3
Modelagem de Interfaces RIA

A abordagem do método SHDM divide a modelagem de interfaces em dois níveis de abstração: o primeiro nível, representado pela Ontologia de *Widgets* Abstratos, independente de plataforma, e a Ontologia de *Widgets* Concretos que mapeia a descrição abstrata para a tecnologia HTML + CSS. A proposta de extensão para os formalismos adotados pelo SHDM visa conferir-lhes a expressividade necessária para modelar os controles de interface pertencentes à geração 2.0, preservando a arquitetura em dois níveis de abstração. No nível concreto, a tecnologia pressuposta será o Javascript.

Entretanto, nossa abordagem ao projeto de aplicações Web ricas leva em consideração apenas a modelagem da apresentação, não contemplando outros aspectos tais como a modelagem distribuída (entre cliente e servidor) dos dados e da lógica de negócios, que também caracterizam a tecnologia RIA. [5]. Além disso, a representação de objetos multimídia (áudio e vídeo) e o suporte à internacionalização e localização também estão fora do escopo do nosso estudo.

3.1. Requisitos para a Modelagem de Interfaces

O modelo de projeto *Abstract Data View* foi criado para especificar clara e formalmente a separação entre a interface do usuário e os componentes de aplicação em um sistema de software, e para oferecer um método de projeto independente de implementação, aumentando o grau de reuso tanto dos componentes de lógica de negócio quanto de interface. [36]

As interações do usuário com os componentes de aplicação, chamados de *Abstract Data Objects* (ADOs), são modeladas através dos *Abstract Data Views* (ADVs) correspondentes, os quais são capazes de expressar tanto as propriedades de percepção dos ADOs, quanto os eventos com os quais eles podem lidar. Além disso, os ADVs admitem mecanismos de composição e herança. [37]

O formalismo dos ADVs é utilizado pelo método OOHDM, na fase de projeto de interface abstrata. [38] Como parte do método SHDM, Moura [2] propõe: uma Ontologia de *Widgets* Abstratos, para a especificação de interfaces abstratas; uma Ontologia de *Widgets* Concretos, que descreve elementos de interface disponíveis nos

ambientes de implementação, por exemplo, os elementos de uma página HTML; e um conjunto de regras para o mapeamento entre as duas ontologias. Cada elemento abstrato será mapeado: (a) para um elemento de interface concreto, que define a sua apresentação; (b) para um elemento navegacional, que provê o seu conteúdo. As regras de mapeamento entre elementos abstratos e concretos permitem ao projetista de uma aplicação gerar automaticamente uma interface concreta a partir de um modelo de interface abstrata.

Este formalismo é suficiente para representar a *estrutura* e o *formato* das apresentações, traduzíveis em documentos HTML e folhas de estilo CSS (*Cascading Style Sheets*). No contexto da Web 2.0, entretanto, a experiência de interação passa a rivalizar com a dos aplicativos *desktop.* A sofisticação dos elementos de interface na Web, agora dotados de comportamento e capazes de reagir eventos, cria a necessidade de um vocabulário para definir o *comportamento* da interface, traduzível, por exemplo, em códigos Javascript.

Considere também a categoria das aplicações onde é possível tirar partido do paralelismo no processamento de uma requisição do usuário. Este é o caso dos sites que oferecem acesso a diversos serviços Web de forma centralizada. Um exemplo típico seria uma aplicação Web cujo objetivo é ajudar os usuários a comparar preços de passagens aéreas, como o site Alibabuy.com. Seus requisitos funcionais incluem capturar os parâmetros de pesquisa informados pelos usuários (cidades de origem e destino, data de partida e retorno, número de passageiros, etc.) e, em seguida, submeter consultas a sites de agências de viagens, esperar pelas respostas, ordená-las e exibi-las para o usuário. Esta aplicação típica tem com característica intrínseca o paralelismo das consultas aos serviços Web das empresas parceiras. Aqui, o designer da aplicação pode optar por a interface ser atualizada de uma só vez, após o retorno de todas as chamadas aos serviços Web (“serialização” das chamadas), ou por oferecer ao usuário uma experiência de interatividade mais rica: atualizar a interface incrementalmente, no retorno de cada chamada.

A proposta de extensão do trabalho de Moura [2] tem como objetivo suprir o método SHDM de uma linguagem capaz de modelar o funcionamento das interfaces neste último tipo de aplicação, onde não somente estão presentes controles sofisticados, mas também onde se faz desejável comunicar aos usuários eventos ocorridos na camada de Modelo. Chamamos este produto da nossa pesquisa de **Linguagem ou Ontologia de Descrição de Interfaces Ricas**[41].

[41] A Ontologia de Descrição de Interfaces Ricas foi construída com o auxílio do editor de ontologias Protégé 3.4 rc2.

3.2. Modelagem de Widgets Abstratos e Concretos

A Ontologia de Descrição de Interfaces Ricas tem por objetivo descrever de que forma os *Abstract Data Objects* (ADOs) serão apresentados ao usuário, detalhando quais elementos perceptíveis (ADVs) estarão disponíveis. Os conceitos da ontologia em seu nível abstrato são[42]:

AbstractInterfaceElement: esta classe representa todos os possíveis elementos que compõem a interface abstrata e é uma generalização das seguintes subclasses:

CompositeInterfaceElement: representa uma composição de elementos abstratos;

ElementExhibitor: representa elementos que exibem algum tipo de conteúdo, como, por exemplo, um texto ou uma imagem;

IndefiniteVariable: representa elementos que permitem a livre entrada de dados, por exemplo, uma caixa de texto em um formulário;

PredefinedVariable: representa elementos que permitem a seleção de um subconjunto a partir de um conjunto de valores pré-definidos, por exemplo: botões de opção e caixas de seleção;

SimpleActivator: representa qualquer elemento capaz de reagir a eventos externos, tal como um *link* ou um botão de ação.

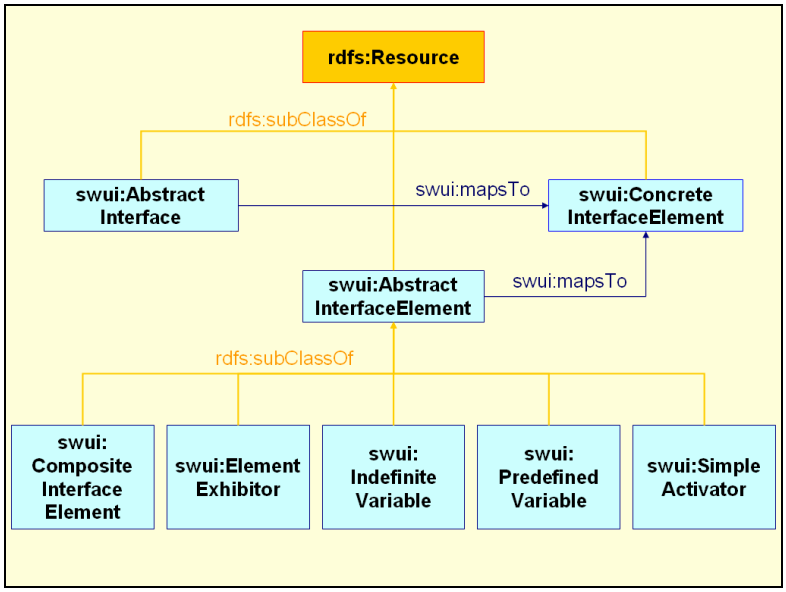

Figura 17 – Classes *AbstractInterface* e *AbstractInterfaceElement*.

[42] O modelo de dados da ontologia, expresso por meio da linguagem RDF, está disponível no Apêndice I. Os conceitos e propriedades da ontologia estendida fornecem o vocabulário para a Linguagem de Descrição de Interfaces proposta por esse trabalho. O prefixo swui nos nomes destes recursos é a abreviação para o namespace http://www.tecweb.inf.puc-rio.br/hyperde/browser/branches/hyperde_amluna/doc/rdf/swui.rdfs#.

AbstractInterface: a composição final de todos os elementos da interface abstrata.

No nível mais alto de abstração, os conceitos definidos pela ontologia preocupam-se apenas em classificar os elementos de interface quanto à sua funcionalidade. Para descrever a forma em que serão apresentados, específica do ambiente de implementação, a ontologia inclui o seguinte conceito:

ConcreteInterfaceElement: os elementos concretos de interface qualquer que seja o ambiente de implementação. A subclasse *DHTMLInterfaceElement* é uma especialização para os controles de interface implementados na plataforma DHTML que, por sua vez, possui as seguintes subclasses:

DHTMLRichControl: os controles de interface que combinam HTML DOM + Javascript + CSS. Inclui os fornecidos por bibliotecas Ajax, tais como Yahoo! User Interface Library (YUI);

HTMLTag: controles de interface fornecidos pela linguagem HTML.

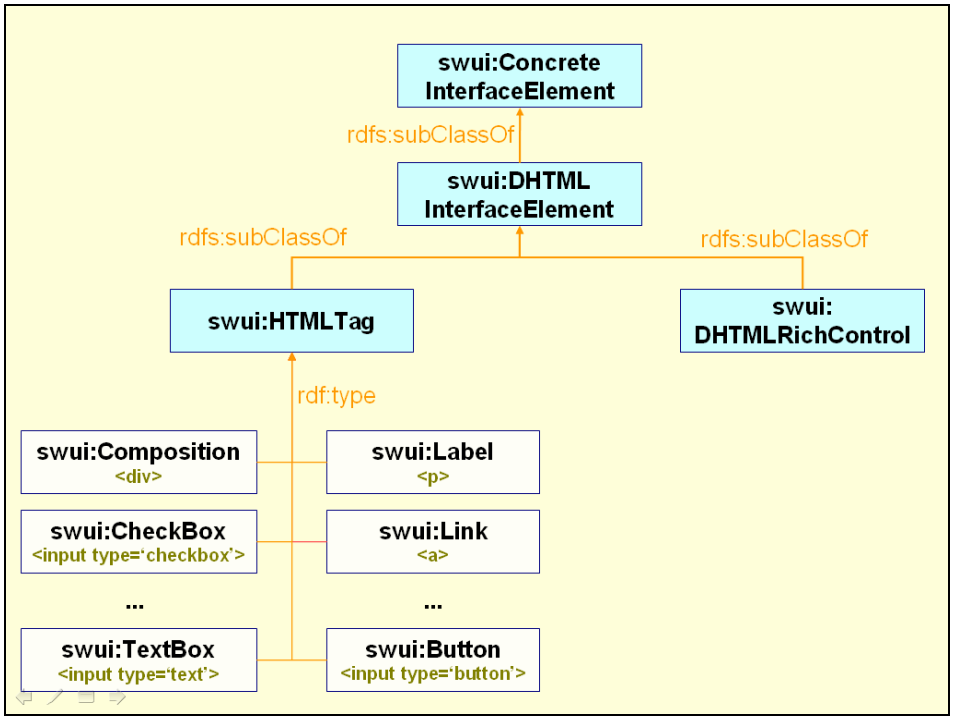

Figura 18 – Classes *ConcreteInterfaceElement*, *DHTMLInterfaceElement*, *HTMLTag* e *DHTMLRichcontrol*.

Para ilustrar a modelagem estrutural utilizando os conceitos da ontologia acima descrita, considere a seguinte interface, a qual chamaremos de “ProfessorView”:

A especificação abstrata está representada no diagrama a seguir: ele permite visualizar os tipos de *widget* escolhidos para representar os atributos do professor, bem como os relacionamentos do tipo composição entre esses *widgets*: a *AbstractInterface* de nome **ProfessorView** é composta pelo *CompositeInterfaceElement* de nome **Professor**, que, por sua vez, é a composição de outros dois *CompositeInterfaceElement*: **PersonalData** e **AcademicData**, e assim por diante.

Figura 19 – Interface *ProfessorView*

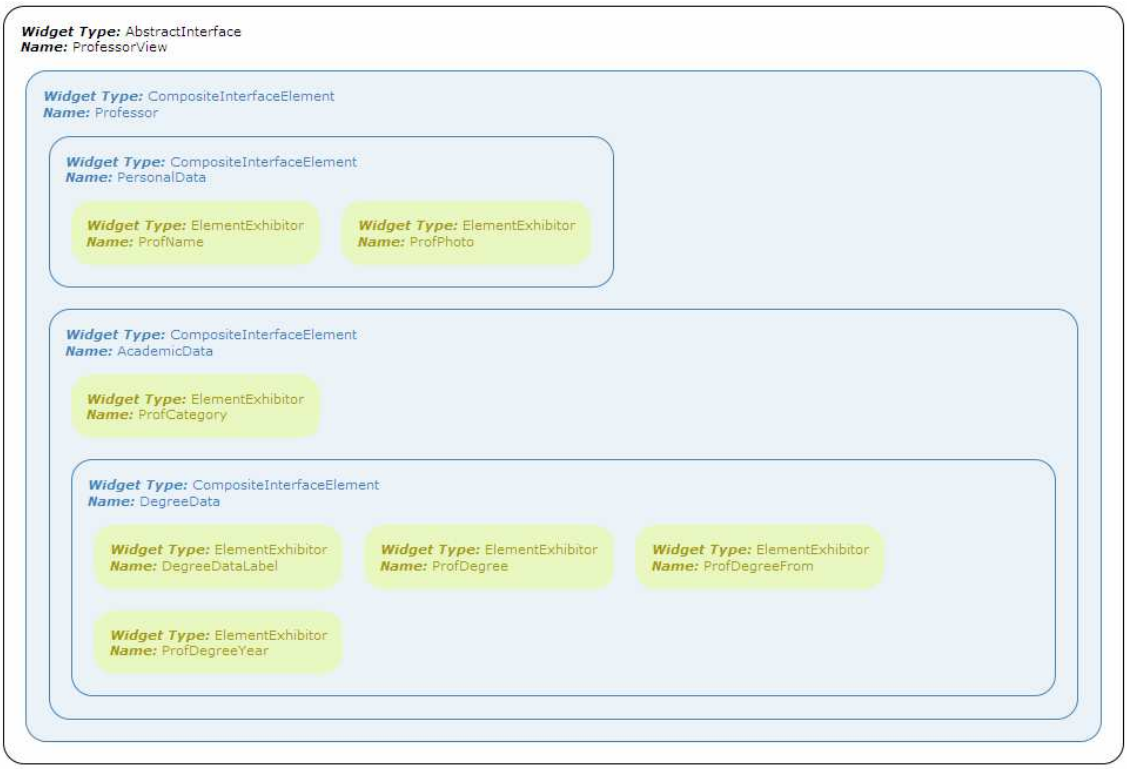

Figura 20 – Especificação abstrata da interface *ProfessorView*

3.3. Modelagem de Transições, Decorações e Eventos

A ontologia de *widgets* abstratos, tal como os *Abstract Data Views*, provê um formalismo para expressar a estrutura e o *layout* da interface. Todavia, para expressar o comportamento dos objetos de interface em reação aos eventos externos, o método

OOHDM emprega um formalismo adicional, conhecido como ADV-*Charts*, uma generalização dos *StateCharts*[43] e dos *ObjectCharts*[44]. O poder de expressividade dos ADV-*Charts* lhes permite modelar que transformações ocorrem na interface quando o usuário interage com a aplicação e como estas interações disparam a navegação. [38] Assim, para capturar os aspectos dinâmicos de um *widget*, a ontologia proposta para o método SHDM deve incluir alguns conceitos presentes no formalismo dos ADV-*Charts*, tais como estados, transições e eventos.

Fialho [41] desenvolve a conceitualização de Transições Navegacionais, ou mudanças de interface em sistemas hipermídia, como resultado da navegação entre objetos ou estados navegacionais. Também discute a utilização de animações em interfaces de sistemas computacionais, como forma enriquecer a experiência do usuário, e o uso de retórica visual, como forma de definir a ordem e as propriedades de efeito e duração das animações. Por fim, estende o uso da metodologia SHDM com a especificação de transições suaves ou animadas e o ambiente HyperDE com o Módulo de Interface e o Módulo de Transições, que dão suporte às etapas de modelagem de interfaces e de transições, respectivamente.

A ontologia que descreveremos a seguir lança mão de alguns dos conceitos definidos por Fialho [41] (transições e estruturas retóricas), generaliza o conceito de animações como decorações, e introduz o conceito de evento.

Transition: representa a alteração do estado da interface;

RhetoricalStructure: um conjunto de decorações, que ocorre durante uma transição ou em resposta a um evento. Agrupa as decorações semanticamente;

Decoration: é definida como a alteração (animada ou não) nos elementos da interface. É especializada nas seguintes subclasses:

InsertElement: uma decoração de entrada, para indica o surgimento de um novo elemento no estado de destino;

RemoveElement: uma decoração de saída, para indicar a remoção de um elemento pertencente ao estado origem;

MatchElements: uma decoração de manutenção, para indicar a presença do elemento nos dois estados da transição;

TradeElements: uma decoração de substituição, para indicar a existência de uma relação entre um elemento que pertence ao estado de origem e outro elemento que pertence ao estado de destino;

[43] *Statecharts* é um formalismo visual concebido por David Harel para especificar sistemas em tempo real do tipo reativo. Este tipo de sistemas, em contraste com os sistemas do tipo transformacional, devem continuamente reagir a estímulos externos e internos. [39]

[44] *Objectcharts* é uma notação que estende o formalismo dos *Statecharts*, e caracterizam o comportamento de uma classe como uma máquina de estados. [40]

EmphasizeElement: uma decoração de destaque, para indicar que uma ação está sendo realizada.

Ao passo que as decorações do tipo *Insert*, *Remove*, *Match* e *Trade* podem ocorrer durante transições entre interfaces, a classe *Emphasize* representa as decorações executadas exclusivamente em resposta a eventos de interface.

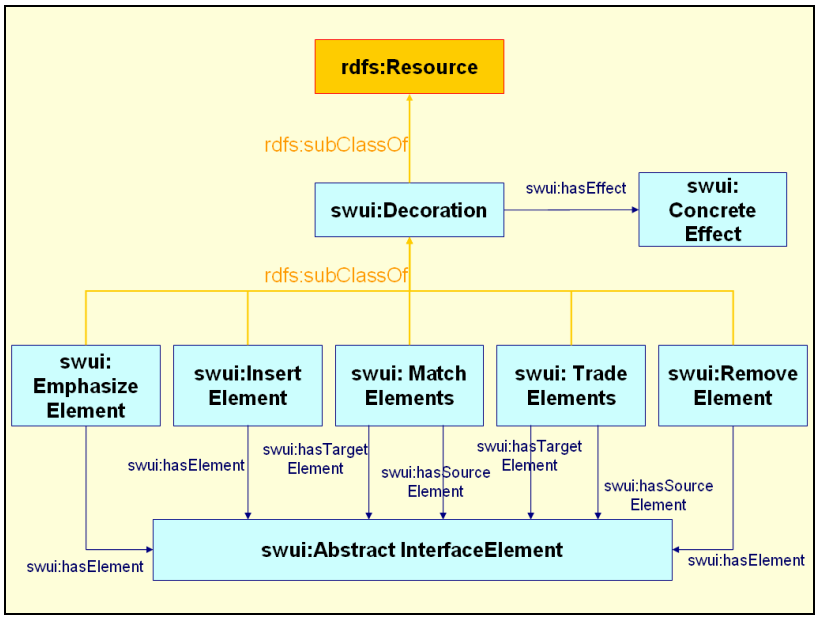

Figura 21 – Classe *Decoration*

ConcreteEffect: um efeito de decoração no ambiente de execução da interface concreta. Possui como subclasse:

DHTMLEffect: os efeitos disponíveis no ambiente de implementação DHTML, realizados por programação Javascript.

Event: a ocorrência de uma ação sobre os elementos de interface.

onActivate: indica que um *SimpleActivator* foi ativado;

onBlur: indica que um elemento de entrada de dados (*IndefiniteVariable* / *PredefinedVariable*) perdeu o foco;

onChange: indica que um elemento de entrada de dados (*IndefiniteVariable* / *PredefinedVariable*) teve seu conteúdo modificado;

onFocus: indica que um elemento de entrada de dados (*IndefiniteVariable* / *PredefinedVariable*) recebeu o foco;

onHover: indica que o ponteiro do mouse foi movimentado sobre um elemento;

onSelect: indica que o texto em um elemento do tipo *IndefiniteVariable* foi selecionado.

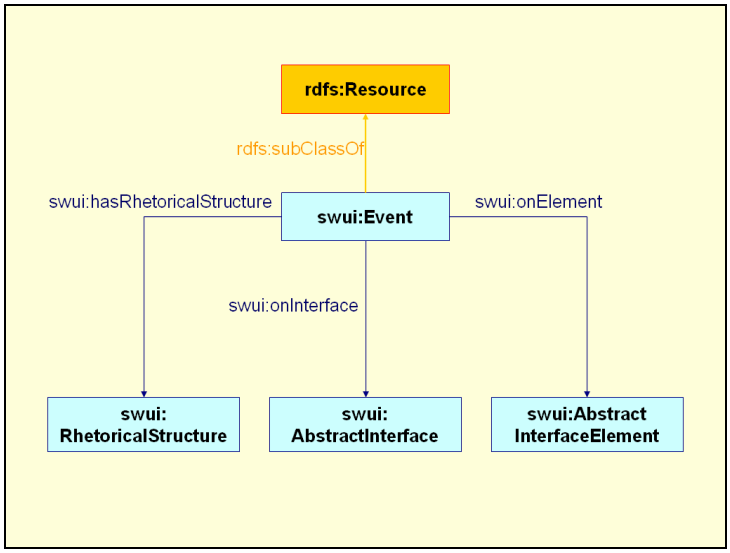

Figura 22 – Classe *Event*

A fim de descrever a ordem das estruturas retóricas entre si, e também a ordem das decorações dentro de cada estrutura retórica, adicionamos as seguintes classes à ontologia:

RhetStructSeq: a seqüência em que as estruturas retóricas ocorrem durante uma transição;

RhetStructSet: o conjunto de estruturas retóricas que ocorrem simultaneamente dentro de uma transição.

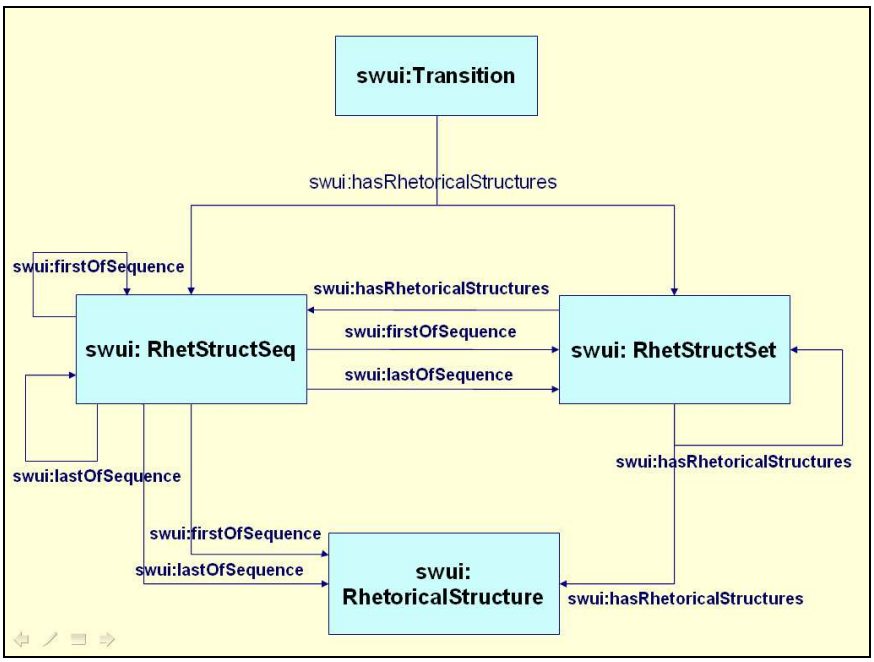

Figura 23 – Classes *Transition*, *RhetStructSeq* e *RhetStructSet*

DecorationSeq: a seqüência em que as decorações ocorrem dentro de uma estrutura retórica;

DecorationSet: o conjunto de decorações que ocorrem simultaneamente dentro de uma estrutura retórica.

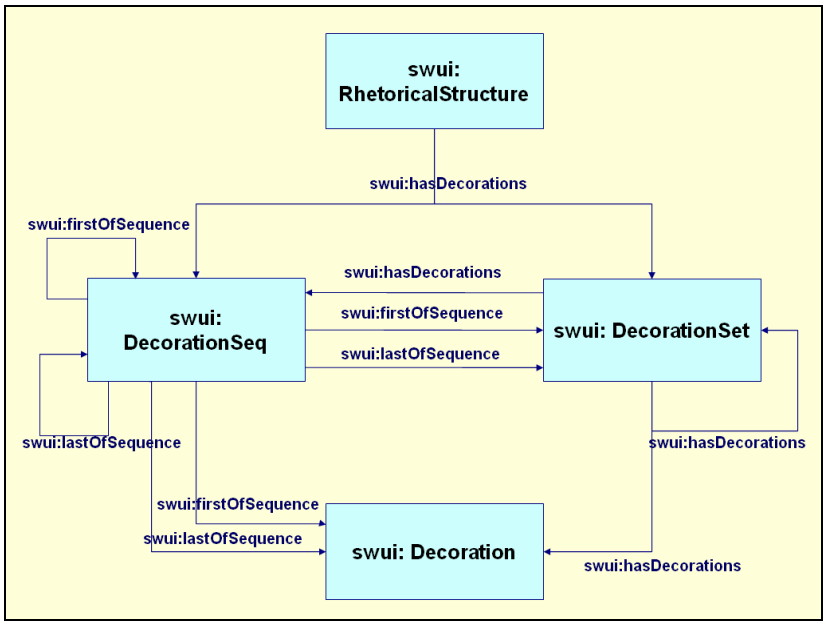

Figura 24 – Classes *RhetoricalStructure*, *DecorationSeq* e *DecorationSet*

3.4. Mapeamento Abstrato-Concreto

As classes descritas acima especificam a estrutura e o comportamento das interfaces em um nível abstrato. No entanto, duas dessas classes modelam as representações concretas correspondentes: *ConcreteInterfaceElement* e *ConcreteEffect*.

Através da propriedade *mapsTo*, as instâncias de *AbstractInterfaceElement* e *AbstractInterface* são mapeadas em instâncias de *ConcreteInterfaceElement*. Assim, é possível especificar como o *widget* abstrato deverá ser exibido, sem tornar a especificação abstrata dependente do ambiente de execução da interface, e, ao mesmo tempo, viabilizar a geração automatizada de interfaces concretas.

A fim de que garantir que a descrição abstrata seja traduzida em um trecho de código concreto válido, algumas regras de consistência são definidas. Elas restringem, por exemplo, que tipos de *widgets* abstratos um elemento concreto pode mapear (propriedade *legalAbstractWidgets* em *ConcreteInterfaceElement*).

Além disso, as instâncias da classe *HMTLTag* indicam, na propriedade *legalTag*, qual *tag* será usada para fazer a tradução do elemento. As instâncias de *HMTLTag* definidas na ontologia são:

a) Elementos concretos que podem disparar ações, incluindo navegação: *Button, Link*;

b) Elementos concretos que exibem textos: *Header1..6, ListItem, Label, Paragraph, Text*;

c) Elementos concretos que exibem figuras: *Image*;

d) Campos de entrada de informação: *CheckBox, ComboItem*, *RadioButton, TextArea, TextBox*;

e) Composições de elementos concretos: *Composition, ComboBox, Form, OrderedList, UnorderedList, Table, TableBody, TableCell, TableFooter, TableHeadCell, TableHeader, TableRow*.

A tabela a seguir informa quais elementos concretos do tipo *HTMLTag* podem representar cada tipo de elemento abstrato:

Widgets Abstratos	*Widgets Concretos*
AbstractInterface	Composition
CompositeInterfaceElement	ComboBox, Composition, Form, ListItem, OrderedList, Paragraph, Table, TableBody, TableCell, TableFooter, TableHeadCell, TableHeader, TableRow, UnorderedList
ElementExhibitor	Image, Label, ListItem, Paragraph, Header1..6, TableCell, TableHeadCell, Text
IndefiniteVariable	TextArea, TextBox
PredefinedVariable	CheckBox, ComboItem, RadioButton
SimpleActivator	Button, Link

Tabela 2 – Regras de mapeamento abstrato-concreto

O mapeamento entre *AbstractInterfaceElement* e *HTMLTag* é suficiente para reger a tradução automática da especificação abstrata em código HTML, pois a propriedade *legalTag* informa qual elemento HTML é a tradução do *widget* concreto. No entanto, as propriedades *tagAttributes* e *cssClasses*, em *AbstractInterfaceElement*, podem ser utilizadas para modificar ou acrescentar atributos à tradução *default*.

As figuras a seguir ilustram como as folhas de estilo podem ser usadas para aplicar uma formatação diferente à interface, junto com a especificação abstrata incluindo o mapeamento para os elementos da linguagem HTML.

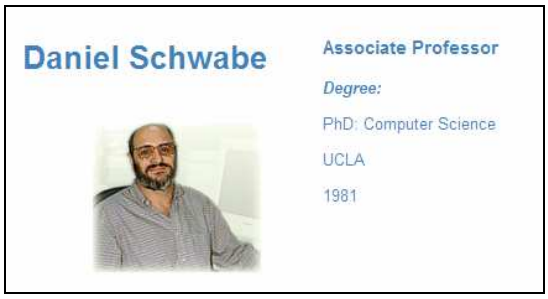

Figura 25 – Interface *ProfessorView* formatada com folha de estilo CSS

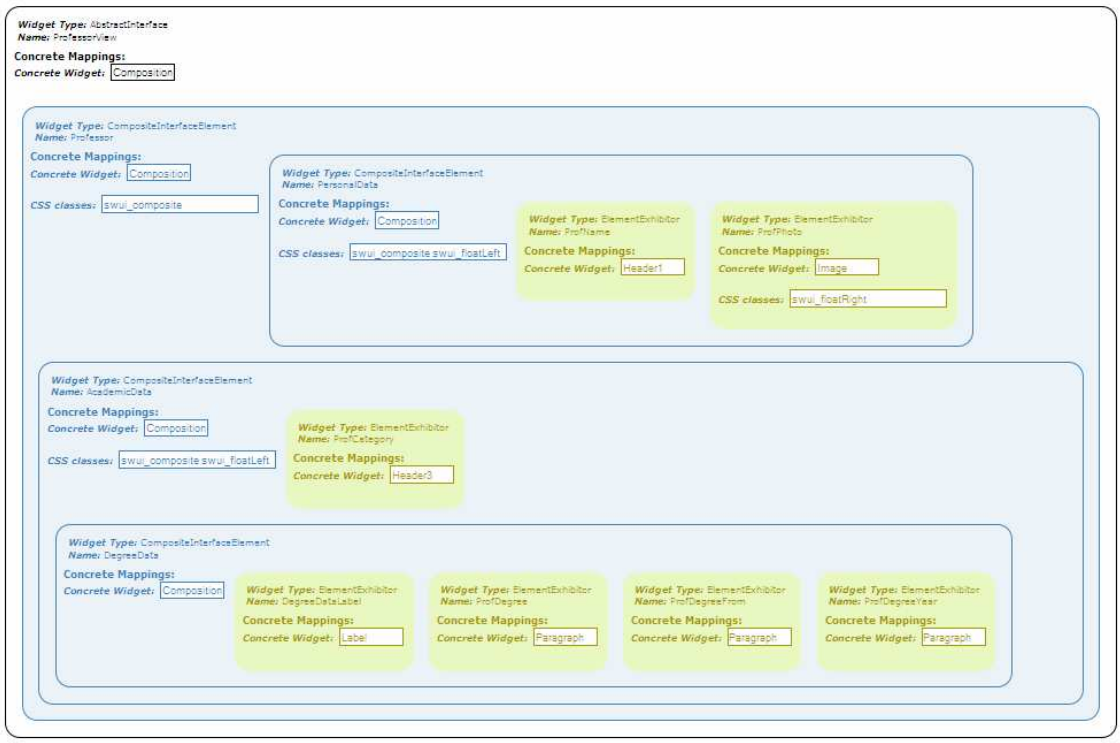

Figura 26 – Especificação abstrata da interface *ProfessorView*, incluindo os mapeamentos abstrato-concreto

O diagrama informa em quais instâncias de *HTMLTag* cada um dos *widgets* abstratos é mapeado (***Concrete Widget***) e os nomes das classes CSS aplicadas aos *widgets* concretos (***CSS classes***).

O mapeamento entre instâncias das classes *Decoration* e *ConcreteEffect* acontece por meio da propriedade *hasEffect*. Ao passo que *Decoration* classifica as decorações quanto à sua semântica (nível abstrato de especificação), *ConcreteEffect* descreve, no nível concreto, a forma como o usuário perceberá a alteração nos elementos de interface. Por exemplo, os diversos efeitos de decoração que podem ser aplicados em elementos HTML via Javascript/ CSS são modelados como instâncias da subclasse *DHTMLEffect*. A propriedade *parameters*, em *Decoration*, indica os parâmetros para o efeito aplicado pela decoração.

Os eventos reconhecíveis pelos elementos de interface variam segundo a sua representação concreta. Por isso, as instâncias da classe *Event* indicam, através da propriedade *concreteEvent*, o nome do evento no ambiente de implementação.

3.5. Mapeamento entre Visão e Modelo

A função dos objetos de interface é mediar a interação dos usuários com os objetos navegacionais. Por isso, a Ontologia de Descrição de Interfaces Ricas necessita prover vocabulário de mapeamento com a camada de modelo da aplicação.

Para satisfazer este requisito, definimos algumas propriedades dos *widgets*, com base nas primitivas do método SHDM.

fromAction: indica qual ação será executada quando um elemento do tipo *SimpleActivator* for ativado ou quando ocorrer um evento;

fromClass: indica à qual classe pertence o objeto navegacional representado pelo elemento do tipo *CompositeInterfaceElement*;

fromAttribute: indica qual atributo do objeto navegacional é representado pelo elemento abstrato.

No SHDM, os objetos navegacionais, também conhecidos como nós navegacionais, são elementos de **contextos de navegação**, conjuntos que dividem o espaço navegacional segundo diferentes critérios, por exemplo: o contexto de todos os alunos do Departamento de Informática ou o contexto dos alunos do curso de Mestrado. Os **índices de navegação**, por sua vez, representam estruturas de acesso que reúnem uma lista de ponteiros (*links*) para os nós navegacionais [38].

Desta forma, os conceitos da ontologia são utilizados com a semântica descrita a seguir:

AbstractInterfaceElement: definem os objetos de percepção que correspondem aos atributos dos nós navegacionais;

CompositeInterfaceElement: os elementos deste tipo representam instâncias das classes navegacionais quando constituem uma composição de elementos representando seus atributos;

SimpleActivator: estes elementos ativam as funcionalidades da aplicação, inclusive a de navegação. Um ativador dispara uma operação de navegação quando corresponde a um atributo navegacional (referenciado pela propriedade *fromAttribute*). Neste caso, o valor do atributo representa a URL de destino da navegação. Alternativamente, o ativador pode disparar uma operação no Modelo, indicada no valor da propriedade *fromAction* do elemento abstrato. Concordemente, as propriedades *fromAction* e *fromAttribute* são mutuamente exclusivas nos elementos do tipo *SimpleActivator*.

Event: eventos de interface gerados pelo usuário, tais como o clique de um mouse ou a digitação no teclado. Eventos podem ativar operações no Modelo, além da execução de estruturas retóricas (conjuntos de decorações).

Quando uma composição representa uma lista de objetos navegacionais, é possível especificar que a lista será ordenada por algum critério definido no Modelo. Esta é a semântica da propriedade *ordered* do elemento *CompositeInterfaceElement*.

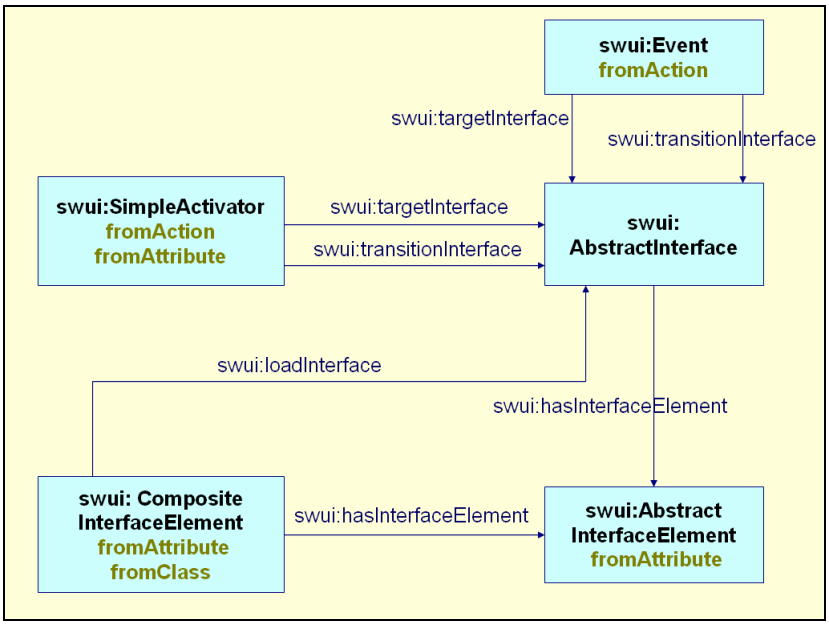

Figura 27 – Propriedades de mapeamento visão-modelo

A propriedade *targetInterface*, em uma instância de *SimpleActivator* ou *Event*, indica (opcionalmente) qual instância de *AbstractInterface* exibirá o resultado da operação disparada pelo ativador. Se o *designer* da aplicação desejar oferecer ao usuário uma transição "suave" entre as interfaces de origem e destino, ele poderá especificar que a interface de destino deverá ser carregada como um componente da interface de origem. A propriedade *loadInterface* indica que um *widget* do tipo *CompositeInterfaceElement* servirá tão-somente para aninhar uma segunda instância de interface abstrata. A interface aninhada, porém, permanecerá invisível até a ocorrência de algum evento que determine a sua transição de entrada, por exemplo, o retorno da execução de uma operação do Modelo. O objetivo de aninhar a interface de destino na interface de origem é poupar o usuário da sensação de "congelamento" da aplicação enquanto espera pela atualização de toda a página em seu navegador.

O mapeamento entre visão e modelo, como acabamos de descrever, permite que o estado da interface seja atualizado para refletir o estado da aplicação sempre que ocorrer uma transição navegacional ou uma operação tiver sido completada. O que dizer, entretanto, se desejarmos comunicar aos usuários não somente o *resultado* de uma operação, mas também o seu *progresso*? Este pode ser um requisito do projeto de interface em uma aplicação como o site Alibabuy.com, mencionado na introdução deste capítulo, caracterizado pela necessidade de se rastrear a execução de uma operação através de seus resultados parciais.

A propriedade *transitionInterface*, quando definida em uma instância de *SimpleActivator* ou *Event*, indica qual instância de *AbstractInterface* será utilizada para informar os usuários do progresso de uma operação no Modelo. Por definição, as

interfaces de transição (*transitionInterface*) devem existir como um componente da interface de origem; portanto, se faz necessário indicar, através da propriedade *loadInterface*, qual composição da interface de origem carregará a interface de transição.

A interface de transição será exibida após o elemento *SimpleActivator* ser ativado, mas antes de a operação ser concluída. O período em que a interface de transição está ativa corresponde ao tempo de processamento da requisição do usuário. O ambiente de execução que dá suporte às interfaces de transição será descrito em detalhes no capítulo 4.

Neste ponto, podemos retornar ao exemplo da interface “ProfessorView”, desta vez para mostrar, na especificação abstrata, o mapeamento entre os objetos de interface e os objetos navegacionais (***Model Binding Class*** e ***Model Binding Attribute***):

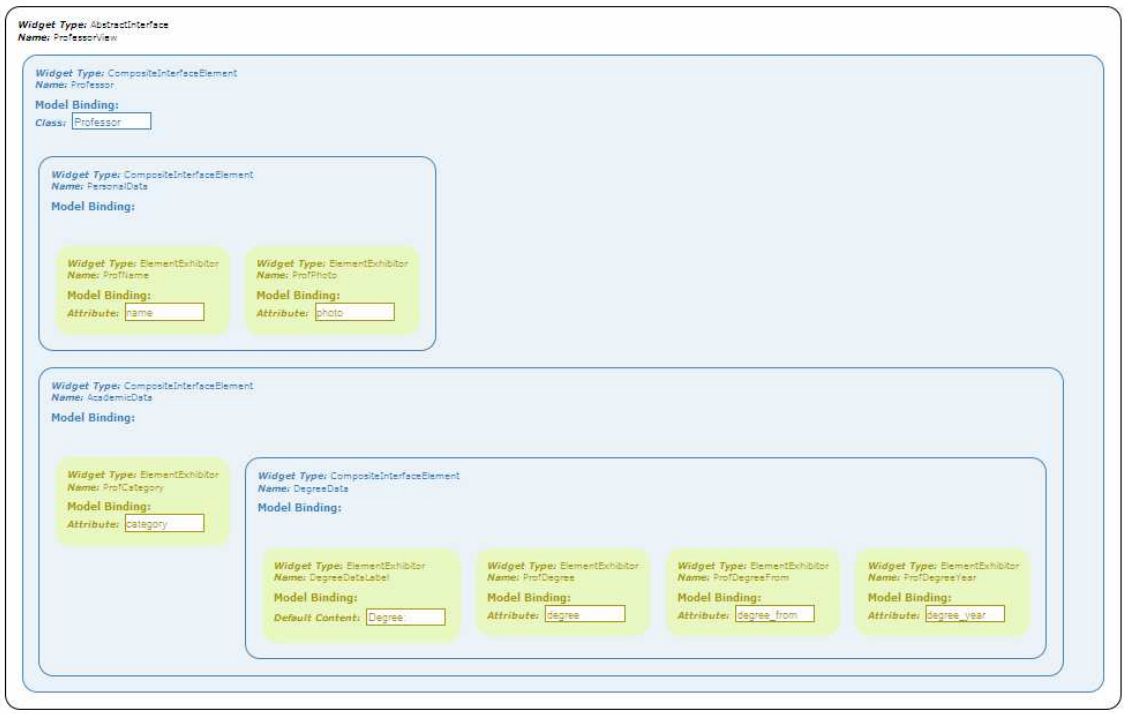

Figura 28 – Especificação abstrata da interface *ProfessorView*, incluindo os mapeamentos visão-modelo

Alguns objetos de interface não correspondem a objetos de navegação. Este é o caso de títulos de página ou de seção, por exemplo. Neste caso, o valor a ser exibido por tais *widgets* é especificado através da propriedade *defaultContent*.

3.6. Uma Linguagem Extensível

O desenvolvimento da Ontologia de Descrição de Interfaces Ricas foi pautado nos seguintes princípios de projeto: modelagem em dois níveis de abstração;

facilidade de extensão e expressividade suficiente para representar controles de interface ricos.

Optamos por seguir a filosofia proposta por Moura [2], a saber: separar a modelagem dos aspectos essenciais da interface das considerações relativas à plataforma tecnológica. A divisão da modelagem em níveis abstrato e concreto contribui fundamentalmente para tornar a linguagem extensível. Todavia, com o fim de facilitar a sua evolução, propusemos a seguinte alteração na Ontologia de *Widgets* Concretos: a inclusão de mais um nível de abstração, representado pela subclasse *DHTMLInterfaceElement*¸ que representa todos os controles de interface pertencentes à plataforma DHTML. (Figura 18)

Ao passo que *ConcreteInterfaceElement* é uma abstração válida para qualquer elemento que possa figurar nas interfaces de aplicações hipermídia, desejamos categorizar as suas instâncias segundo a solução tecnológica empregada. Por exemplo: a subclasse *HTMLTag* agrupa os elementos concretos providos pela linguagem HTML, enquanto que a subclasse *DHTMLRichControl* representa os controles de interface escritos com HTML, Javascript e CSS.

A linguagem pode ser estendida para representar outras classes de controles, por exemplo, os disponíveis nas plataformas Flex ou Silverlight, bastando, para isso, criar novas subclasses de *ConcreteInterfaceElement*.

O mesmo princípio aplica-se à classe *ConcreteEffect*. Ela é especializada na classe *DHTMLEffect*, que representa os efeitos aplicáveis a elementos do tipo *DHTMLInterfaceElement*, implementados por meio da linguagem Javascript. Se precisássemos definir efeitos em um ambiente de implementação que não a plataforma HTML, bastaria modelar uma abstração para eles como subclasse de *ConcreteEffect*.