado, nestes casos, a exibição completa do grafo. Com o intuito de resolver este problema, duas soluções complementares foram implementadas: o esboço do gráfico e o *zoom*.

O esboço do grafo é uma interface adicional, que funciona como um referencial exibindo constantemente todos os nodos do grafo independente do *zoom* utilizado (Figura 3.2). Desta forma, ao interagir com grafos de grande porte, o usuário pode focar em determinados nodos, mantendo a noção do conjunto total de nodos do grafo.

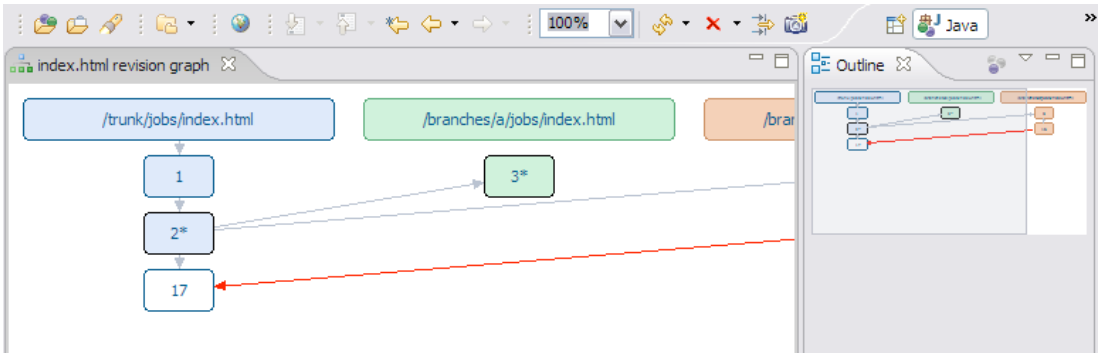


Figura 3.2: Zoom e esboço do grafo

Na interface proposta e desenvolvida neste trabalho, explorou-se o esboço do grafo, desta mesma forma, conjuntamente com o *zoom*. A diferença existente entre o gerador de grafos de revisão e o grafo de dependência exibido na interface proposta, é a disposição dos nodos. Os nodos na interface de exibição de versões, em função da sua estrutura possuir uma única raiz, foram expostos de maneira hierárquica horizontal, exibindo da esquerda para a direita, serviço, operações e tipos.

4 INTERFACE DE EXIBIÇÃO DE VERSÕES DE SERVIÇOS WEB

Baseado no modelo de versionamento proposto em Yamashita (2011), no qual particiona-se a interface de descrição do serviço em *features*, buscou-se desenvolver uma interface. Esta interface tem o intuito de possibilitar um fácil acesso as diversas versões de *features* armazenadas no repositório utilizado pelo modelo, além de exibi-las utilizando o grafo derivado da relação de dependência entre elas.

Conhecendo-se o modelo de versionamento e a necessidade da apresentação das versões utilizando um grafo, projetou-se uma interface que possui as seguintes funcionalidades básicas:

- Acesso as versões do repositório;
- Agregação de novas versões de serviços ao repositório;
- Exibição utilizando grafo de dependência;
- Exibição dos pontos de compatibilidade/incompatibilidade das versões;

O conjunto de funções disponibilizadas para o acesso ao repositório de versões tiveram Java como linguagem de origem, por isso optou-se pela continuidade do desenvolvimento nesta linguagem. Considerou-se que uma eventual troca acarretaria migração desnecessária do código, enquanto a permanência permitiria o reuso dos componentes já desenvolvidos.

Tendo a linguagem definida, iniciou-se uma pesquisa por um *framework* que suporta-se o desenvolvimento gráfico em Java e que disponibiliza-se a gama de componentes necessários para construção da interface, principalmente no que tange a exibição do grafo. Definidas as ferramentas e tendo um projeto capaz de suprir as necessidades definidas pelas funcionalidades desejadas, iniciou-se a implementação.

Este capítulo visa descrever como o projeto foi desenvolvido utilizando o *framework* escolhido. Primeiramente haverá uma breve descrição das ferramentas utilizadas, em seguida cada funcionalidade proposta será discutida individualmente. Durante o processo de descrição, serão ressaltados os motivos das escolhas efetuadas, tendo em vista o embasamento teórico apresentado nos capítulos anteriores.

4.1 Arquitetura e Ferramentas Utilizadas para Implementação

O *framework* Swing[10] consiste em um conjunto de ferramentas para desenvolvimento de interfaces gráficas em Java. Swing possui uma aparência nativa que

[10] http://docs.oracle.com/javase/tutorial/uiswing/

tenta emular as aparência de algumas plataformas, isto se dá em função da não utilização de código específico da plataforma em que ele é executado. Ao invés disso, ele é escrito inteiramente em Java sendo independente de plataforma. Esta flexibilidade foi um dos fatores para a escolha do Swing para o desenvolvimento da interface proposta.

Swing possui um conjunto de componentes avançados para construção de interfaces, como janelas roláveis, árvores, tabelas e listas. No entanto, Swing não possui suporte específico para exibição de grafos, o que era essencial para o projeto, por isso foi necessário buscar uma nova API que cumprisse este papel.

JGraphX[11] é uma biblioteca de exibição de grafos *open source* baseada em Swing, disponibilizada pela licença BSD[12]. JGraphX foi escolhida para complementar o desenvolvimento da interface principalmente pela boa documentação e por ser baseada em Swing, permitindo facilitada interação com os outros componentes da interface. Durante a descrição da interface, alguns de seus componentes serão referenciados e descritos em mais detalhes.

A arquitetura da interface proposta baseia-se no relacionamento entre os componentes Swing, as funcionalidades desenvolvidas em Yamashita (2011), o repositório de versões e os componentes pertencentes a API JGraphX. O esboço do funcionamento desta arquitetura pode ser visualizado na Figura 4.1.

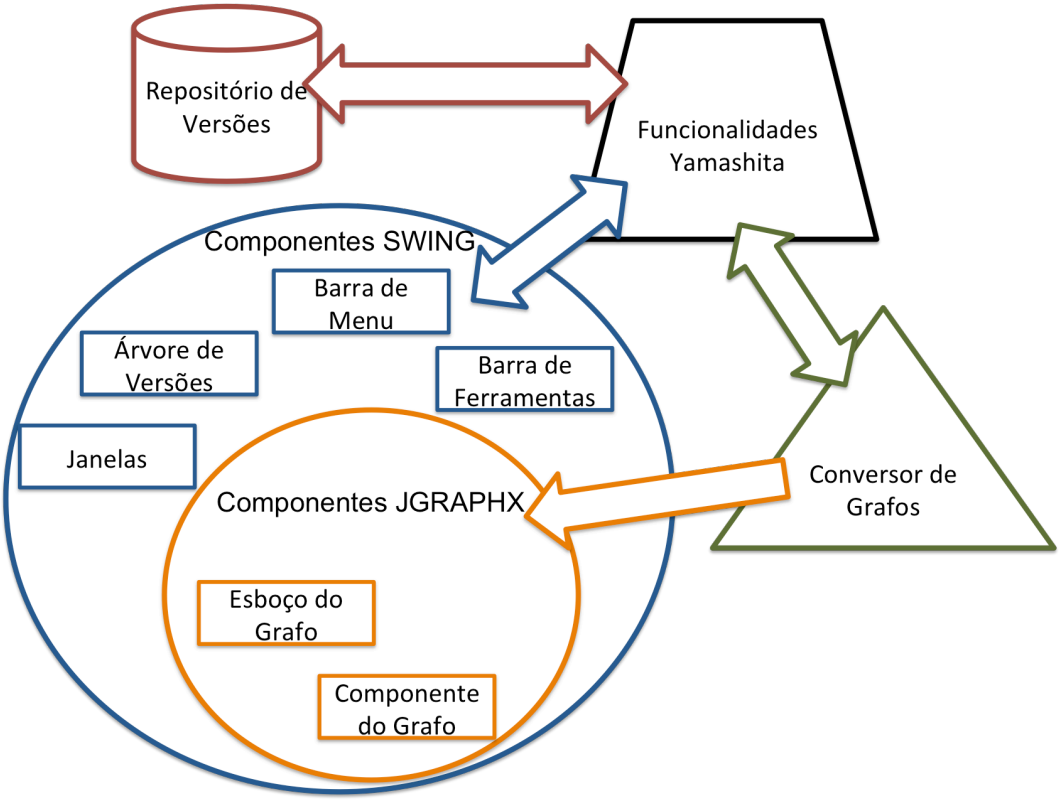


Figura 4.1: Arquitetura da Interface Proposta

[11] http://www.jgraph.com/jgraph.html

[12] http://pt.wikipedia.org/wiki/Berkeley_Software_Distribution

Os diversos componentes Swing da interface interagem com as funcionalidades desenvolvidas em Yamashita (2011), de forma a buscarem informações do repositório de versões. A principal entidade no relacionamento entre a interface e as funcionalidades supracitadas é o chamado Conversor de Grafos. Ele recebe um grafo em um formato diferente do utilizado pelos componentes pertencentes a API JGraphX, e o converte de forma que este possa ser exibido na interface.

4.2 Acesso as Versões do Repositório

O primeiro ponto a ser abordado na construção da interface, é o acesso as versões armazenadas no repositório. Apesar das versões estarem vinculadas a um grafo, sua estrutura é hierárquica, de forma que todas possuem uma raiz e subjacentes nodos, logo sua apresentação poderia se dar utilizando-se uma árvore. Em razão da natureza recursiva das gramáticas de grafos, haverá alguns nodos pais e filhos repetidos na árvore (GUDURU, 2011).

O Swing possui um componente específico para visualização de árvores, chamado JTree (Figura 4.1). Com ele é possível apresentar dados de forma hierárquica e, principalmente dinâmica. A questão do dinamismo associada a este componente nos é muito interessante, pois em função do grande volume de versões as quais devemos permitir acesso, faz-se necessária uma exibição parcial das mesmas. Sendo transferida ao usuário a responsabilidade de escolha de quais versões interagir, não polui-se a interface agregando agilidade na busca por versões. Este tópico será discutido em mais detalhes no Capítulo 5.

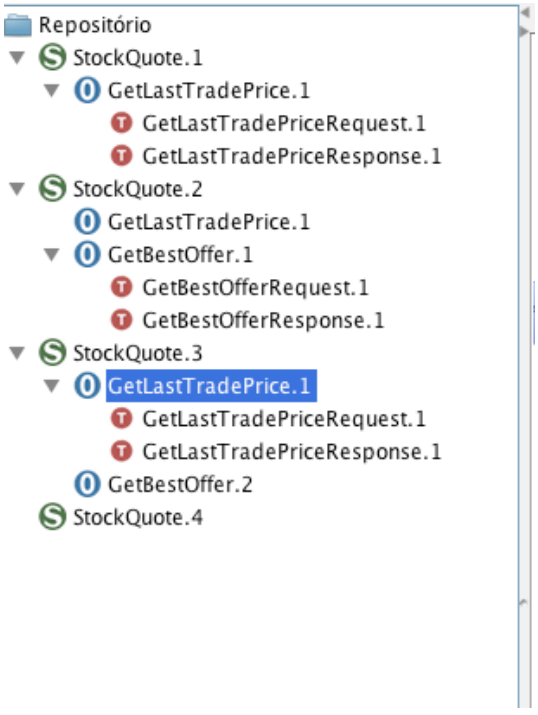


Figura 4.2: JTree – visualização de versões armazenadas

Com o intuito de facilitar a visualização do tipo de *feature* associado a cada nodo da árvore, utilizou-se três imagens diferentes. A letra ”S” envolta na cor verde, representa um nodo vinculado a um serviço. A letra “O” envolta na cor azul, representa

um nodo vinculado a uma operação. Por fim, a letra "T" envolta na cor vermelha representa um nodo vinculado a um tipo.

4.2.1 Pesquisa de Versões na Árvore

Considerando um universo em que muitas versões estão armazenadas no repositório, apesar da estratégia de exibição dinâmica descrita anteriormente, faz-se necessária uma funcionalidade específica de pesquisa. Para tanto possibilita-se que o usuário digite o nome parcial de uma versão de *feature*, de forma que o sistema retorne as versões correspondentes marcando-as na árvore de exibição (Figura 4.2).

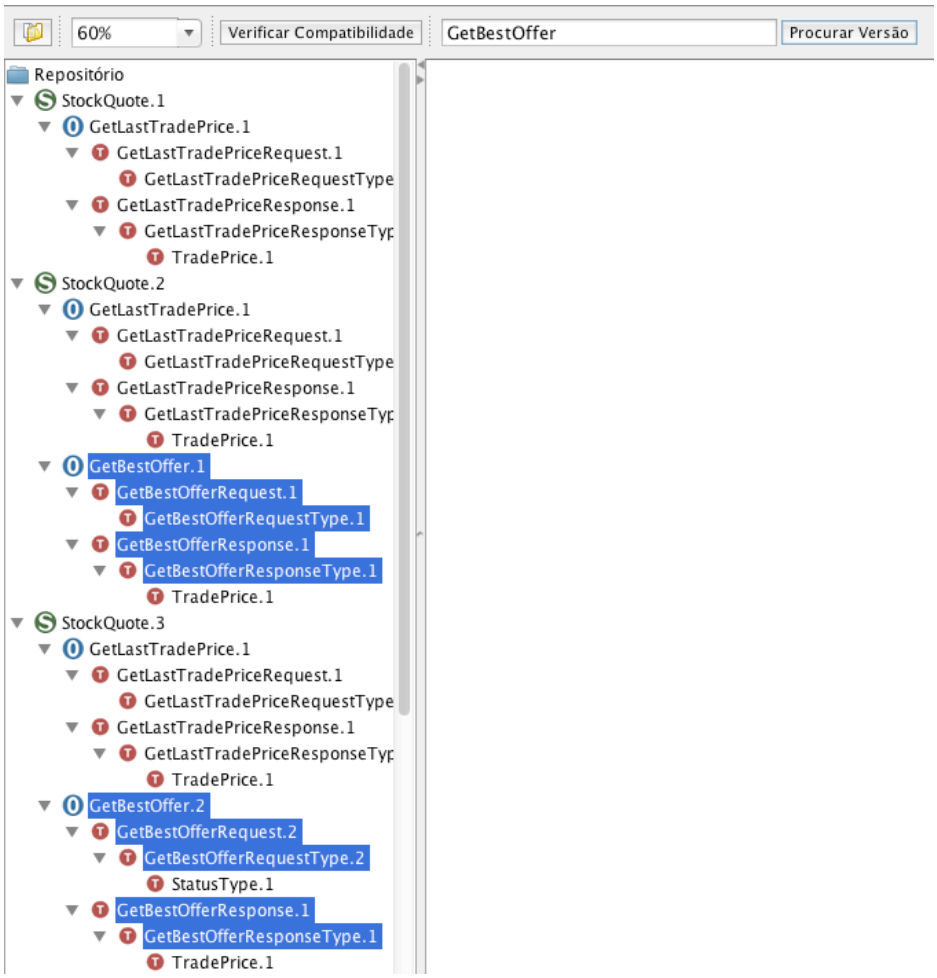


Figura 4.3: Exemplo de busca de *feature*

É importante ressaltar, que a pesquisa proposta considera o nome da versão da *feature*, e não o nome da *feature*. O nome da versão da *feature* é composto pelo nome da feature, seguido de ".", seguido do número da versão. Por exemplo, a primeira versão da *feature* StockQuote, tem o nome StockQuote.1.

A pesquisa por versões funciona em duas partes. Primeiramente busca-se no repositório, todas as versões prefixadas pela *String* de entrada fornecida pelo usuário.

Conseguintemente as respectivas versões encontradas são assinaladas na árvore como demonstrado na Figura 4.2.

Ainda é possível fazer uma pesquisa com escopo reduzido, pois isto permite otimizar o tempo de resposta, resultando numa melhor experiência de usuário na utilização do sistema. Como demonstrado na Figura 4.3, o usuário pode escolher se deseja pesquisar um tipo de *feature* específico (serviço, operação, ou tipo), ou ainda selecionando com um botão secundário (e.g. o botão direito do *mouse*) um determinado nodo de *feature* na árvore, reduzir o escopo às versões de *feature* dependentes deste nodo.

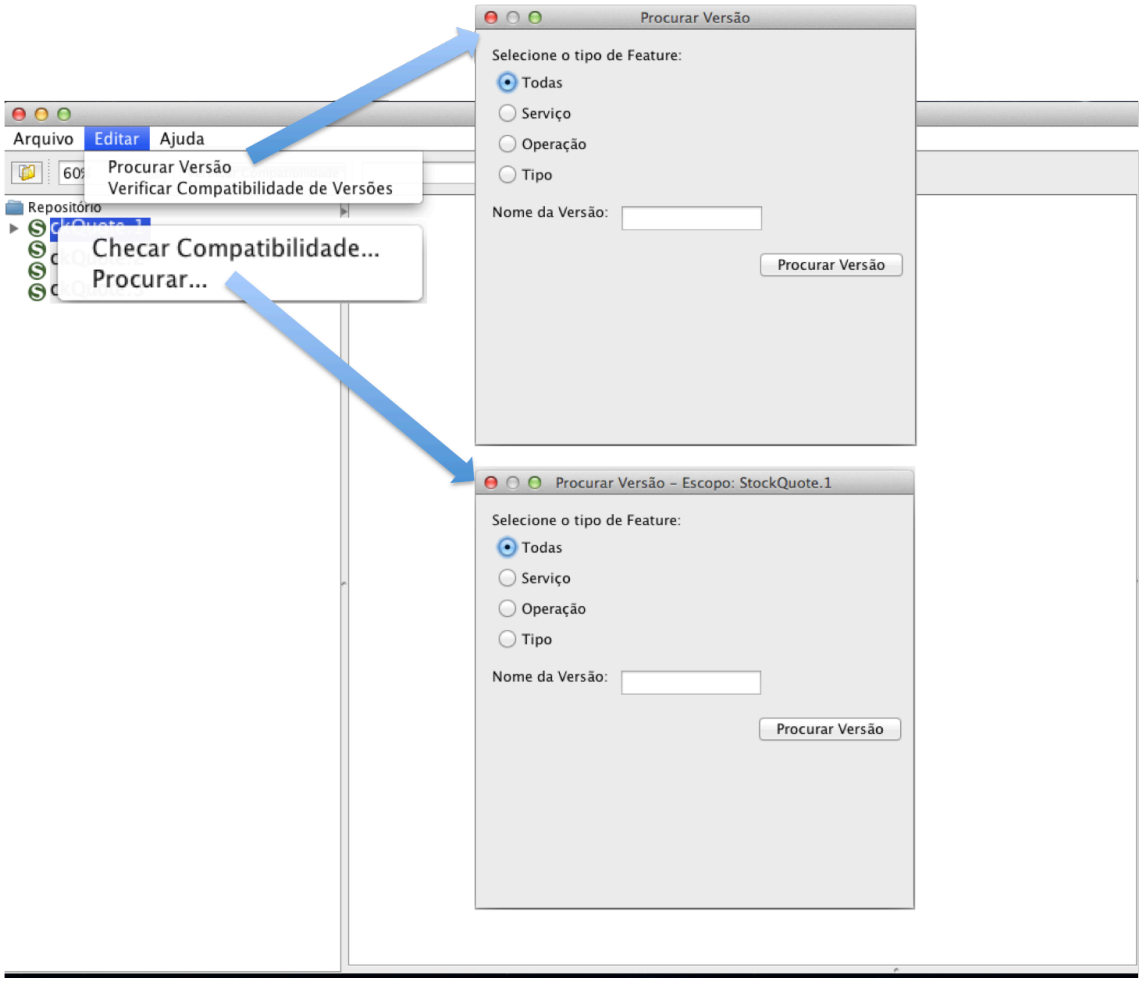


Figura 4.4: Tipos de busca de *feature*

Por exemplo, como observado na Figura 4.2, as versões dos serviços StockQuote.2 e StockQuote.3 possuem *features* dependentes prefixadas pela *String* "GetBestOffer". Como a busca efetuada neste exemplo não possui escopo reduzido, todas as ocorrências encontradas foram marcadas. Caso o escopo fosse reduzido, por exemplo a StockQuote.2, apenas as ocorrências dependentes deste serviço seriam marcadas.

4.3 Agregação de Novas Versões de Serviços ao Repositório

Naturalmente, a interface deve permitir que novas versões sejam incorporadas ao repositório. Para tanto é permitido que o usuário busque descrições de interface de serviço WSDL no computador em que o *software* está rodando, para que elas sejam agregadas ao repositório de acordo com o modelo e o algoritmo descritos no Capítulo 2.

Para adicionar uma versão ao repositório, o usuário deve fornecer o nome do serviço correspondente e um arquivo com extensão .wsdl, como descrito na Figura 4.4.

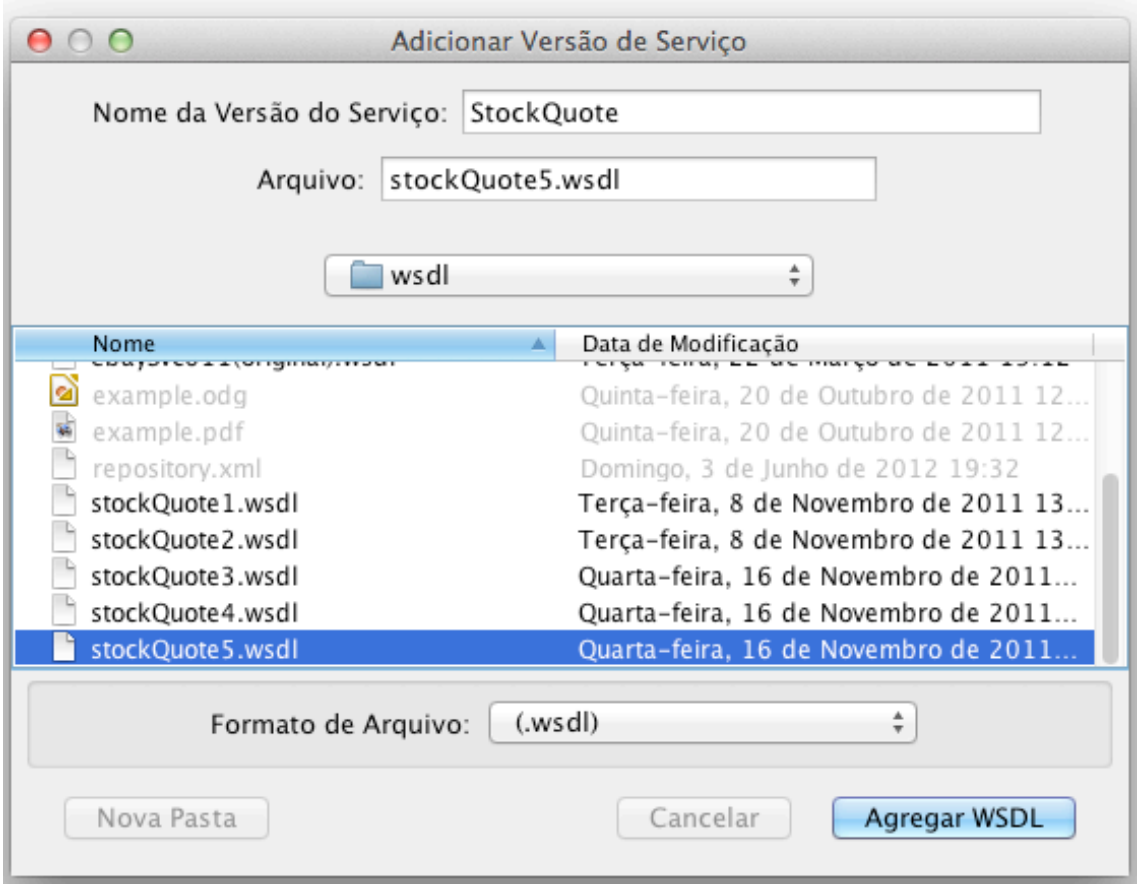


Figura 4.5: Agregação de versão de serviço

É importante ressaltar que o nome do serviço fornecido pelo usuário irá impactar diretamente no algoritmo responsável pela agregação da versão ao repositório. Por exemplo, considerando-se que o repositório possuísse somente a primeira versão do serviço StockQuote armazenada. Imagina-se o cenário no qual o usuário tenta agregar o documento WSDL, correspondente a essa versão já armazenada, porém fornecendo um nome diferente de “StockQuote”, como por exemplo “stockQuote”. Neste caso, uma nova versão seria agregada ao repositório com os mesmas dependências de StockQuote.1, embora com o nome stockQuote.1. Porém, se o usuário fornece o nome idêntico ao do serviço armazenado, no caso “StockQuote”, o algoritmo é capaz de identificar que a versão já está armazenada e a interface assinala que a versão já consta no repositório, não sendo permitida a agregação.

4.4 Exibição Utilizando Grafo de Dependência

O propósito principal da interface de exibição de serviços é apresentação do grafo de dependência. O grafo representa as relações entre as *features* armazenadas no repositório, sendo de responsabilidade do sistema a apresentação interativa deste grafo.

Explorando a funcionalidade de gerenciamento de eventos da JTree, é possível captar a interação do usuário com as versões. Quando um duplo clique em um determinado nodo é detectado, o grafo é exibido na tela. Para tanto, é preciso utilizar um método chamado buildGraph, pertence ao conjunto de funcionalidades implementadas em Yamashita (2011). No entanto, está funcionalidade retorna um grafo construído utilizando um formato diferente da nossa API de exibição JGraphX, por isso uma conversão é efetuada.

Após a conversão, teremos um conjunto de nodos rotulados pelo nome da versão da *feature*, composto por NomeDaFeature.NúmeroDaVersão. Eles serão exibidos dentro de um componente chamado mxGraphComponent, que possui uma série de funcionalidades de exibição. Algumas delas, utilizadas pela aplicação, serão descritas nas próximas seções.

4.4.1 Layout do Grafo

JGraphX disponibiliza uma funcionalidade que automaticamente dispõe os nodos de um grafo em um determinado *layout*. Dentre as opções oferecidas temos o estilo de exibição hierárquico horizontal, no caso das nossas *features* temos a seguinte configuração resultante: o primeiro nível é formado por somente um nodo, rotulado com o nome da versão da *feature* selecionada, em seguida as dependências deste nodo são apresentadas nos níveis seguintes, como demonstrado na Figura 4.5.

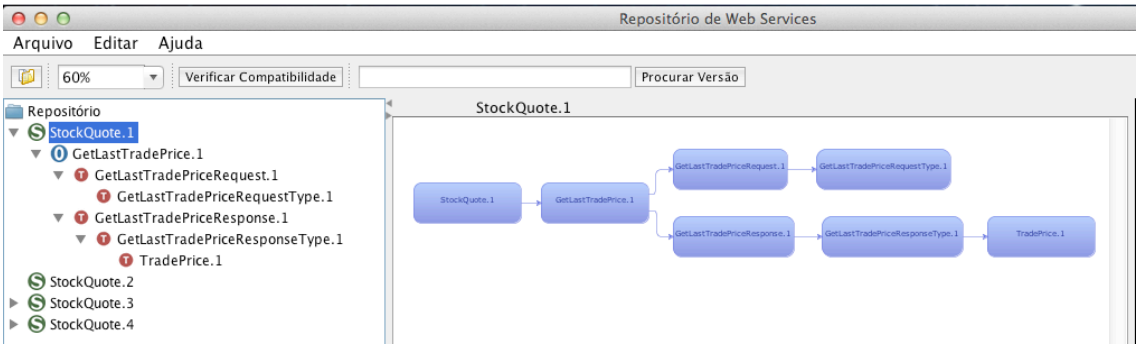


Figura 4.6: Exibição grafo de dependência versão StockQuote.1

4.4.2 Zoom e o Esboço do Grafo

Tendo em vista a grande quantidade de *features* armazenadas quando versionados serviços de grande porte, acarretando grandes grafos de dependência, fez-se necessário disponibilizar ao usuário mecanismos de *zoom*. Com este mecanismo seria possível aproximar certas áreas do grafo de dependência de forma a facilitar sua avaliação.

Conjuntamente com a funcionalidade de *zoom*, utilizou-se a classe mxGraphOutline pertencente a API JGraphX. Esta classe possibilita visualizar um esboço do grafo apresentado e determinar em que parte do mesmo o zoom está sendo aplicado, como demonstrado na Figura 4.6.

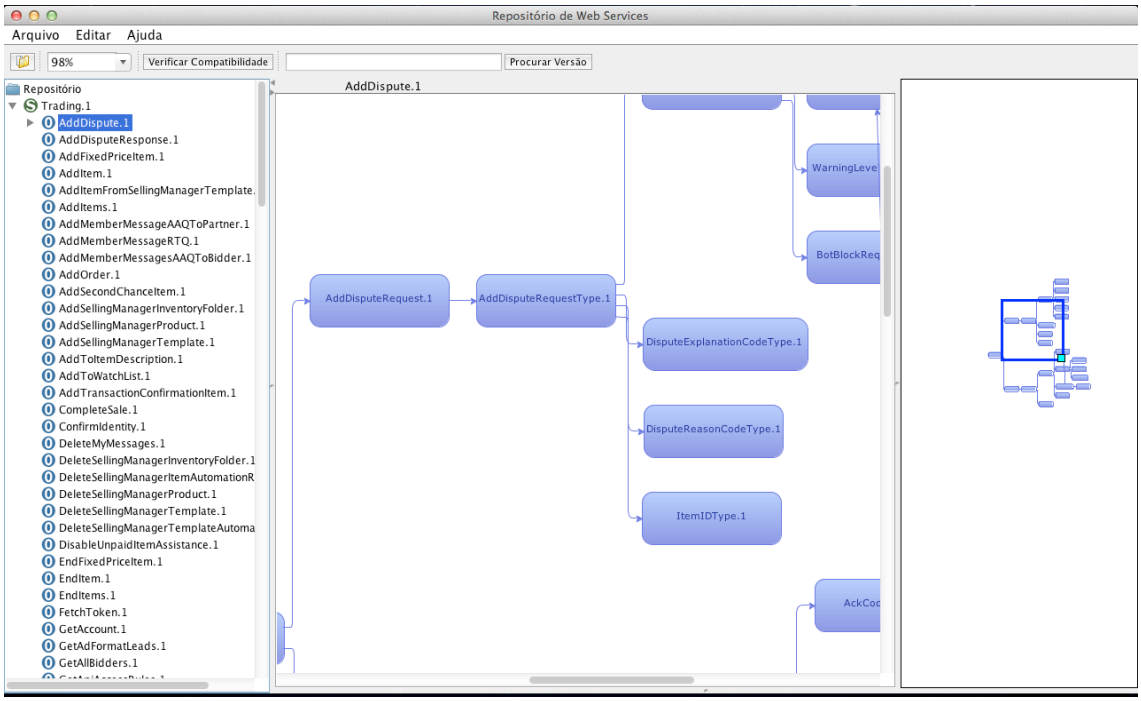


Figura 4.7: Exemplo de utilização da funcionalidade de esboço

A funcionalidade de *zoom* pode ser acessada pela barra de ferramentas, utilizando-se a classe JComboBox disponibilizada pelo Swing. Esta classe permite visualizar dados no formato de uma lista, permitindo ao usuário selecionar um item. Outra forma de aplicar o *zoom* no grafo é utilizar o *scroll* do mouse sobre o esboço do mesmo.

4.5 Exibição dos Pontos de Compatibilidade/Incompatibilidade das Versões

A funcionalidade principal para verificação da evolução das versões armazenadas no repositório é a verificação de compatibilidade. Para verificar a compatibilidade das versões armazenadas, o usuário precisa selecionar duas versões de *features* do mesmo tipo. Existem três maneiras de efetuar está seleção: a primeira é utilizando um botão disponibilizado na barra de ferramentas, a segunda é acessada utilizando o botão secundário do mouse sobre o nodo da árvore, e escolhendo a opção "Checar Compatibilidade...", e a terceira é selecionando no menu "Editar" a opção "Checar Compatibilidade" (Figura 4.7).

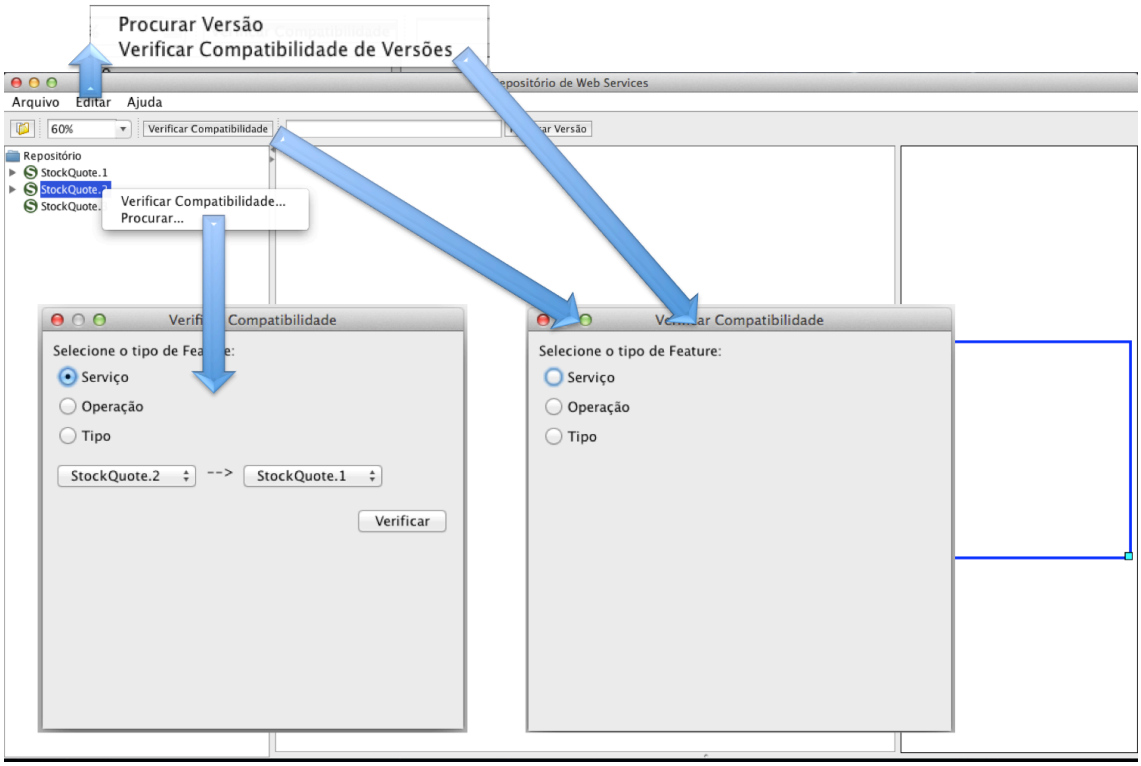


Figura 4.8: Tipos de verificação de compatibilidade

A janela de controle para seleção das versões a serem comparadas possui primeiramente um filtro pelo tipo da *feature* (serviço, operação, ou tipo) e logo em seguida dois JComboBox são apresentados. Cada JComboBox apresenta a lista das *features* existentes no repositório, correspondentes ao tipo selecionado pelo usuário, sendo entre eles desenhada uma seta responsável por indicar o sentido da verificação.

Note-se que no exemplo da Figura 4.7, quando a verificação de compatibilidade é acessada a partir da árvore de versões, a versão correspondente ao nodo clicado pelo usuário é previamente selecionada no JComboBox do lado esquerdo, além do seu respectivo tipo. Nas demais possibilidades de acesso a funcionalidade, cabe ao usuário determinar as versões das *features* a serem comparadas, bem como seu respectivo tipo. Destaca-se que nesses casos, a seleção das versões somente estará disponível, após a seleção do tipo das mesmas.

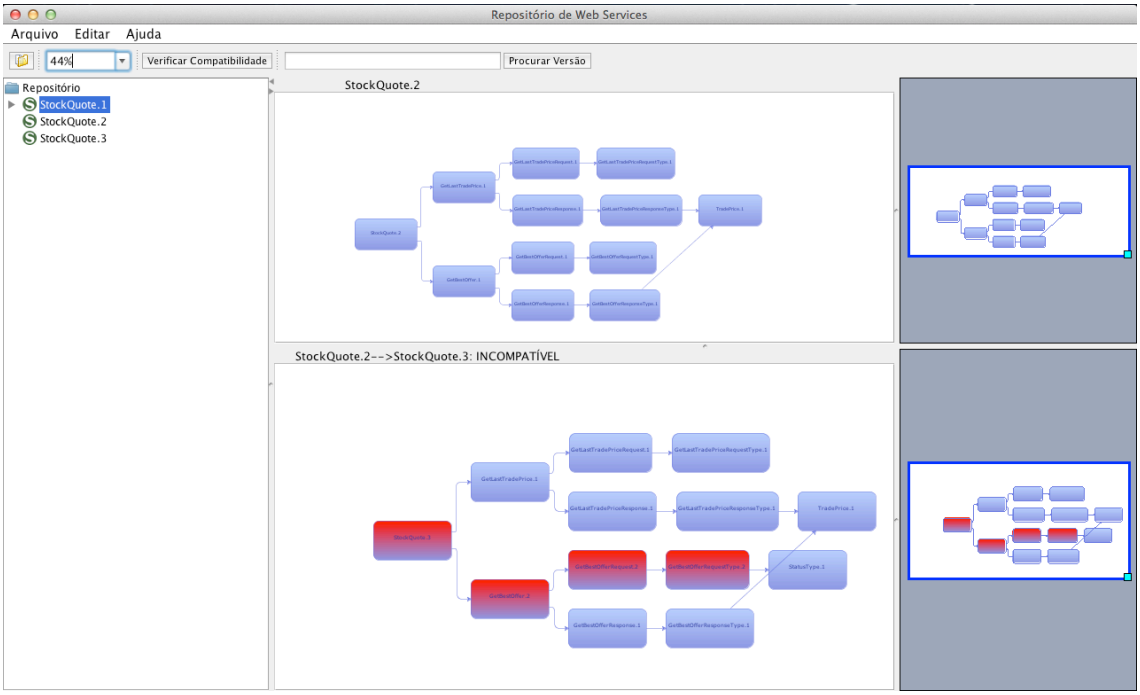


Figura 4.9: Resultado da verificação de compatibilidade

O resultado da verificação de compatibilidade entre as versões StockQuote.2 e StockQuote.3, no sentido StockQuote.2 → StockQuote.3, é representado na Figura 4.8. O grafo de dependência referente a StockQuote.2 é apresentado acima, e abaixo é apresentado o resultado da verificação de compatibilidade. Este resultado é composto por um veredito, no caso "StockQuote.2 → StockQuote.3: INCOMPATÍVEL", além do grafo de dependência referente a StockQuote.3 com os nodos incompatíveis marcados em vermelho.

No caso acima, a incompatibilidade se dá baseada no caso 3 da Tabela 2.1, em função da adição de um tipo a um tipo já existente. É possível observar que, por dependência, dois tipos, uma operação e o próprio serviço foram versionados.

4.6 Considerações Finais

A ferramenta desenvolvida como demonstrado nas seções anteriores, permite a exibição das *features* armazenadas segundo o modelo de versionamento descrito no Capítulo 2. A análise das versões pode ser efetuada de forma intuitiva utilizando o grafo de dependência, e ainda com o auxílio do esboço do grafo, mesmo com grande número de nodos, é possível ter uma experiência produtiva. A ferramenta de busca permite pesquisar todas as versões de *features* armazenadas, ou apenas em escopos predefinidos, tendo os resultados apresentados na árvore de versões.

O acesso as versões se dá de maneira ágil e otimizada quando utilizada a árvore de versões, sendo possível acessar as funcionalidades de busca e de verificação de compatibilidade diretamente na mesma. A compatibilidade é avaliada em pares e os pontos de compatibilidade/incompatibilidade são exibidos permitindo verificar as zonas de impacto das mudanças, em qualquer etapa do ciclo de vida do serviço.

A ferramenta desenvolvida carece de alternativas na exibição dos resultados da busca, que ainda restringe-se a árvore de versões, e na exibição dos resultados da verificação de compatibilidade, que utiliza apenas a coloração de nodos para ressaltar as versões de *features* incompatíveis. Quanto ao carregamento de novas versões, sendo disponibilizada a agregação de apenas um único arquivo por vez, acarreta retrabalho quando o usuário tem interesse em agregar múltiplas versões de uma única vez.

5 ANÁLISE DE PERFORMACE DA INTERFACE

Com o intuito de verificar o comportamento da interface diante da manipulação de dados reais, utilizou-se *web services* provenientes do eBay[13] para efetuar uma série de experimentos. Os resultados serão apresentados individualmente, considerando as funcionalidades da interface descritas no capítulo anterior, tendo-se aprofundado esta descrição quando necessário.

Primeiramente, analisou-se o universo de dados disponíveis para a avaliação, pois nossas conclusões estarão restritas à eles. A análise se deu considerando dois cenários: o número de *features* envolvidas em uma interação e o número de *features* armazenadas no repositório no momento da interação. Em seguida, a partir destes cenários, o tempo de resposta das interações foi discutido, diferenciando-se, quando possível, o tempo referente a recuperação dos dados do tempo referente a exibição dos dados. Finalmente foram discutidos os resultados obtidos, bem como o modelo de armazenamento adotado.

5.1 Universo de Dados Utilizados

Para verificar o comportamento da interface, utilizou-se interfaces de descrição de serviço provenientes do eBay. Cada serviço foi nomeado de acordo com a ordem de agregação no repositório, que respeitou a ordem de liberação dos WSDL pelo eBay. A composição dos mesmos, quando considerado número de *features* é exibida na Figura 5.1.

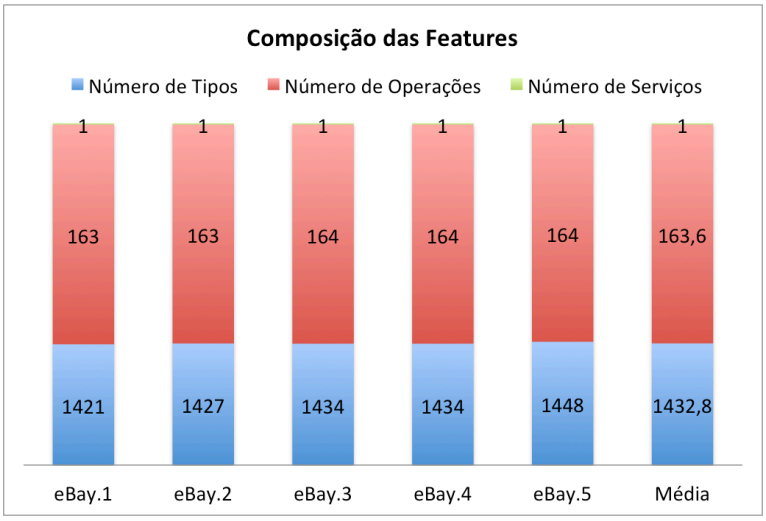


Figura 5.1: Composição dos serviços

[13] http://global.ebay.com/

O gráfico demonstra o número de *features* referentes a tipos, a operações e a serviços, de acordo com o modelo de armazenamento descrito no Capítulo 2, para cada interface de descrição de serviço a ser versionada. Observa-se uma uniformidade na constituição dos dados, tendo em vista que o número de operações e tipos não foge muito da média em nenhum WSDL liberado.

5.2 Agregação de Versões ao Repositório

Como descrito no Capítulo 4, o usuário tem a possibilidade de agregar novas versões ao repositório, selecionando um WSDL. A agregação se dá em quatro passos:

- Conversão do WSDL para o modelo de versionamento proposto. O resultado desta conversão é denominado Ghost.
- Detecção de diferenças entre o Ghost e as versões armazenadas no repositório. Resultando em um novo arquivo no modelo proposto, denominado Diff, contendo as diferenças encontradas, ou seja exatamente o que deve ser adicionado ao repositório.
- Agregação do Diff ao repositório.
- Atualização da árvore de exibição de serviços.

A Figura 5.2 exemplifica de forma visual estes passos.

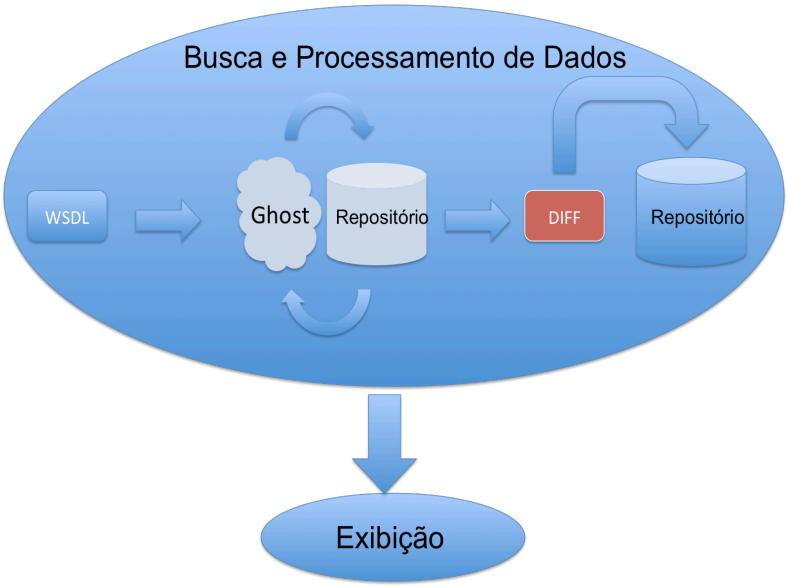


Figura 5.2: Passos da agregação de versões

Neste caso, denominou-se os três primeiros passos anteriormente citados, simplificadamente como Busca e o último passo denominou-se como Exibição. Os resultados da análise do tempo de processamento da Busca e da Exibição, para cada WSDL agregado, pode ser observado na Figura 5.3 e Figura 5.4 respectivamente.

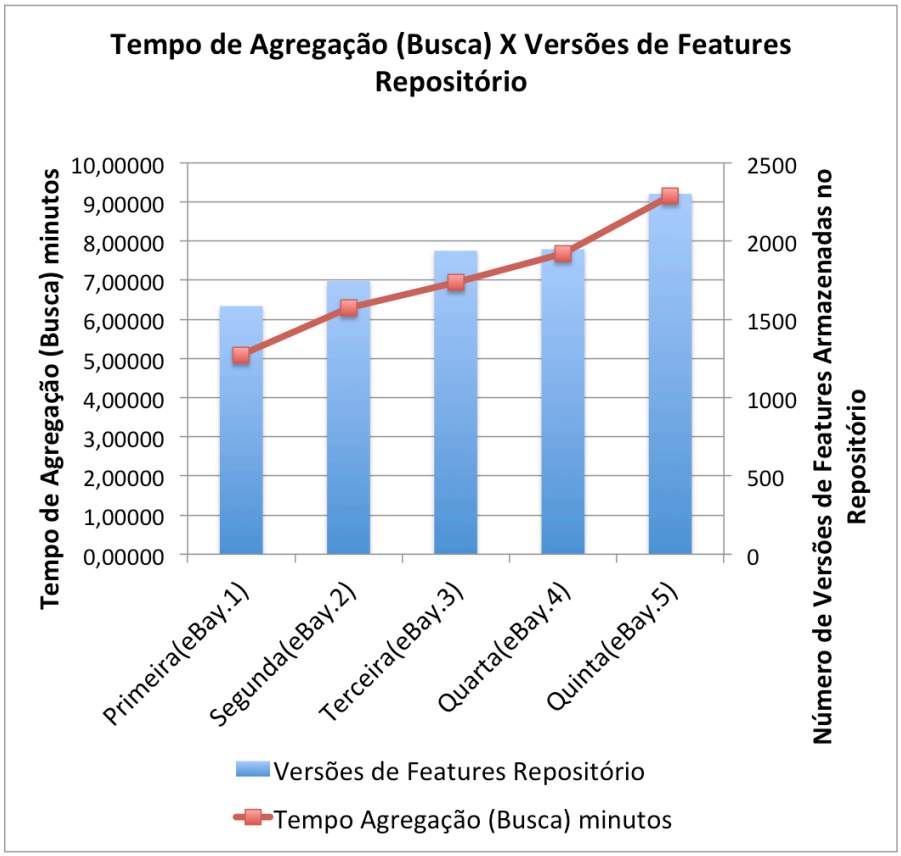


Figura 5.3: Gráfico referente ao tempo de agregação (Busca)

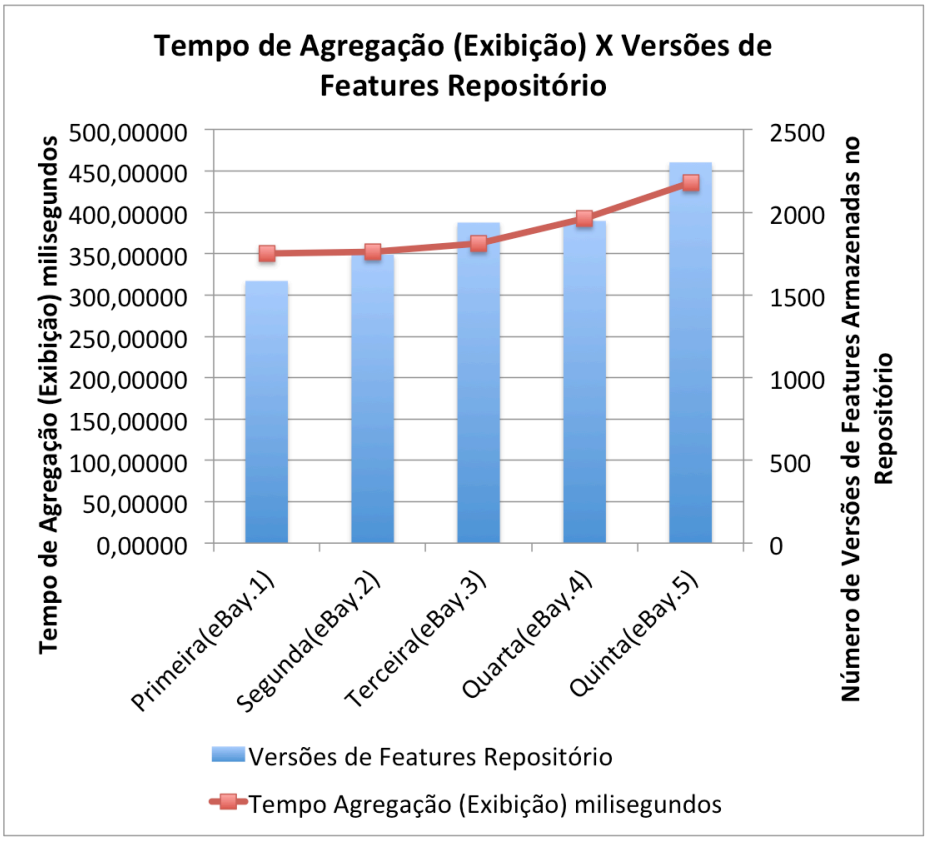


Figura 5.4: Gráfico referente ao tempo de agregação (Exibição)

Na Figura 5.3 em vermelho, temos o crescimento do tempo - em minutos - relativo a Busca, decorrido ao agregar as interfaces de descrição de serviços ao repositório, e em azul o crescimento do número de versões de *features* no repositório após cada agregação. O mesmo ocorre na Figura 5.4, no entanto o tempo de agregação é dado em milissegundos e é relativo a Exibição.

O tempo de processamento referente a Busca dos dados cresceu aproximadamente 1 minuto por agregação, de acordo com o número de versões armazenadas no repositório. Este crescimento se deve principalmente a geração do Diff, pois ,neste passo, é necessário percorrer todo repositório em busca de diferenças. Portanto, o crescimento dos dados armazenados tem impacto direto no tempo de processamento.

Quanto ao tempo de processamento referente a Exibição dos dados, vemos que não há grande variação, pois simplesmente reconstrói-se a árvore, exibindo seu primeiro nível referente aos serviços, sendo necessário simplesmente buscar as versões de serviços do repositório.

5.3 Acesso as Versões do Repositório

Quanto a funcionalidade de acesso as versões do repositório, a análise de *performance* foi feita sempre baseada na expansão dos nodos referentes as operações da versão eBay.1. Desta forma, o número de nodos envolvidos na interação é constante, sendo considerada somente a variação do número de versões armazenadas no repositório.

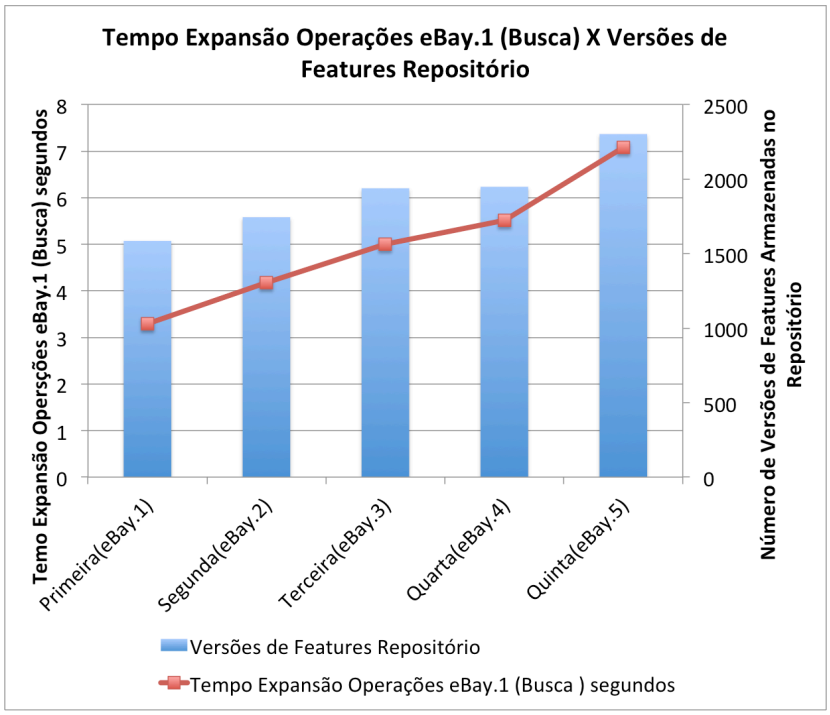


Figura 5.5: Gráfico referente ao tempo expansão de operações da versão de serviço eBay.1 (Busca)

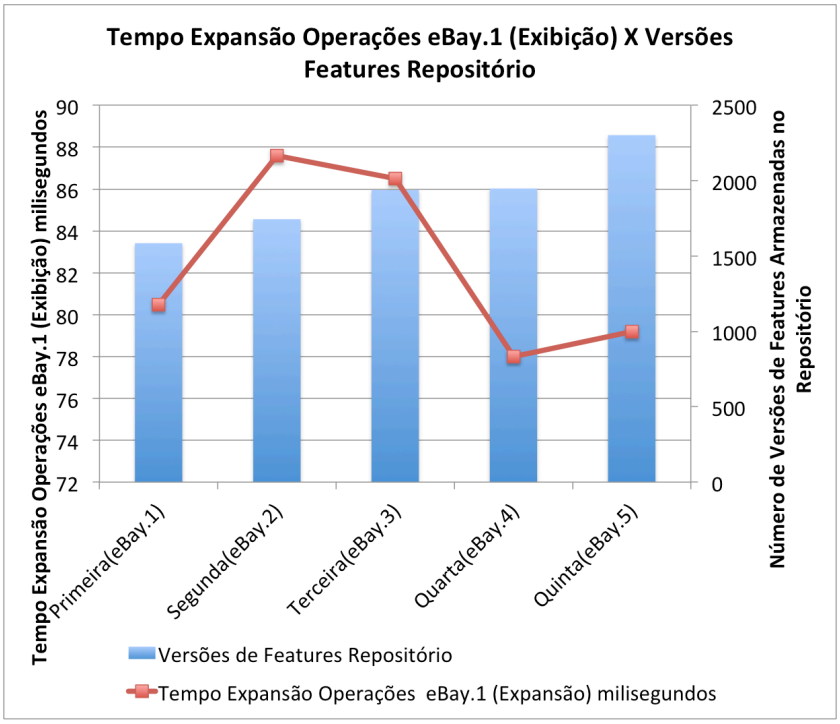


Figura 5.6: Gráfico referente ao tempo expansão de operações da versão de serviço eBay.1 (Exibição)

Na Figura 5.5 é possível visualizar os resultados da análise de *performance* na expansão das operações da versão de serviço eBay.1, no que tange o que denominou-se a Busca dos dados. Na Figura 5.6 os resultados apresentados são referentes ao que denominou-se a Exibição dos dados. Em ambos os casos, Busca e Exibição, temos em vermelho o tempo de expansão, em segundos e milissegundos, respectivamente, e em azul o número de versões de *features* armazenadas após cada agregação.

A Busca dos dados corresponde a pesquisa das operações no repositório dependentes da versão de serviço eBay.1. Como esta execução é vinculada diretamente ao repositório, o crescimento do número de versões armazenadas tem um impacto direto no tempo de resposta. A Exibição dos dados corresponde a adição dos nodos buscados na árvore de exibição, tendo em vista que esta execução se dá independente do repositório, não teve crescimento vinculado ao número de versões armazenadas. Inclusive, é possível observar uma variação negativa neste tempo, mesmo após o aumento do número de versões armazenadas.

5.3.1 Pesquisa de Versões na Árvore

A avaliação de *performance*, referente a pesquisa de versões, foi feita considerando sempre o escopo referente a versão de operação AddDispute.1, ou seja, apenas os nodos filhos da mesma foram considerados na busca. Ainda, a String utilizada para busca foi sempre “Add”. Estas duas variáveis foram mantidas constantes, de forma a manter os nodos envolvidos na interação inalterados, tendo apenas considerada a variação de versões armazenadas no repositório.

Na avaliação desta funcionalidade, fez-se uma separação entre o tempo de recuperação das versões que possuem nome iniciado pela *String* “Add” e a exibição destas versões na árvore da interface, denominados respectivamente, Busca e Exibição.

Os resultados referentes a Busca podem ser visualizados na Figura 5.7 e os resultados referentes a Exibição podem ser visualizados na Figura 5.8. Para ambas, Busca e Exibição, em vermelho temos o tempo decorrido, respectivamente em milissegundos e segundos, e em azul o número de versões de *features* após cada agregação.

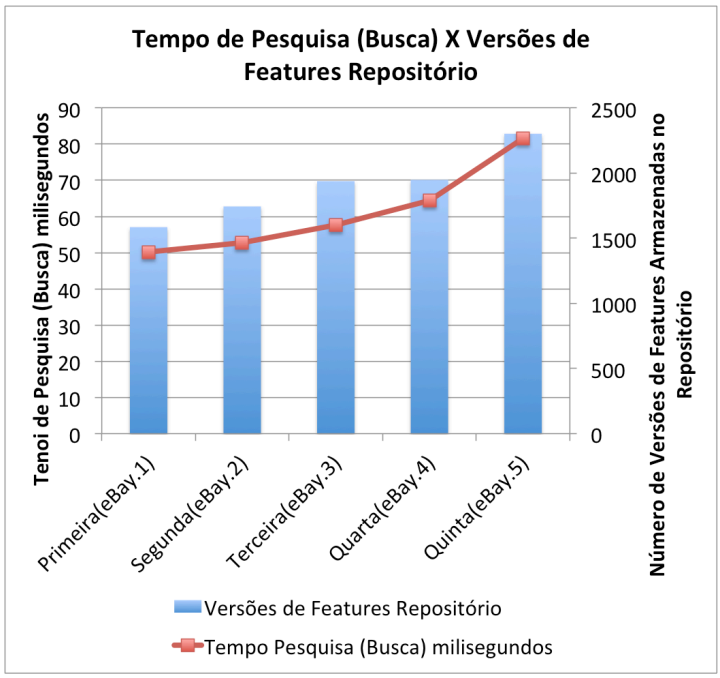


Figura 5.7: Gráfico referente ao tempo pesquisa pela *String* "Add" no escopo da versão AddDispute.1 (Busca)

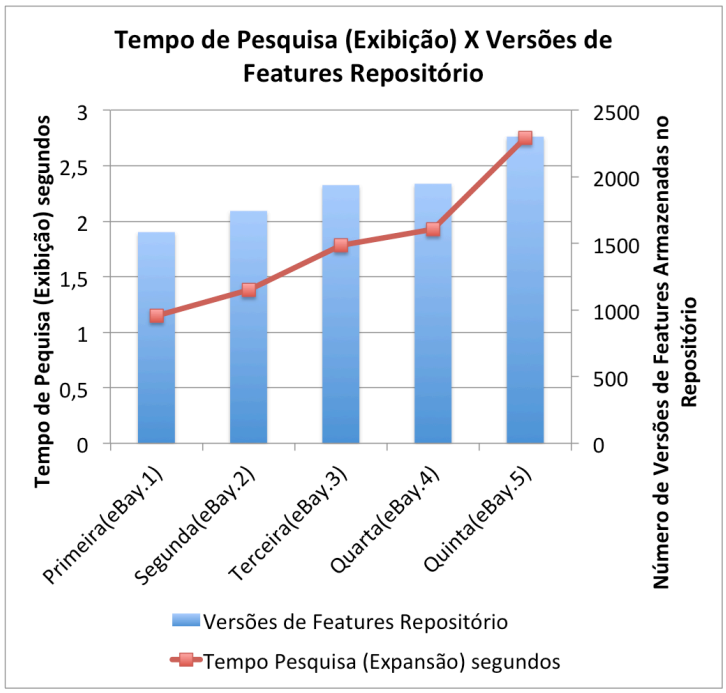


Figura 5.8: Gráfico referente ao tempo pesquisa pela *String* "Add" no escopo da versão AddDispute.1 (Exibição)

Diferentemente das análises anteriores, na avaliação do tempo de pesquisa, a Exibição é mais demorada que a Busca. A Busca precisa percorrer todas as versões existentes no repositório, e a Exibição, precisa expandir a árvore dinamicamente com o intuito de percorrê-la e marcar as versões retornadas. Como visto anteriormente, a expansão consiste igualmente em uma Busca e uma Exibição. Por isso essas sucessivas expansões consistem em interações constantes com o repositório, aumentando o tempo de resposta da Exibição dos resultados da pesquisa. Este relacionamento é exemplificado na Figura 5.9.

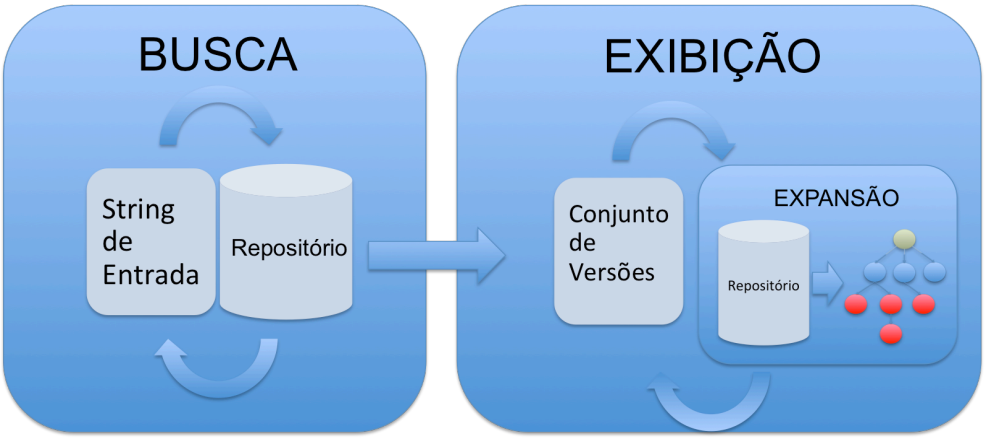


Figura 5.9: Funcionamento pesquisa por String

5.4 Exibição Utilizando Grafo de Dependência

A análise de performance da exibição do grafo de dependência possui três passos: recuperação do grafo do repositório, conversão para o formato da API de exibição e aplicação do *layout* de disposição dos nodos. Os resultados destes três passos podem ser observados, respectivamente, na Figura 5.10, na Figura 5.11 e na Figura 5.12. Em todos os gráficos temos em vermelho o tempo decorrido em segundos e em azul o número de versões de *features* armazenadas no repositório, após cada agregação.

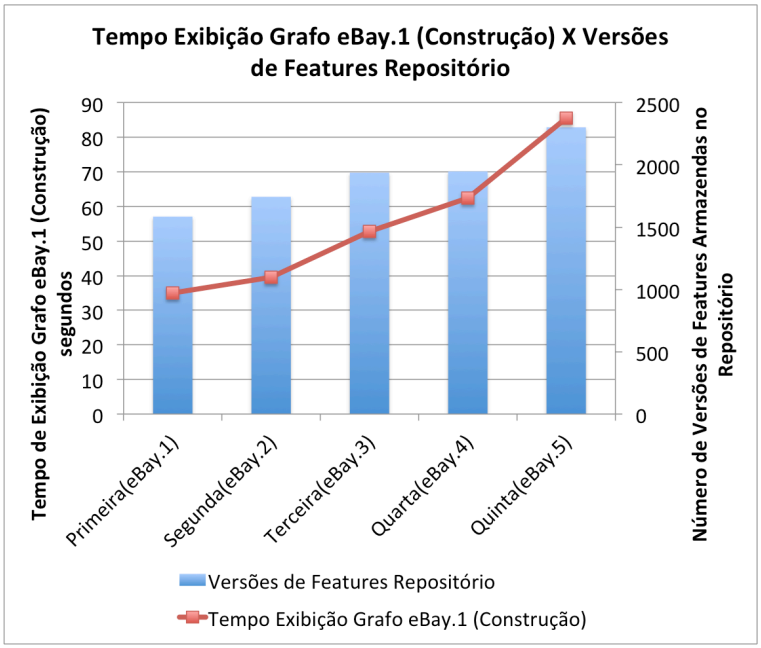


Figura 5.10: Tempo de exibição grafo eBay.1 (Construção)

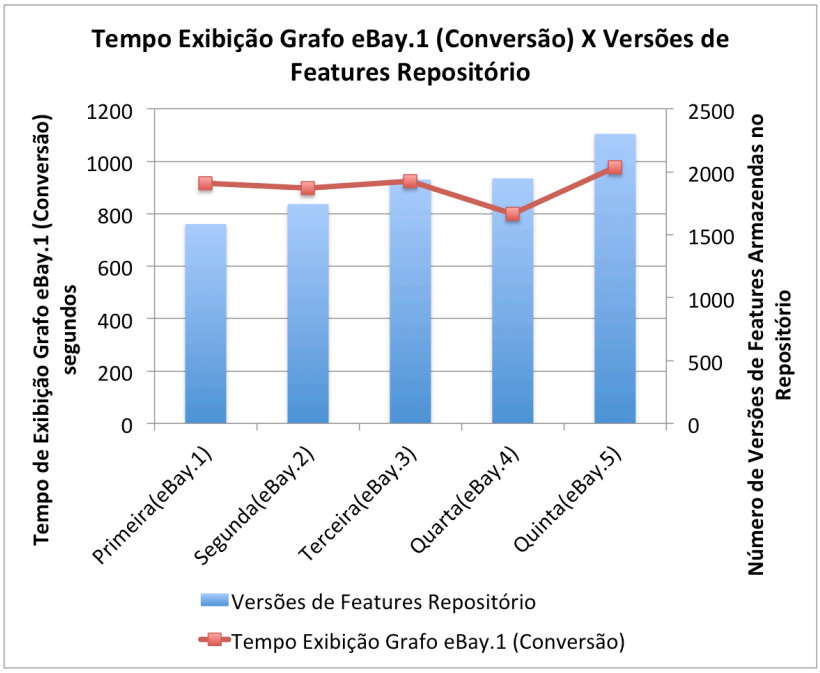


Figura 5.11: Tempo de exibição grafo eBay.1 (Conversão)

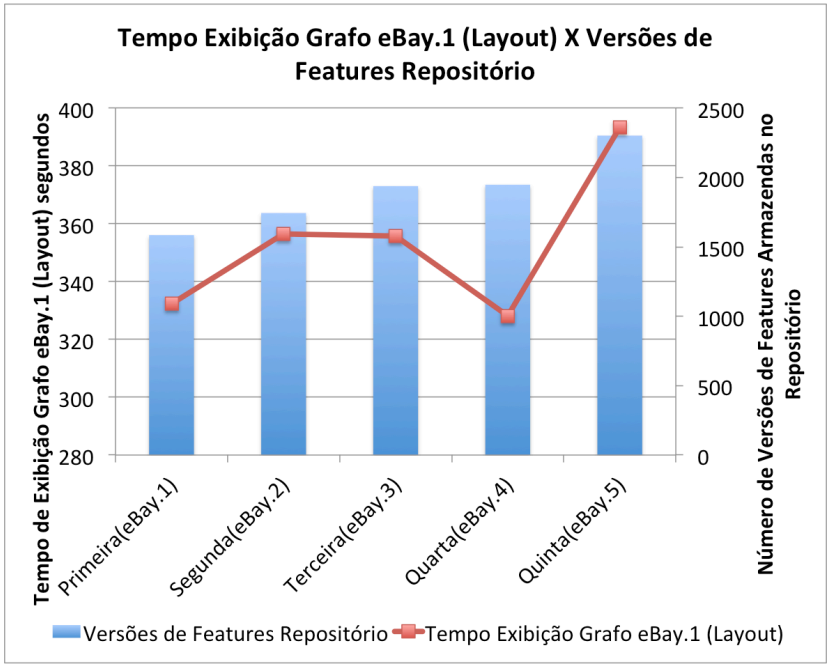


Figura 5.12: Tempo de exibição grafo eBay.1 (*Layout*)

Observando os resultados pode-se concluir que os passos de Conversão e de *Layout* oscilaram seus tempos de resposta, pois independem das agregações e do consequente aumento do número de versões armazenadas. A construção do grafo tem maior peso no tempo de resposta, e está diretamente ligada ao aumento do número de versões armazenadas, tendo em vista que o grafo é construído acessando diversas vezes o repositório.

5.5 Exibição dos Pontos de Compatibilidade/Incompatibilidade das Versões

A análise da exibição dos pontos de compatibilidade/incompatibilidade das versões foi dividida em três partes. A construção dos grafos das versões a serem comparadas, denominada Construção. A análise de compatibilidade, denominada Análise. A exibição propriamente dita dos grafos, constituída pela conversão dos grafos para a API de exibição e a aplicação do *layout*, denominada Exibição. Os resultados podem ser visualizados na Figura 5.13, na Figura 5.14 e na Figura 5.15, na forma de gráficos, nos quais temos em vermelho o tempo decorrido em segundos, excetuando-se a Análise que está em milissegundos, e em azul o número de versões de *features* armazenadas após cada agregação.

De maneira a fixar o número de *features* envolvidas na interação, todas as verificações de compatibilidade foram feitas entre a versão de *feature* eBay.1, com relação a ela mesma (eBay.1 -> eBay.1). Naturalmente o veredito de compatibilidade desta análise é compatível, o que não tem influência na análise de *performance*, pois o número de nodos visitados se mantem igual.

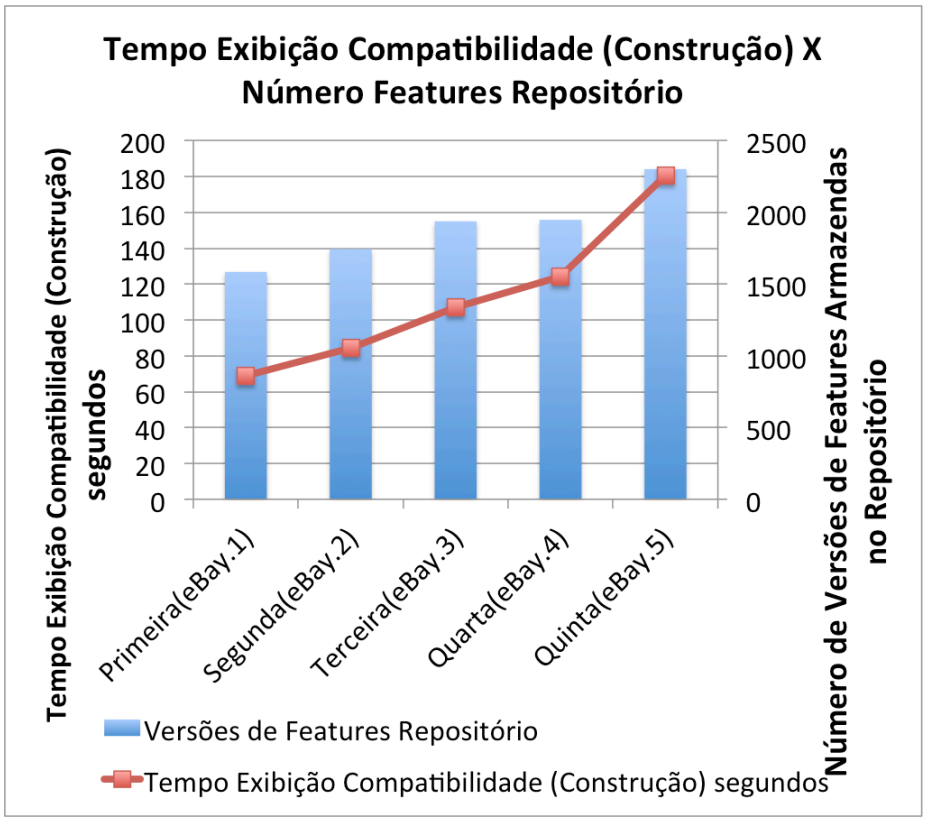


Figura 5.13: Tempo de exibição da compatibilidade eBay.1 -> eBay.1 (Construção)

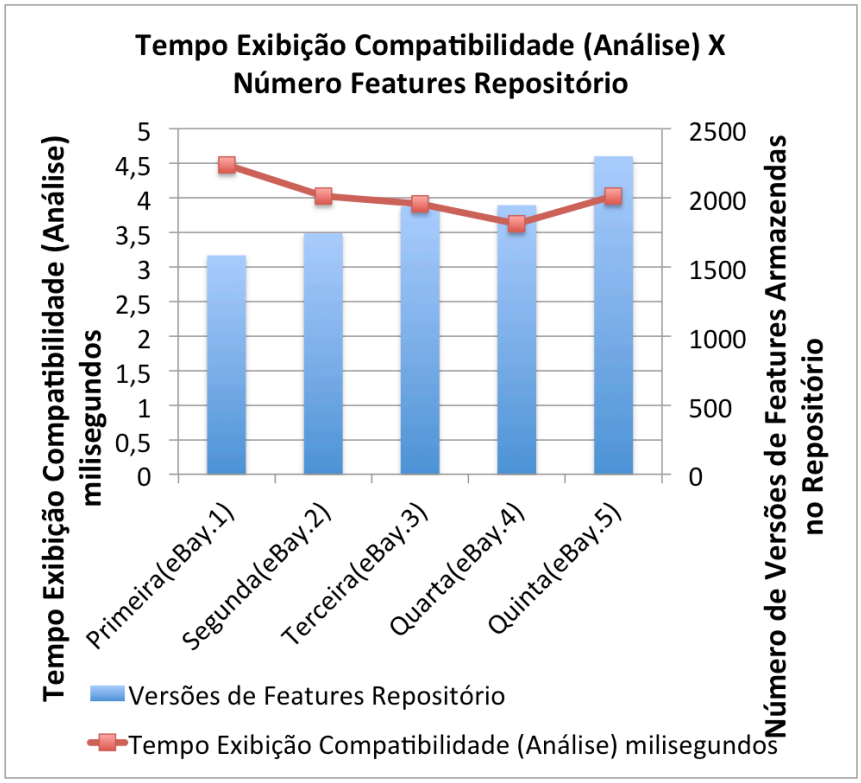


Figura 5.14: Tempo de exibição da compatibilidade eBay.1 -> eBay.1 (Análise)

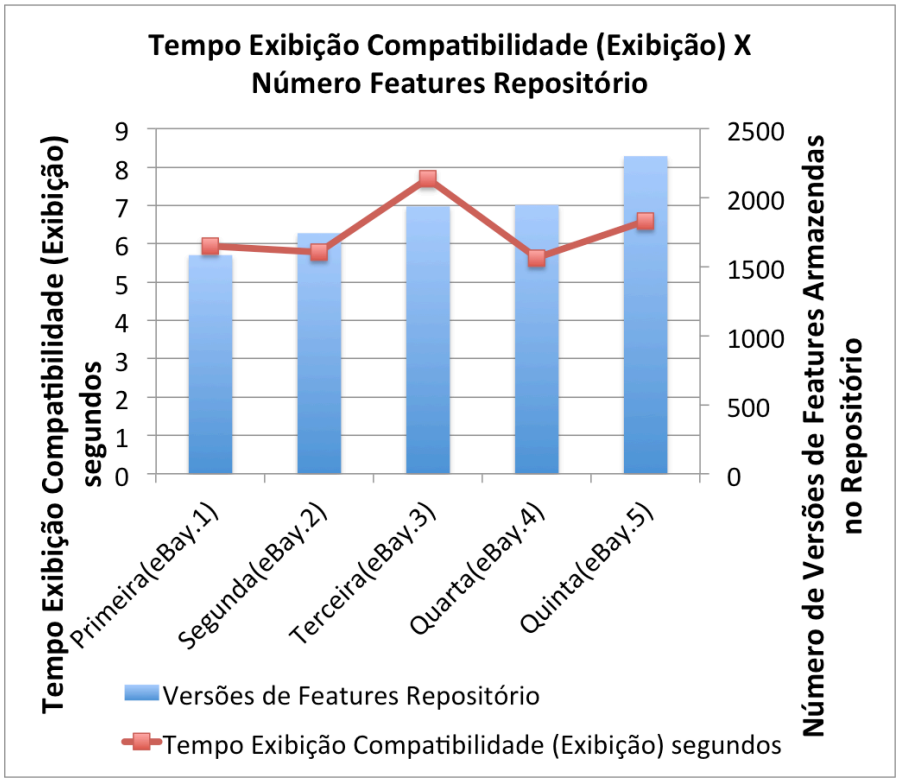


Figura 5.15: Tempo de exibição da compatibilidade eBay.1 -> eBay.1 (Exibição)

É possível concluir a partir dos gráficos que os tempos de Análise e de Exibição oscilam, pois ambos independem do acesso ao repositório. Enquanto, o tempo de Construção, como citado anteriormente, precisa interagir com o repositório, por isso é possível perceber seu valor crescer juntamente com o número de versões armazenadas.

5.6 Considerações Acerca dos Resultados Obtidos

Na funcionalidade de agregação o maior tempo de resposta foi obtido na última agregação, referente a versão de serviço eBay.5, chegando a 9,58924 minutos. Sendo apenas 436,00400 milisegundos gastos com a exibição na árvore de versões e 9,15324 minutos gastos com a agregação propriamente dita.

Na funcionalidade de acesso as versões, o tempo de expansão máximo foi de 7,159342 segundos, sendo 79,183 milisegundos referentes a exibição dos dados na árvore e 7,080159 segundos referente a busca das 169 versões de operações as quais são dependentes da versão de serviço eBay.1.

A funcionalidade de busca foi o único caso avaliado no qual a busca dos dados foi mais rápida do que a apresentação dos mesmos. Este caso difere dos outros, pois na apresentação ainda há interação com o repositório, pois ela é composta por seguidas expansões dos níveis da árvore. No seu pior caso o tempo de resposta final chegou a 2,834017 segundos, sendo composto por 81,516 milisegundos de recuperação de versões e 2,752501 segundos de exibição das mesmas na árvore.

Na funcionalidade de exibição do grafo de dependência, o tempo de resposta mais alto foi de 86,946372 segundos, sendo composto por 85,57479 segundos de construção do grafo de dependência, 978,367 milisegundos para conversão para a API de exibição e 393,215 milisegundos para aplicação do *layout* no grafo.

Na funcionalidade de exibição dos pontos de compatibilidade/incompatibilidade no pior caso foram gastos 3,011042967 minutos para a construção dos dois grafos dependência a serem comparados, 4,028 milisegundos para a avalição de compatibilidade e 6,597412 segundos para a exibição propriamente dita dos grafos.

5.6.1 Tecnologia de Armazenamento do Repositório

As versões do repositório são armazenadas em um arquivo XML único. A busca, em decorrência desta escolha de projeto, é feita utilizando XPath, uma linguagem de busca em documentos XML. A grande problemática relacionada a utilização de um arquivo XML para persistência dos dados é a escalabilidade. Isto pode ser observado ao analisarmos o tempo de acesso aos dados, bem como tempo de conversão dos dados para um grafo de dependência, o que pode ser comprovado pelo crescimento dos tempos de agregação e pelos altos tempos de construção dos grafos, seja na exibição simples, seja na exibição dos pontos de compatibilidade.

Não está no escopo deste trabalho avaliar se a utilização de um banco de dados relacional, ou um banco de dados de grafos, poderia ter um desempenho melhor com o crescimento dos dados armazenados. No entanto, a sofisticação dos sistemas de gerência de banco de dados, levam a crer que os tempos de retorno dos dados seriam menos afetados pelo volume de dados armazenados, se comparados com a utilização de um arquivo XML único. A opção por um banco de dados de grafos, poderia, ainda, eliminar o tempo de construção do grafo, nas funcionalidades em que é necessária a exibição dos grafos de dependência.

6 CONCLUSÃO

Estre trabalho implementou e descreveu uma interface de exibição de versões de *web services*, que possibilita a visualização, pesquisa e análise de compatibilidade de versões. A análise de *performance* demonstrou que, dentro do universo de dados utilizados, o tempo de resposta na exibição do grafo e da expansão da árvore de versões, desconsiderando-se o tempo de recuperação dos dados, não compromete a experiência do usuário na utilização do sistema.

Quanto a escalabilidade do sistema de armazenamento, entende-se que há a necessidade de uma remodelagem, pois numa perspectiva de seguidos lançamentos de versões, e a consequente agregação das mesmas no repositório, a lentidão na recuperação dos dados prejudicaria a experiência do usuário na utilização do sistema.

A interface obtida fica como prova de conceito, e um ponto de partida para integração com o novo modelo de armazenamento a ser desenvolvido, tendo-se provado que as tecnologias adotadas para o seu desenvolvimento, permitem um bom desempenho, no que tange o tempo de resposta na utilização de suas funcionalidades.

Para trabalhos futuros, vislumbra-se, além da necessidade de remodelagem do modelo de armazenamento, a exibição dos resultados obtidos na análise de compatibilidade entre versões, bem como na pesquisa por versões, utilizando uma lista. Desta forma, usuário poderia ter uma nova experiência na visualização dos resultados, pois esta apresentação apesar de menos visual, quando comparada com a exibição via nodos coloridos do grafo, ou marcações nos nodos da árvore, permitiria uma avaliação mais prática dos resultados.

REFERÊNCIAS

YAMASHITA, M.; BECKER K.; GALANTE R. **A Flexible Approach for Accessing Service Compatibility at Feature Level**. Simpósio Brasileiro de Banco de Dados, Florianópolis, p. 105-112, Out. 2011.

BECKER, K. et al. **Automatic Determination of Compatibility in Evolving Services**. International Journal of Web Service Research, Hershey, v.8, n.1, p. 21-40, Jan./Mar. 2011.

ANDRIKOPOULOS, V.; BENBERNOU, S.; PAPAZOGLOU, M. **On the Evolution of Services**, IEEE Transactions on Software Engineering, p. 609-628, Mar. 2011.

FOAKEFS, M. et al. **An Empirical Study on Web Service Evolution**, IEEE International Conference on Web Services, p. 49-56, Jul. 2011.

ZOU, Z. et al. **On Synchronizing with Web Service Evolution**, IEEE International Conference on Web Services, p. 329-336, Sep. 2008.

FANG, R. et al. **A Version-aware Approach for Web Service Directory**, IEEE International Conference on Web Services, p. 406-413, Jul. 2007.

BROWN, K.; ELLIS, M. **Best Practices for Web Service Versioning**, Jan. 2004. Disponível em: < http://www.ibm.com/developerworks/webservices/library/ws-version/ >. Acesso em: jun. 2012.

ROBINSON, Matthew; VOROBIEV, Pavel. **Swing**. 2nd ed. Greenwich: Manning, 2003.

GUDURU, S. R. **Graph Data Conversion and Tree Visualization**. 2011. 53 f. Dissertação (Degree of Master of Science) – Computer Science Department, NDSU, Fargo.

WIKIPEDIA WEB SERVICE. Disponível em <http://pt.wikipedia.org/wiki/Web_service>. Acesso em: Jun. 2012.

JGRAPH API. Disponível em: <http://www.jgraph.com/jgraph.html> Acesso em: Jun. 2012.

ORACLE SWING DOCUMENTATION. Disponível em: < http://docs.oracle.com/javase/1.4.2/docs/api/javax/swing/package-summary.html> Acesso em: Jun. 2012.